Stable Roommates and Constraint Programming A first stab

Patrick Prosser

School of Computing Science, University of Glasgow

Abstract. In the stable roommates 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 specialised n-ary constraint that uses $O(n)$ additional space and establishes arc-consistency in $O(n^2)$ time. An empirical study is performed and it is observed that the n-ary constraint model can read in, model and output all matchings for an instances with $n = 1,000$ in about 2 seconds on current hardware platforms.

1 Introduction

In the stable roommates problem (SR) [6, 5] 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 stable marriage problem (SM) [2, 10, 3, 5, 11, 8] is a specialised 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 for some time now, probably the first efficient model being reported in 2001 [4], a 4-valued model in [9], a specialised binary constraint in [13] and an efficient n-ary constraint in [12]. And 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?". In this paper I partially answer this question. I present a remarkably simple constraint model for SR, using $O(n^2)$ constraints. A more compact and computationally efficient encoding is then proposed and it is demonstrated that this can solve instances of SR with up to 1,000 agents in about 2 seconds.

[1] For sake of brevity I will assume all agents as male, and hope that this offends no one.

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 [5] (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	1: 8 2 3 6 4 7	(1,7) (2,3) (4,9) (5,10) (6,8)
2 : 4 3 8 9 5 1 10 6 7	2: 4 3 8 9 5 1 10 6	(1,7) (2,8) (3,5) (4,9) (6,10)
3 : 5 6 8 2 1 7 10 4 9	3: 5 6 2 1 7 10	(1,7) (2,8) (3,6) (4,9) (5,10)
4 : 10 7 9 3 1 6 2 5 8	4: 9 1 6 2	(1,4) (2,8) (3,6) (5,7) (9,10)
5 : 7 4 10 8 2 6 3 1 9	5: 7 10 8 2 6 3	(1,4) (2,9) (3,6) (5,7) (8,10)
6 : 2 8 7 3 4 10 1 5 9	6: 2 8 3 4 10 1 5 9	(1,4) (2,3) (5,7) (6,8) (9,10)
7 : 2 1 8 3 5 10 4 6 9	7: 1 8 3 5	(1,3) (2,4) (5,7) (6,8) (9,10)
8 : 10 4 2 5 6 7 1 3 9	8: 10 2 5 6 7 1	
9 : 6 7 2 5 10 3 4 8 1	9: 6 2 10 4	
10 : 3 1 6 5 2 9 8 4 7	10: 3 6 5 2 9 8	

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 [5].

A quadratic time algorithm, essentially linear in the input size, was proposed in [6]. The algorithm has two phases. The first phase is a sequence of proposals, similar to that in the Gale Shapley algorithm [2], 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 integer arrays $pref$ and $rank$ such that $pref_{i,j}$ is the preference $agent_i$ has for $agent_j$ and $rank_{i,j}$ is the agent in the j^{th} position of $agent_i$'s preference list. Using sr10 as our example $pref_{3,5} = 1$ ($agent_5$ is $agent_3$'s first choice) $pref_{3,6} = 2$, $rank_{3,1} = 5$ and $rank_{3,2} = 6$. We also assume that we have constrained integer variables $agent_1$ to $agent_n$, each with a domain of preferences, initially $\{1 \dots n\}$. Consequently when $agent_i$ has been assigned the value x from its domain this means that $agent_i$ is matched with his x^{th} choice, and that is $agent_{i,j}$ where $j = rank_{i,x}$.

We can now make a declarative statement of the properties that a stable matching must have, and we can do this with three constraints. Given two agents, $agent_i$ and $agent_j$, 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 and is expressed by constraint (1) below. Also, when $agent_i$ is matched to $agent_j$ then $agent_j$ is matched to $agent_i$, and this is expressed by constraint (2). Finally we have the symmetric constraint to constraint (1) namely that when $agent_j$ is matched to an agent that he prefers less than $agent_i$, $agent_i$ must match up with an agent he prefers to $agent_j$, and this is constraint (3).

$$\forall_{i\in[1..n-1]}\forall_{j\in[i+1..n]}\ agent_i > pref_{i,j} \implies agent_j < pref_{j,i} \quad (1)$$
$$\forall_{i\in[1..n-1]}\forall_{j\in[i+1..n]}\ agent_i = pref_{i,j} \iff agent_j = pref_{j,i} \quad (2)$$
$$\forall_{i\in[1..n-1]}\forall_{j\in[i+1..n]}\ agent_j > pref_{j,i} \implies agent_i < pref_{i,j} \quad (3)$$

This constraint is shown pictorially in Figure 2. The three constraints are shown for $agent_1$ and $agent_3$ 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 middle picture is for constraint (2) where $agent_1$ is matched to $agent_3$, both taking the pair of red values. The bottom picture is for constraint (3) where $agent_3$ is matched in the green region to an agent he prefers less than $agent_1$ consequently $agent_1$ must be matched in his green region also, to an agent he prefers to $agent_3$.

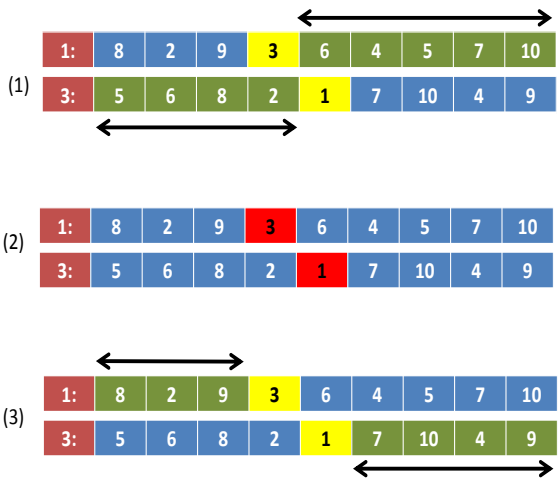

Fig. 2. A pictorial representation of the three constraints acting between $agent_1$ and $agent_3$.

This constraint is not new, having been proposed for SM. Establishing arc-consistency [7, 14] in this 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 [4], one using 4-valued variables [9] and one using a specialised n-ary constraint [12].

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 7 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 $agent_1 \leftarrow 1$ (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.

The model was implemented in the choco constraint programming toolkit [1] using Java. The code for this simple model is shown in Listing 1. The first thing to note is that everything is zero-based, such that the first agent is $agent_0$ and the last $agent_{n-1}$. Lines 14 to 27 read in the problem instance, building the arrays $pref$ and $rank$. To address SR with incomplete lists we add i to the end of $agent_i$'s preference list (lines 24 and 25) such that an unmatched agent is matched to itself. The constraint model is produced in lines 29 to 41 with constraint (1) posted in lines 33 and 34, constraint (2) in lines 35 and 36, and constraint (3) in lines 37 and 38. In lines 43 to 49 the choco toolkit searches for a first solution and prints it out.

```
1  import java.io.*;
2  import java.util.*;
3  import static choco.Choco.*;
4  import choco.cp.model.CPModel;
5  import choco.cp.solver.CPSolver;
6  import choco.kernel.model.Model;
7  import choco.kernel.solver.Solver;
8  import choco.kernel.model.variables.integer.IntegerVariable;
9
10 public class StableRoommates {
11
12     public static void main(String[] args) throws IOException {
13
14         Scanner sc   = new Scanner(new File(args[0]));
15         int n        = sc.nextInt();
16         int[][] pref = new int[n][n];
17         int[][] rank = new int[n][n];
18         for (int i=0;i<n;i++){
19           for (int k=0;k<n-1;k++){
20               int j = sc.nextInt() - 1;
21               pref[i][j] = k;
22               rank[i][k] = j;
23            }
24           rank[i][n-1] = i;
25           pref[i][i]   = n-1;
26         }
27         sc.close();
28
29         Model model             = new CPModel();
30         IntegerVariable[] agent = makeIntVarArray("agent",n,0,n-1);
31         for (int i=0;i<n-1;i++)
32           for (int j=i+1;j<n;j++){
33               model.addConstraint(implies(gt(agent[i],pref[i][j]),
34                                           lt(agent[j],pref[j][i])));
35               model.addConstraint(ifOnlyIf(eq(agent[i],pref[i][j]),
36                                            eq(agent[j],pref[j][i])));
37               model.addConstraint(implies(gt(agent[j],pref[j][i]),
38                                           lt(agent[i],pref[i][j])));
39           }
40         Solver solver = new CPSolver();
41         solver.read(model);
42
43         if (solver.solve().booleanValue())
44           for (int i=0;i<n;i++){
45               int j = rank[i][solver.getVar(agent[i]).getVal()];
46               if (i<j) System.out.print("("+ (i+1) +","+ (j+1) +") ");
47            }
48         System.out.println();
49     }
50 }
```

Listing 1. A simple encoding, StableRoommates.java

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

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 30 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. In fact, it does so in less time.

4 A more efficient model

Our constraint model can be made more computationally efficient by adopting, and modifying, the models in [4, 9]. However, these models are bulky and quickly exhaust memory on relatively modest sized instances of SM (n = 400 in [12]). Therefore we propose an n-ary SM constraint (SMN), similar to that proposed in [12], 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 14 to 27 of Listing 1). The means of reducing the computational cost is by eliminating the redundancies in the simple model 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 an $agent_i$, $O(n)$ 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 specialised 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 $agent_1$ to $agent_n$, that an *agent* domain is initially a list of preferences $\{1 \dots n\}$, 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 $agent_i$ 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 17 and 18 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 $agent_i$ 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 $j = rank_{i,x}$, 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 $agent_i$ 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 $agent_i$ 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 $O(n^2)$.

Algorithm 1: deltaMin (awakeOnInf in Listing 2).

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

Algorithm 2: deltaMax (awakeOnSup in Listing 2).

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

removeValue(i,x) (Algorithm 3) The value x has been removed from the domain of $agent_i$ consequently the corresponding agent ($agent_j$ where $j = rank_{i,x}$) 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 ← rank_{i,x}
4     remove(agent_j, pref_{j,i})
```

instantiate(i) (Algorithm 4) The variable $agent_i$ has been assigned the value y (line 3) and corresponds to being matched to $agent_j$ where $j = rank_{i,y}$. 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). We then ensure that the matching is symmetric: if $agent_i$ is matched to $agent_j$ then

$agent_j$ is matched to $agent_i$ (lines 10 and 11). Finally we update the upper and lower bounds for the domain (lines 12 and 13). **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 instantiated with a value at most once during propagation. Consequently, over all n agents this has a cost of $O(n^2)$

Algorithm 4: instantiate (awakeOnInst in Listing 2).

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

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 initialised (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 [6]. **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 ← n
6      for i ← 1 to n do
7          deltaMin(i)
```

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

Listing 2. SRN.java

5 Empirical Study

Experiments were performed over random problems, with random instances generated using Algorithm 6, where lines 6 to 8 perform a Knuth-shuffle. The list is then outputted, omitting the current agent (line 9). Experiments were performed 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 investigate our three models: (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.

Algorithm 6: randomRoommates, generate an instance of size n.

```
1 randomRoommate(int n)
2 begin
3     print(n)
4     prefList ← {1...n}
5     for i ← 1 to n do
6         for j ← n downto 2 do
7             k ← random(j)
8             swap(prefList_j, prefList_k)
9         print(prefList \ i)
```

Figure 4 presents six scatter plots. Plots on the left are for $10 \leq n \leq 100$ and on the right $100 \leq n \leq 1,000$ (and or $n > 100$ we omit SR). The top row of graphs gives the time to build the model, the middle row the time to enumerate all matchings and the bottom row gives the total time where total time is the time to read the instance, model, solve and output the run time statistics. In all scatter plots time is measured in milliseconds. What we see is that build time for SR and SRB are pretty much the same and are growing quadratically (as expected) whereas SRN appears to grow linearly. The solve time for SRB is significantly faster than SR but these dwarf that of SRN: when $n = 1,000$ SRB takes minutes whereas SRN takes seconds. The total run times shows that SR is does not scale beyond $n = 100$ and at $n = 1,000$ SRN typically takes 4 minutes whereas SRN takes 2 seconds, i.e. SRN is two orders of magnitude faster.

In Table 1 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 $100 \leq n \leq 1,000$. Also tabulated is the average number of nodes reported by the choco toolkit where a node is a decision made, and that decision might be one that leads to a failure and a backtrack. The last column is the proportion of instances that had matchings. What is most interesting, from the constraint programming perspective, is that there are so few nodes and this suggests that backtracking is in some sense bounded.

We now investigates the problem (given n what proportion of instances have matchings?) and our best run times (using SRN, finding a first matching)

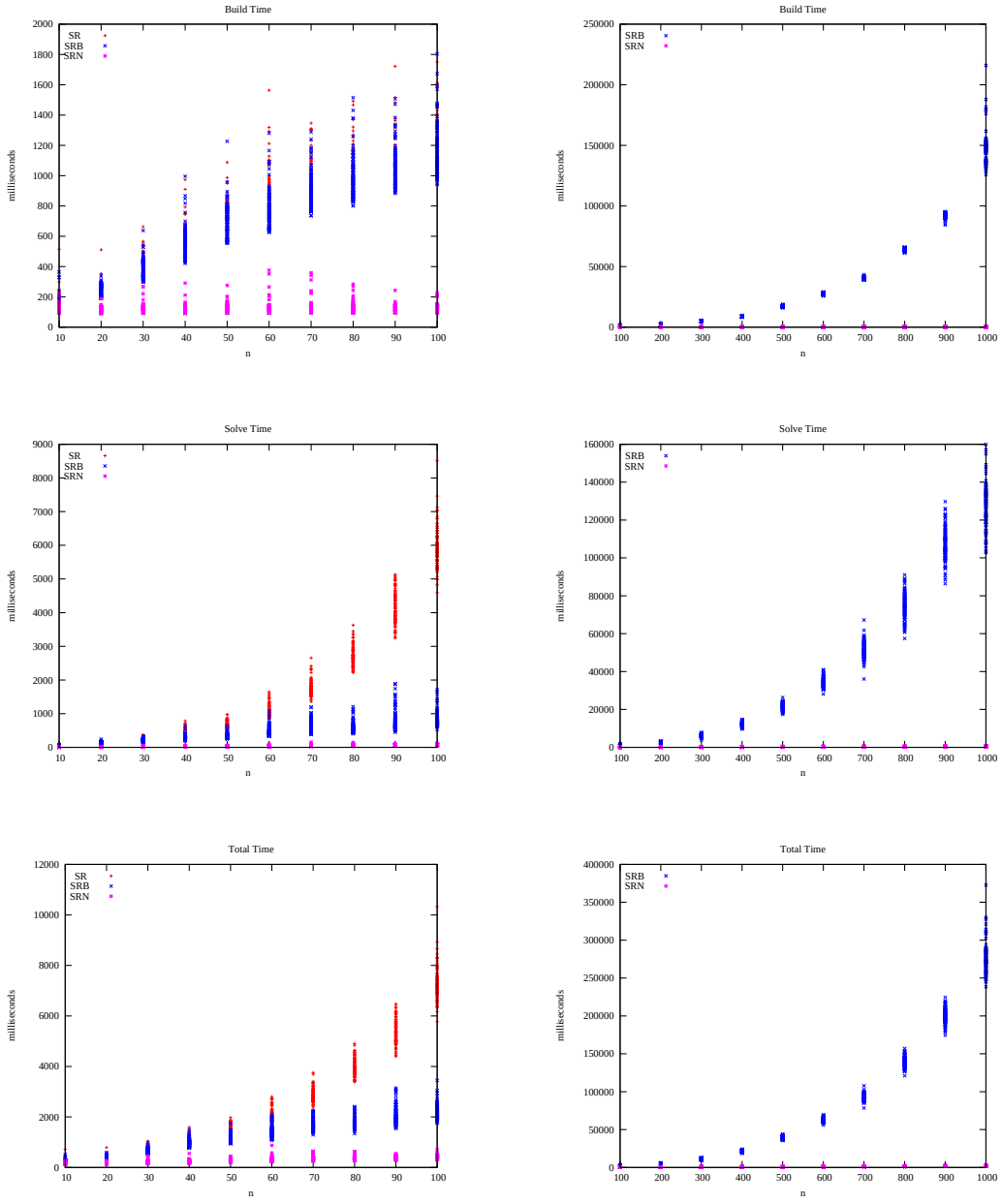

Fig. 4. Performance of the models: top row is build time, middle row solve time, bottom row is total time.

n	cpu time	nodes	matched
100	0.423	4	0.63
200	0.511	6	0.52
300	0.645	7	0.53
400	0.768	7	0.38
500	0.950	7	0.45
600	1.094	7	0.41
700	1.290	7	0.42
800	1.555	8	0.44
900	1.786	8	0.39
1,000	2.046	8	0.40

Table 1. Proportion of instances with solutions and average run times (figures in brackets are from [6]). Sample size of 1,000

against the results in [6]. Shown in Table 2 are the proportion of instances with matchings, for $n \in \{10 \dots 90\}$ with a sample size of 1,000, and average cpu time measured in seconds. The time includes reading in the instance, building the model, finding a matching and printing out run time statistics. The figures in brackets are those reported in [6], where a PDP11/44 was used and the algorithm was coded in Pascal, 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 $40 \leq n \leq 90$. Table 2 shows that our best constraint encoding, allied to nearly 30 years of hardware advance, has resulted in an improvement of little over a factor of seven. A back of the envelope calculation suggests that the hardware used in the study today is about 15,000 times faster than that used in 1985. So why are we only 7 times faster? Have we squandered all those hard won hardware advances? Part of the explanation is that the distance from the application to the hardware has been ever increasing. And this has been done to increase the ease of use of the machine. In this case the journey to the hardware goes through my program then on to the choco constraint programming toolkit (allowing us to more easily model and solve combinatorial problems), then to java and its virtual machine (giving us all that we expect of a modern programming language, i.e. ease of use, richness, portability) and then finally to the hardware. Nevertheless it is hard to believe that this has used up a factor of 2,000 of the 15,000 times speed up. Of course, part of that cost could be due to the programmer.

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.

n	Proportion with matchings	Average cpu time
10	0.889 (0.868)	0.168 (0.142)
20	0.834 (0.815)	0.178 (0.374)
30	0.781 (0.766)	0.199 (0.727)
40	0.736 (0.745)	0.226 (1.183)
50	0.727 (0.710)	0.258 (1.760)
60	0.704 (0.725)	0.285 (2.367)
70	0.706 (0.670)	0.317 (3.110)
80	0.670 (0.675)	0.339 (3.964)
90	0.670 (0.690)	0.377 (4.875)

Table 2. Proportion of instances with solutions and average run times (figures in brackets are from [6]). Sample size of 1,000

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 specialised 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.

So, why is it "A first stab"? This is, I believe, the first constraint programming solution to this problem, and there is lots of work still to do. The first piece of work is to explain the relationship between the rotations in [6] and maintaining arc-consistency in the model during search. Clearly, the constraint model can fail during search, but are these failures bounded by some polynomial, i.e. can we prove that that exponential behaviour will not occur? Then there is the issue of how we might put the constraint model to good use, i.e. can we have a richer model where stable matching is only one of possibly many criteria that must be satisfied? And then there is the direct step into hard variants of stable roommates, by enhancing the n-ary constraint to deal with incomplete lists and ties in preference lists. And of course, there is the goal of winning back some of the speed up from nearly three decades of hardware advances.

References

1. choco constraint programming system. http://choco.sourceforge.net/.
2. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.
3. D. Gale and M. Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11:223–232, 1985.
4. I.P. Gent, R.W. Irving, D.F. Manlove, P. Prosser, and B.M. Smith. A constraint programming approach to the stable marriage problem. In *CP'01*, pages 225–239, 2001.

5. D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, 1989.
6. Robert W. Irving. An efficient algorithm for the "stable roommates" problem. *J. Algorithms*, 6(4):577–595, 1985.
7. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
8. David Manlove. *Algorithmics of Matching under Preferences*, volume 2 of *Theoretical Computer Science*. World Scientific, 2013.
9. D.F. Manlove and G. O'Malley. 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)*, pages 10–17, 2005.
10. A.E. Roth. 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.
11. A.E. Roth and M.A.O. Sotomayor. *Two-sided matching: a study in game-theoretic modeling and analysis*, volume 18 of *Econometric Society Monographs*. Cambridge University Press, 1990.
12. C. Unsworth and P. Prosser. 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.
13. C. Unsworth and P. Prosser. A specialised binary constraint for the stable marriage problem. In *SARA05*, 2005.
14. Pascal van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.