Stable roommates problem solver

Filip Bártek

September 28, 2014

1 Problem definition

Let's have n participants. Each participant knows some of the participants, let's call these her potential partners. Each participant has a linear ordering of her potential partners according to preference.

Note that a participant may or may not consider herself a potential partner, i.e. the relation of potential partnership needn't be irreflexive.

A matching is an equivalence relation on participants that has classes of size at most 2, i.e. assigns each participant one or none partner. Matching must assign a potential partner to each of the participants.

An instability in a matching is a pair of participants each of whom prefers (according to their personal preference relations) the other to their current partner.

A stable matching is a matching that doesn't admit an instability.

In stable roommates problem (SRP), given preferences of each participant, the task is to find a stable matching.

1.1 Perfect matching

A perfect matching is a matching in which every participant is assigned somebody else.

Once we can solve general SRP, we can force a perfect matching by making sure that no participant considers herself a potential partner.

2 Constraint model

In this section we'll describe a model for SRP instance on n participants.

We'll use the symbol $\mathbb{N}_n$ to denote the set $\{1, \ldots, n\}$. Each participant is uniquely identified by a number from $\mathbb{N}_n$.

Note that the choice of variables and constraints corresponds to the capabilities of `clpfd`, a library that is used prominently in the implementation of the solver.

2.1 Variables

Problem instance We'll represent an instance of SRP with n participants as a collection of preference lists $P = (P_1, \ldots, P_n)$. P_i is a preference list expressing preferences of participant i. It's a k_i-tuple of participant identifiers (i.e. numbers from $\mathbb{N}_n$). All potential partners of participant i are listed in P_i in order of decreasing desirability without duplicities.

Problem solution $m : \mathbb{N}_n \to \mathbb{N}_n$ assigns each participant her partner.

Auxiliary variables Let $s : \mathbb{N}_n \times \mathbb{N}_n \to \mathbb{N}_n$ be a score function. $s(i, j)$ represents how desirable participant j is according to participant i. The lower the score, the more desirable participant j is.

s is defined uniquely for a given instance P. For every participant i:

- j-th potential partner is assigned score j
- every participant which is not a potential partner is assigned a score n

As a convenience, let $\bar{s}(i)$ denote the score that participant i assigns to her partner.

2.2 Constraints

Since the score function s depends uniquely and trivially to the problem instance P, construction of s from P is realized using standard Prolog code (without the use of `clpfd`) and as such, I'll leave out formal definition of the corresponding constraint.

Similarly, the properties of P are not enforced formally so I'll leave out their formal definitions. Informal description of these properties are available in the previous section.

The following constraints constrain the solution m based on P and s:

Constraint	Explanation
$\forall i. m(i) \in P_i$	Matching satisfies potential partners
$\forall i, j. m(i) = j \Leftrightarrow m(j) = i$	Matching is symmetric
$\forall i. \bar{s}(i) = s(i, m(i))$	$\bar{s}$ corresponds to s and m
$\forall i, j. \bar{s}(i) \leq s(i, j) \vee \bar{s}(j) \leq s(j, i)$	All pairs are stable

Note especially the last of these constraints as it captures the essence of the problem, that is the requirement of stability. An instability occurs when a pair of participants prefer each other to their partners. The constraint ensures that in each pair of participants, at least one of them prefers her partner (i.e. assigns her a lower score) to the other participant.

3 Implementation

I've implemented the constraint model introduced in the previous section using SICStus Prolog and its `clpfd` library. The implementation is available in the attached file `srp.pl`.

3.1 Usage

The main entry point is the predicate `srp/2`. Its typical usage is `srp(Preferences, Matching)`, where `Preferences` is a fully instantiated list of ordered lists of potential partners and `Matching` is unbound. If a solution of SRP instance described by `Preferences` exists, it is unified with `Matching`. Otherwise `srp` fails.

Internally, `srp` uses `clpfd` library to search for the solution. The searching procedure executed by `clpfd:labeling/2` can optionally be customized by passing a list of options in the first argument of `labeling`. This argument of `labeling` is exposed in the third argument of the extended `srp/3`. For example, one may execute `srp(Preferences, Matching, [ffc, assumptions(K)])`.

3.2 Example instances

Examples of problem instances for the solver are included in file `srp.pl` in unit tests for `srp/2` (in a block starting with statement `:- begin_tests(srp_2).`).

3.3 Correctness

Correctness of the implementation was assessed using a set of tests on small manually constructed and solved SRP instances (up to size 8).

The tests are bundled in the source file `srp.pl` and can be accessed using `plunit` library interface (typically by executing `plunit:run_tests.`).

3.4 Performance

I've examined the performance of the solver using various problem instances (esp. of various sizes) and search strategies.

3.4.1 System configuration

The performance measurements were performed on a computer with the following configuration:

Model	HP Compaq 6510b
CPU	Intel Core2 Duo T8100 2.10 GHz
RAM	2.50 GB
Operating system	Windows Vista Business 32-bit
SICStus Prolog	4.2.3

3.4.2 Instance generation

For the purposes of measuring performance of the solver, instances of SRP were generated randomly in the following manner: for a given n, each of the n participants' preference lists is a random permutation of $\mathbb{N}_n$ such that every possible permutation occurs with equal probability.

3.4.3 Measurement procedure

The following instance sizes (values of n) were examined: 0, 1, 2, 4, 8, 16, 24, 32, 40, 48, 64, 72, 80 and 88. I didn't proceed past the size 88 because at $n = 96$ the solver failed because of insufficient memory.

For every examined value of n, 10 random instances were generated.

For sizes up to 64, each of these 10 instances was solved using each of the three relevant basic search strategies offered by `clpfd:labeling/2`:

- `leftmost`,
- `ff` (first-fail) and
- `ffc` (most constrained).

Since `leftmost` performed time-wise significantly worse than `ff` and `ffc` (see figure 5), I left it out of measurements on sizes larger than 64.

Two values were measured: *time to find a solution* and *number of assumptions taken in finding the first solution.*

Time was measured using SICStus Prolog built-in predicate `statistics/2` with keyword `runtime`. The solving procedure was repeated 100 times on instances of size at most 32, 10 times on instances of size between 40 and 56 and 1 time on instances of size above 56. (see figure 1). The running times discussed further on are averages across these executions.

Assumptions are the choices made by `clpfd:labeling/2` during the search for a solution. Their number is extracted from the search procedure using the option `assumptions(K)`.

In all cases, the solver was only run until it found any solution or concluded that no solution exists for the given instance. Rationale: since there is no defined distinction between the valid solutions, I assume that the user of the solver will typically be satisfied with any valid solution, or knowledge of its absence.

The numbers of assumptions were only measured on soluble instances. For all instance sizes, at least 5 of the 10 generated instances had a solution (see figure 2).

3.4.4 Results

The raw measured data can be examined in the attached file `data.csv`.

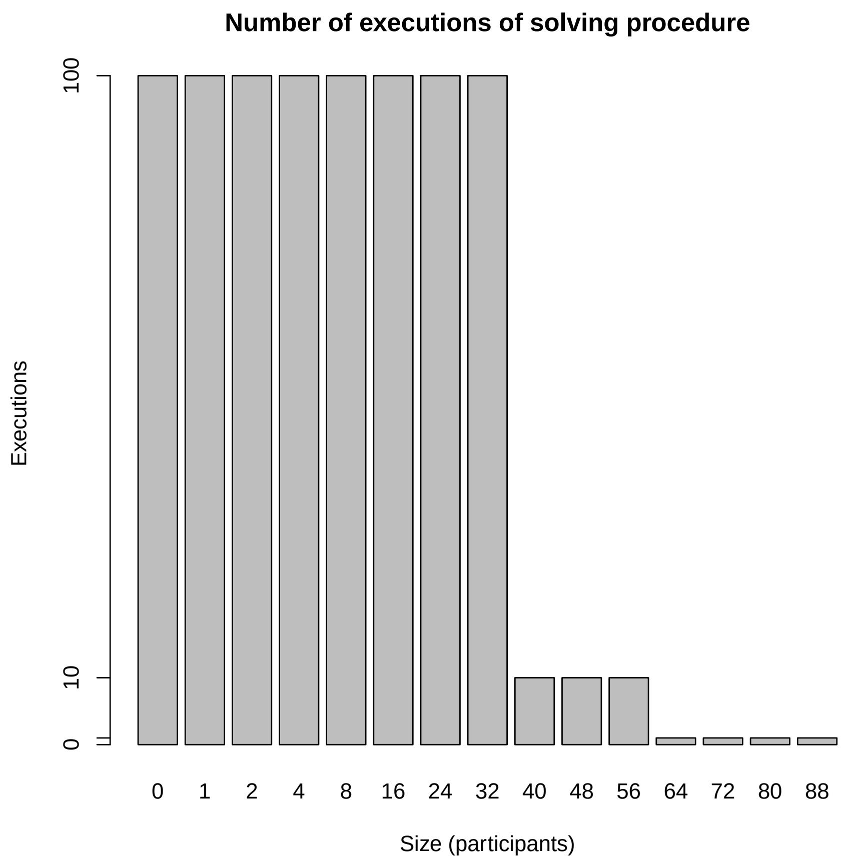

Figure 1: Number of executions of solver for time measurement

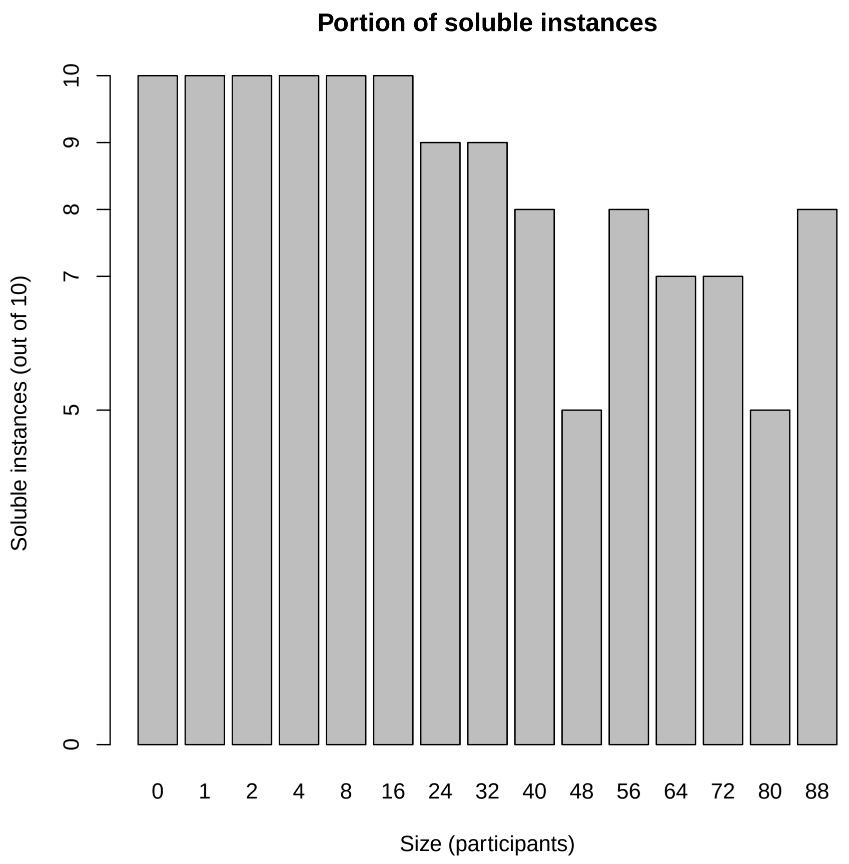

Figure 2: Number of soluble instances

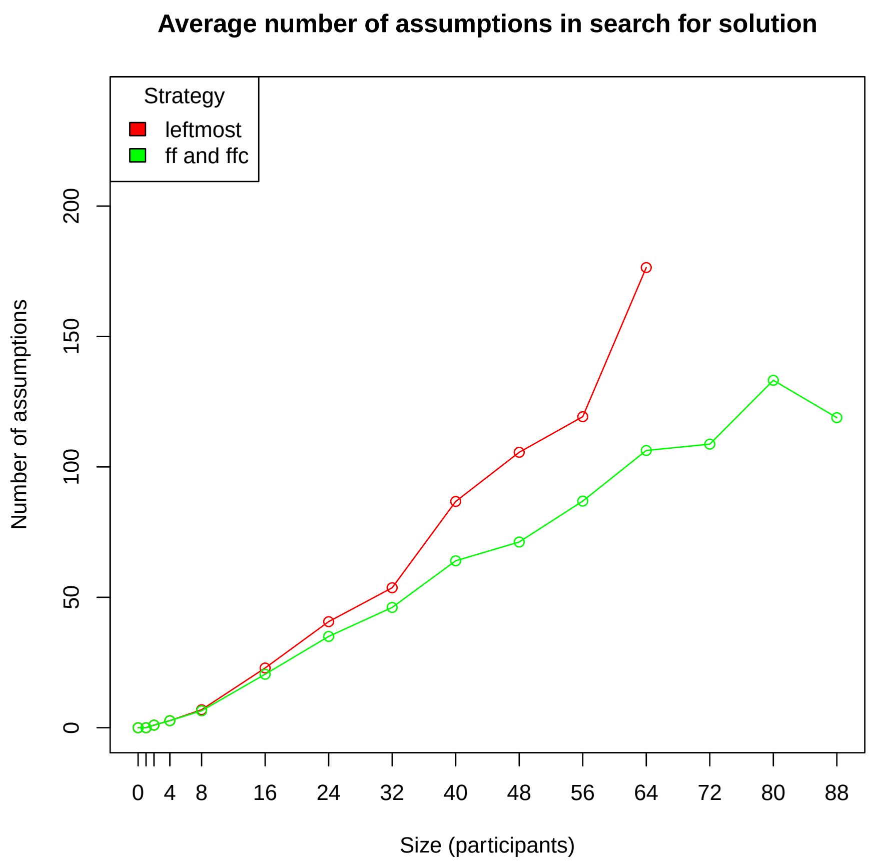

Figure 3: Average number of assumptions

Assumptions The average numbers of assumptions made in the search for the first solution are shown in figure 3. The averages run across the soluble instances of a given size (at least 5 and up to 10, as indicated in 2).

Note that the numbers of assumptions happen to match for strategies `ff` and `ffc` (on an instance-to-instance basis, not only the averages). Since `ffc` chooses the most constrained variable where `ff` simply chooses the first declared variable, I suppose that the implementation simply happens to declare the variables in descending order of number of suspended constraints.

In the chart you can see that the number of assumptions rises with instance size and rises more quickly for the strategy `leftmost`. This is to be expected since `ff` and `ffc` use more elaborate heuristics for lowering the expected number of assumptions.

You can see the dispersion of number of assumptions among instances of a

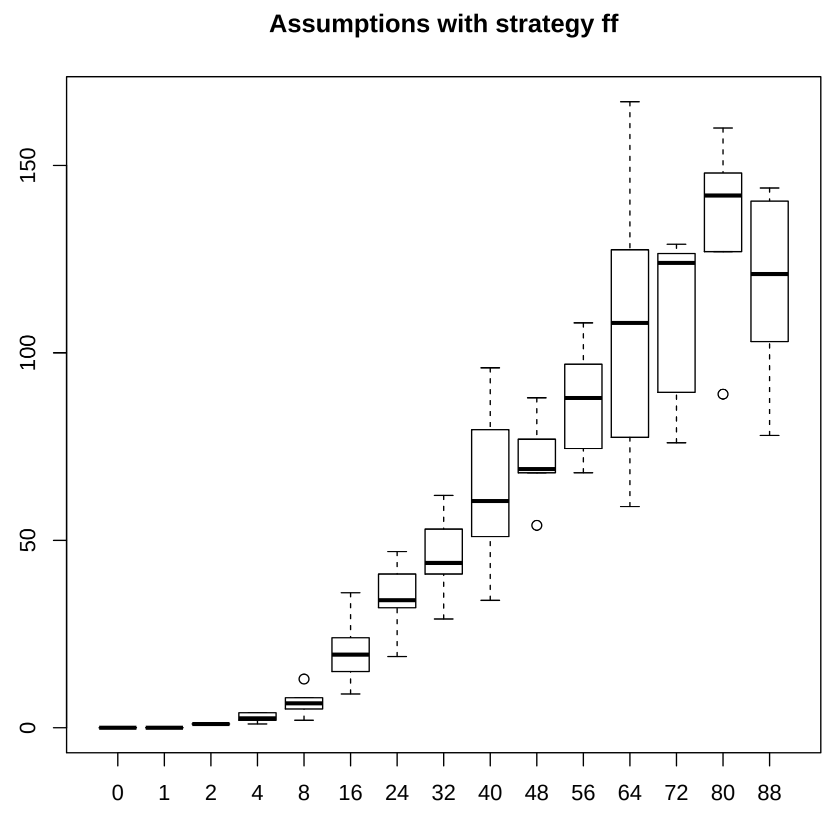

Figure 4: Dispersion of number of assumptions with search strategy `ff`

given size for search strategy `ff` in figure 4.

Running times Average running times for various instance sizes and search strategies are shown in figure 5. The averages run across 10 instances of a given size.

As expected from the numbers of assumptions, the strategies `ff` and `ffc` perform similarly well. Their running times are hardly distinguishable in the chart. This suggests namely that the overhead of using the more complex heuristics of `ffc` over `ff` is negligible.

You can also observe a steep jump in running time of the strategy `leftmost` at instance size 64. Since `ff` and `ffc` adapt to larger instance sizes more smoothly and perform well even on smaller instances, they appear to be better choices in general for this solver.

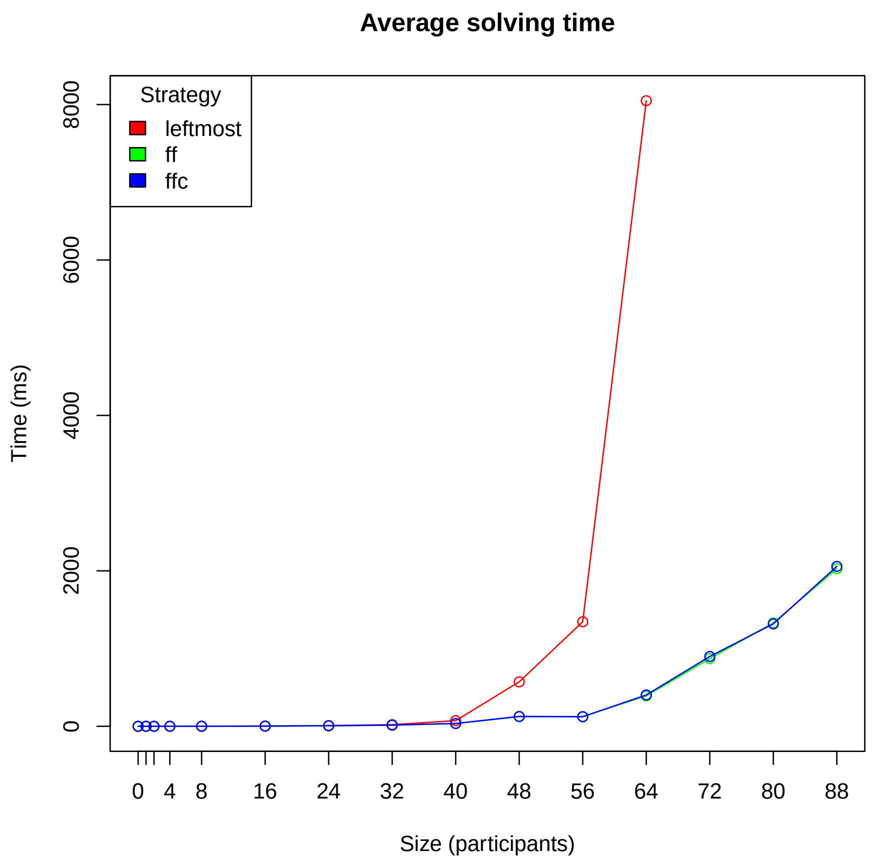

Figure 5: Average solving time

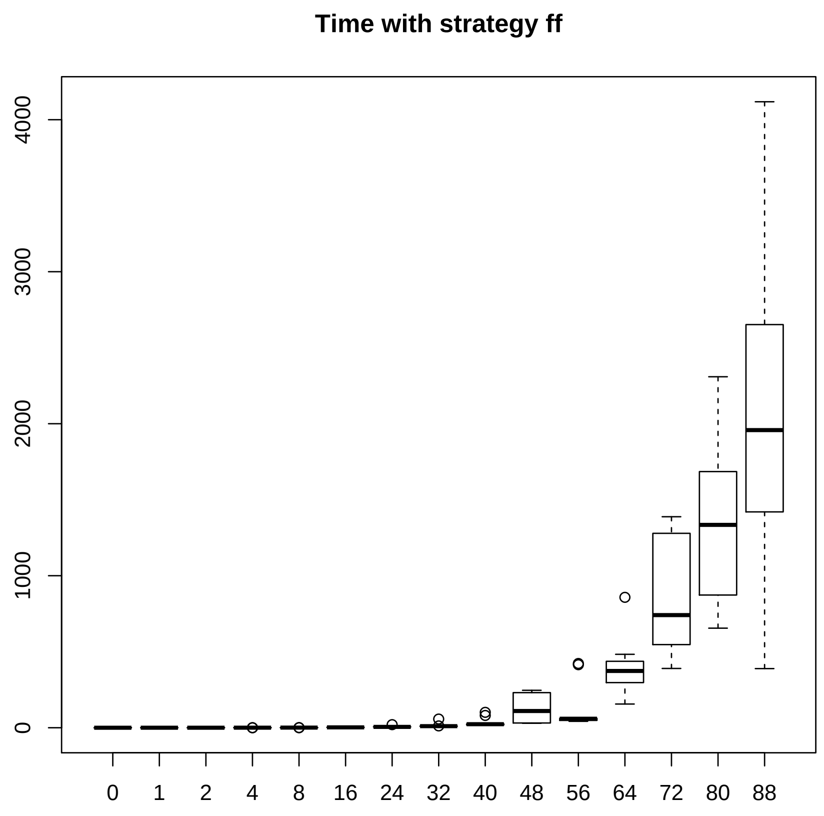

Figure 6: Dispersion of running time with search strategy `ff`

The running times differ a lot among instances of a given size. As an example a box plot of running times with strategy `ff` is shown in figure 6.

Note that SRP was shown to be soluble in quadratic time.[1]

4 Conclusions

It appears that the delivered implementation of SRP solver can be used for fast solving of problem instances of sizes up to approximately 88. The measured average time for solving an instance of size 88 with search strategy `ff` is approximately 2 seconds.

[1]Source: Wikipedia - Stable roommates problem

The search strategies `ff` and `ffc` perform significantly better than the strategy `leftmost` on instances of size at least 48.

Glossary

`clpfd` Constraint Logic Programming over Finite Domains. 1–3

`ff` first-fail. 4, 7, 8, 10

SRP stable roommates problem. 1–4, 10