Constraints on Set Variables for Constraint-based Local Search

(MSc thesis at Uppsala University)

Rodrigo Gumucio

SAIS yearly workshop 2012

May 14, Örebro, Sweden
(Video link from La Paz, Bolivia)

Motivating problem: The Social Golfer problem

- Imagine that in a golf club, $g \cdot s$ players meet once a week in order to play golf in g groups of size s.

- The challenge is to schedule a tournament over w weeks such that any two players meet in at most one week.
- An instance of this problem is denoted by

$$\texttt{golf-}g{\cdot}s\texttt{-}w$$

where

```
g is the number of groups
s is the size of each group
w is the number of weeks
g·s is the total number of players
```

- The figure above shows a solution to `golf-3·3-4`.

golf-g·s-w is a constraint problem

- The social golfer problem can be modelled with constraints:
  - Each of the $g \cdot s$ players plays in exactly one group each week.
  - All g groups of a week are of the same size s.
  - Any two players meet in at most one week.
- It can be modelled with either *integer* or *set* variables, and hence with either *integer* or *set* constraints, respectively.

A *set model* is given by:

- A 2d matrix of set variables: $Players_{gw} \equiv$ the set of players meeting in group g of week w.
- A new AtMost1(*Players*) set constraint to ensure any two players meet at most once.

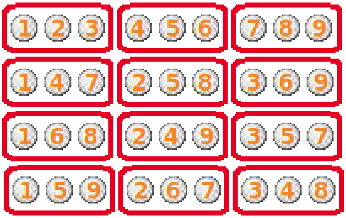
An *integer model* is given by

- A 3d matrix of int variables: $Player_{gsw} \equiv$ the player of **slot** s in group g of week w.
- A SocialTournament(*Player*) integer constraint to ensure any two players meet at most once.

`golf-g·s-w` integer and set models

Consider again the `golf-3·3-4` instance:

Model with set variables:



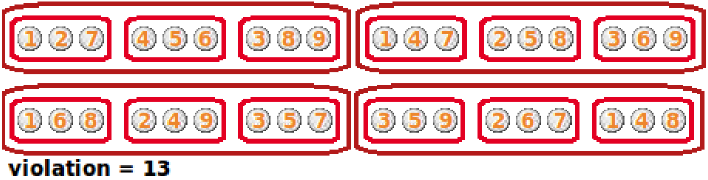
- *Players*$_{gw}$ has $3 \cdot 4$ set vars.
- A single set constraint: ATMOST1(*Players*).
- No need to introduce a concept outside the problem formulation.

Model with int variables:

- *Player*$_{gsw}$ has $3 \cdot 3 \cdot 4$ int vars.
- A single integer constraint: SOCIALTOURNAMENT(*Player*).
- Needs to introduce the concept of player **slot** within a group.

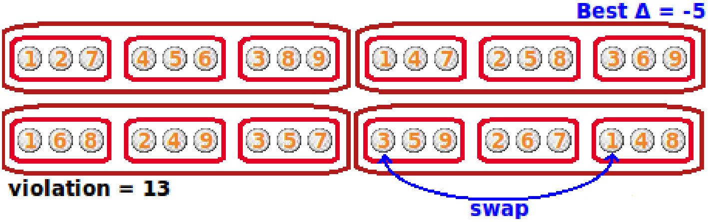
Constraint-based local search

- Constraint-based local search is a useful technique to find solutions to constraint problems using stochastic local search.
- It trades the completeness and quality of a systematic search technique (like constraint programming) for speed and scalability.



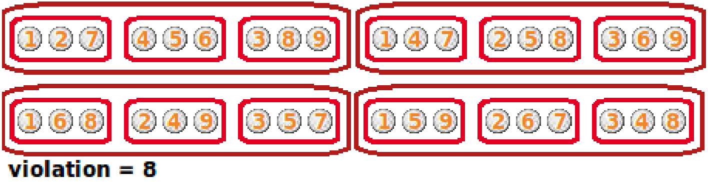
Constraint-based local search

- Constraint-based local search is a useful technique to find solutions to constraint problems using stochastic local search.
- It trades the completeness and quality of a systematic search technique (like constraint programming) for speed and scalability.



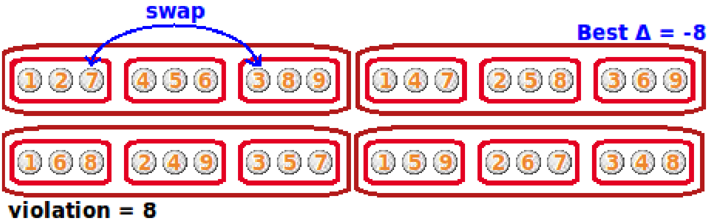
Constraint-based local search

- Constraint-based local search is a useful technique to find solutions to constraint problems using stochastic local search.
- It trades the completeness and quality of a systematic search technique (like constraint programming) for speed and scalability.



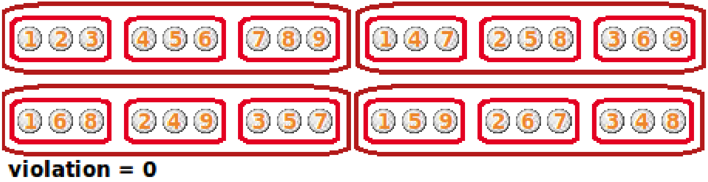
Constraint-based local search

- Constraint-based local search is a useful technique to find solutions to constraint problems using stochastic local search.
- It trades the completeness and quality of a systematic search technique (like constraint programming) for speed and scalability.

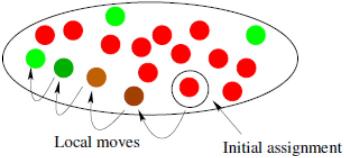


Constraint-based local search

- Constraint-based local search is a useful technique to find solutions to constraint problems using stochastic local search.
- It trades the completeness and quality of a systematic search technique (like constraint programming) for speed and scalability.

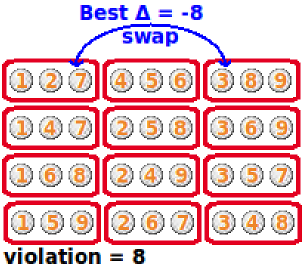


- To find solutions using local search:
  1. (Randomly) initialise all the variables.
  2. Re-assign a few variables: local move.
  3. If the new assignment is not good enough, then go to step 2.



Constraints in constraint-based local search

- Constraints are used mainly to:
  - Guide the local search to promising regions in the search space.
  - Determine when a given assignment is regarded as a solution.
- Constraints are implemented by a set of functions:
  - Violation functions help to select a promising variable (of a promising constraint) to re-assign in a move.
  - Differentiation functions help to make a move in a good direction for a constraint or variable.



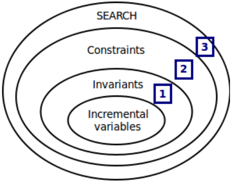
- The violation functions for AtMost1 basically count the number of times two players meet after the first allowed time.
- AtMost1 needs a differentiation function for swap moves.
- These functions must be very fast.

Main contributions

- Solid evidence that, using constraint-based local search, solving problems modelled with sets has the following advantages:
  - It can reduce the solution time.
  - It can even be a necessity in terms of memory.
- The design and implementation of an extension of a constraint-based local search solver (namely `Comet`) by:
  - Adding the notion of set constraint.
  - Providing the notion of set constraint system (i.e., a constraint combinator for constraints on set variables).

Comet's local search architecture

- Comet is a language and tool for modelling and solving constraint problems, using systematic or local search.
- It represents the state-of-the art in constraint-based local search.



Comet's architecture

- Unfortunately, it does not support set constraints for local search.
- Fortunately, it does support user-defined invariants on set incremental variables.

- An extension is possible: The architecture is open, and set constraints can be built on top of set invariants!
- Two new features are needed at the constraint layer:
  - Support for user-defined set constraints.
  - Support for a mechanism to combine such constraints, that is at least one constraint combinator.

Extension of Comet's local search

The extension consists of:

- A SetConstraint<LS> interface together with an abstract class that provides a mechanism to define set constraints.
- A set constraint system that provides a mechanism to combine set constraints.

```
interface SetConstraint<LS> {
  ...
  var{set{int}}[] getSetVariables();

  var{int} violations();
  var{int} violations( var{set{int}} s);

  int getSwapDelta( var{set{int}} s, int u,
                    int v, var{set{int}} t);
  ...
}
```

- To define a set constraint, extend and specialise the abstract class (named UserSetConstraint<LS>).
- The provided constraint combinator (i.e., the set constraint system) could be done only through a tricky implementation.
- The full source code is in my MSc thesis.

The Social Golfer problem: the AtMost1 constraint

- My AtMost1 constraint is the set version of the SocialTournament integer constraint of [Dynamic Decision Technologies, 2010].
- The essence of the integer version: count the number of times players a and b meet, denote it by $\#(a, b)$, and maintain it incrementally.
- The same can be done with set variables: keep the set of groups where a and b meet, denote it by $m(a, b)$, and maintain it incrementally.
- Note: $|m(a, b)| = \#(a, b)$.
- The violation and differentiation functions are based on these values.
- The constraint is satisfied whenever $|m(a, b)| \leq 1$ for all a and b.

The Social Golfer problem: search

The tabu search algorithm of [Dynamic Decision Technologies, 2010] (based on [Dotú and Van Hentenryck, 2007]) is adapted for the set approach:

```
...
while (violations > 0 && (System.getCPUTime() - t0) < timeout )
  selectMin(w in Weeks,
            g1 in Groups,s1 in Slots:  conflict[w,g1,s1] > 0,
            g2 in Groups:  g2 != g1, s2 in Slots,                  select golfers
            delta = tourn.getSwapDelta(golfer[w,g1,s1], golfer[w,g2,s2]) :
            tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] < it || violations + delta < best)
    (delta) {
    golfer[w,g1,s1] :=: golfer[w,g2,s2]; ...
```

```
...
while(violations > 0 && (System.getCPUTime() - t0) < timeout)
  selectMin(w in Weeks,
            g1 in violatedGroups[w], g2 in Groups:  g2 != g1,      select groups
            s1 in golfersInConflict[w,g1],
            s2 in group[w,g2],                                     select golfers
            delta = tourn.getSwapDelta(group[w,g1], s1, s2, group[w,g2]):
    ((tabu[w,s1,s2] < it) || violations + delta < best))
    (delta){
    group[w,g1].delete(s1); group[w,g2].insert(s1);
    group[w,g2].delete(s2); group[w,g1].insert(s2); ...
```

The Social Golfer problem: results over 25 runs

Run time (milliseconds)

instance	integer model					set model				
g-s-w	n	average	min.	max.	std.dev.	n	average	min.	max.	std.dev.
6-3-8	25	166367.08	7447	549890	152743.43	24	378689.79	64166	1165813	324722.24
6-4-6	25	72048.84	2958	229617	61596.19	25	49789.28	16503	190741	50348.29
6-5-6	0	> timeout	-	-	-	3	360961.33	2291	714505	356134.68
7-3-9	25	7847.88	780	24463	5118.96	25	3134.92	2685	5575	802.60
7-4-7	24	352799.88	1907	831812	205866.61	25	196922.04	3726	423436	127522.16
7-6-4	25	85.80	65	126	26.44	25	71.12	69	80	2.63
8-3-10	25	1100.08	444	2834	502.67	25	348.64	286	490	38.19
8-4-8	25	694801.16	17162	1306108	512511.30	25	73475.88	1680	319205	76351.06
8-5-6	25	357.84	166	1237	286.40	25	167.80	73	441	104.77
8-6-5	25	2622.16	678	8278	1870.69	25	1491.72	619	2788	469.61
8-7-4	25	443.28	197	1016	226.10	25	451.96	283	1221	225.39
8-8-5	0	> timeout	-	-	-	8	30385.25	6668	188221	63817.12
9-3-12	5	815324.00	28589	1577964	574503.40	23	791874.87	6454	1659081	544002.65
9-4-9	25	347040.68	111125	1136076	268002.54	25	38815.56	6211	66977	15127.62
9-5-7	25	8130.24	686	25774	7055.40	25	1129.12	1099	1183	19.26
9-6-6	25	245999.64	20040	875592	238568.03	25	16756.64	4404	43071	10762.36
9-7-5	25	23233.52	20090	38746	4959.37	25	20025.24	1550	91299	28131.22
9-8-4	25	2325.52	1857	3563	584.27	25	3205.56	432	7740	2588.93
9-9-5	3	496988.00	44865	748887	392401.56	0	> timeout	-	-	-
10-3-13	25	35562.04	4894	188130	42931.09	25	7534.56	1312	36976	10748.78
10-4-10	25	960130.00	42674	1591866	535777.30	25	33905.04	2254	121984	34411.04
10-5-8	25	125212.08	9853	626327	158165.31	25	5563.68	1815	24087	5322.64
10-6-7	0	> timeout	-	-	-	6	710276.33	198163	1705522	624786.04
10-7-5	25	435.32	429	446	4.59	25	405.76	394	438	12.04
10-8-5	2	950746.50	616029	1285464	473362.03	12	729656.17	41395	1633916	489606.14
10-9-4	25	26255.20	1860	143770	30612.23	25	17792.92	1431	49714	13697.91

The Social Golfer problem: solved instances

	$s=3$		$s=4$		$s=5$		$s=6$		$s=7$		$s=8$		$s=9$		$s=10$	
g	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$	w	$\delta(w)$
6	8	0	6	0	6	0	3	0	-	-	-	-	-	-	-	-
7	9	0	7	0	6	-1	4	-1	8	0	-	-	-	-	-	-
8	10	0	8	0	6	0	5	-3	4	0	5	-4	-	-	-	-
9	12	0	9	0	7	0	6	-3	5	0	4	0	5	-5	-	-
10	13	0	10	0	8	0	7	$+1$	5	0	5	$+1$	4	0	3	0

The $\delta(w) \geq 0$ values are relative [Dotú and Van Hentenryck, 2007], which uses the same meta-heuristic; the negative ones are relative the state of the art.

Two instances not solved by [Dotú and Van Hentenryck, 2007] were solved (as by [Cotta et al., 2006] and [Harvey and Winterer, 2005]):

- `golf-10-6-7`
- `golf-10-8-5`

Schur's problem

- A set T of integers is sum-free if $a, b \in T \rightarrow a + b \notin T$.
  Example: $\{1, 3, 5\}$. Counterexamples: $\{1, 3, 4\}$ and $\{1, 2\}$.
- Schur's problem, denoted `schur-k-n`, is about finding a partition of the set $\{1, \ldots, \mathtt{n}\}$ into `k` sum-free sets.
  Let $S(k)$ denote the largest such n.
- $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, $S(4) = 44$, but $S(5)$ is unknown.
  Example: $S(2) = 4$ as $\{1, 2, 3, 4\} = \{1, 4\} \cup \{2, 3\}$.
- Modelling this problem with integer variables will not scale:
  A set model requires k SUM-FREE constraints, while
  an integer model requires $O\left(k \cdot n^2\right)$ SUM-FREE constraints.
- To solve this problem with constraint-based local search:
  - A SUM-FREE set constraint is needed.
  - A tabu-search meta-heuristic is used for simplicity.

Schur's problem: results

GC memory usage (KB)

instance	integer model					set model				
o	n	average	min.	max.	std.dev.	n	average	min.	max.	std.dev.
3-13	25	31015.56	21224	37010	5209.47	25	18371	16955	18550	487.42
4-37	25	244420.28	161948	304502	46818.22	23	18295	17020	19334	559.48
4-38	25	258275.08	178221	319811	45790.06	18	18039	17078	19690	846.86
4-39	21	272206.86	200513	336613	49212.87	17	18777	17622	19685	493.68
4-40	7	277904.29	200637	331583	50458.18	5	17745	17275	18714	582.44
4-41	7	286657.71	210572	346756	56444.48	2	19155	18902	19408	357.80
4-42	6	327303.50	255207	364786	45433.86	1	17817	17817	17817	-
4-43	2	300813.00	239060	362566	87331.93	1	17401	17401	17401	-
4-44	1	396890.00	396890	396890	-	1	18214	18214	18214	-

VM memory usage (KB)

instance	integer model					set model				
o	n	average	min.	max.	std.dev.	n	average	min.	max.	std.dev.
3-13	25	62914.56	32768	65536	9073.05	25	32768	32768	32768	0.00
4-37	25	492830.72	262144	524288	86943.33	23	32768	32768	32768	0.00
4-38	25	513802.24	262144	524288	52428.80	18	32768	32768	32768	0.00
4-39	21	524288.00	524288	524288	0.00	17	32768	32768	32768	0.00
4-40	7	524288.00	524288	524288	0.00	5	32768	32768	32768	0.00
4-41	7	524288.00	524288	524288	0.00	2	32768	32768	32768	0.00
4-42	6	524288.00	524288	524288	0.00	1	32768	32768	32768	-
4-43	2	524288.00	524288	524288	0.00	1	32768	32768	32768	-
4-44	1	524288.00	524288	524288	-	1	32768	32768	32768	-

Schur's problem: solved instances

- Problem instances require less memory using the set model.
- Both the integer and set models find the best solutions to the closed instances, that is Schur numbers up to $S(4) = 44$.
- Unfortunately, the advantage in memory consumption is not enough to find $S(5)$, which thus remains open.

Steiner triple systems

- A similar experiment was done for Steiner triple systems.
- As expected, the set approach required much less memory.
- Instances much larger than with an integer model are solved.
- Check my MSc thesis for details.

Conclusion and contributions

I have demonstrated that set variables for constraint-based local search are not only a convenience for faster & higher-level modelling. Set variables, and hence set constraints, can be necessary because solutions to problem instances with integer variables:

- may not be found otherwise,
- would not fit into memory, or
- take much more time to be solved.

I have also contributed an extension of the constraint-based local search back-end of `Comet` to support set constraints.

My MSc thesis is at
`http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-159180`.

Bibliography

Cotta, C., Dotú, I., Fernández, A., and Van Hentenryck, P. (2006).
Scheduling social golfers with memetic evolutionary programming.
In *Hybrid Metaheuristics*, volume 4030 of *LNCS*, pages 150–161.

Dotú, I. and Van Hentenryck, P. (2007).
Scheduling social tournaments locally.
AI Communications, 20(3):151–162.

Dynamic Decision Technologies (2010).
Comet tutorial (version 2.1.1), section 21.2.

Harvey, W. and Winterer, T. (2005).
Solving the MOLR and social golfers problems.
In *Proceedings of CP'05*, volume 3709 of *LNCS*, pages 286–300.