

Comparing classical methods for solving binary constraint satisfaction problems with state of the art evolutionary computation

Jano van Hemert

`jvhemert@liacs.nl`

`http://www.liacs.nl/~jvhemert`

LIACS

Niels Bohrweg 1

2333 CA Leiden

The Netherlands



Presentation outline

- ✓ Motivation
- ✓ Description and explanation of the problem — BINCSP
- ✓ Two pairs of algorithms
 - ① “Classic” algorithms — BT & FC-CBJ
 - ② Evolutionary algorithms — MID & SAW
- ✓ Experiments & results
 - ① Comparison experiments — varying density and tightness
 - ② Scale-up experiments — varying the number of variables
- ✓ Concluding remarks

Motivation

- ☞ Many comparison studies between EAs and non-evolutionary methods on a specific problem
- ☞ None so far we know of that compares EAs and other methods on the general model of binary constraint satisfaction

What is a constraint satisfaction problem?

Definition 1 (Constraint Satisfaction Problem) *A Constraint Satisfaction Problem is a tuple $\langle Z, D, C \rangle$ where*

- *Z is a set of variables,*
- *D is a function that maps a finite set of objects of arbitrary type to Z*
- *and C is a set of constraints that restrict certain simultaneous object assignments.*

👉 Abbreviation: Constraint Satisfaction Problem \rightarrow CSP

Objective in CSPs

☞ Solution to a CSP: assign a value from D to each of the variables in Z such that none of the constraints in C is violated

Possible objectives

- ✓ *Finding a solution*
- ✓ Finding all solutions
- ✓ Proving that there is no solution for a given problem
- ✓ Finding the partial solution with the most instantiated variables for an unsolvable problem

Examples

- ✓ Graph colouring: given a graph find a k -colouring of the nodes such that nodes connected are coloured with a different colour
- ✓ n -Queens: given a $n \times n$ chess board and n queens, place the queens on the board such that no queen attacks another queen
- ✓ SAT: given a boolean formula, find an assignment of variables such that the formula evaluates to true

👉 These are all decision problems

Binary Constraint Satisfaction Problems

Definition 2 (Binary Constraint Satisfaction Problem) *A Binary Constraint Satisfaction Problem is a CSP where all constraints are associated with at most two variables.*

- ☞ This does not restrict the generality of our CSP model as every CSP can be transformed into a binary CSP [Tsang, 1993]
- ☞ The transformation can influence the efficiency of the solving method [Bacchus & van Beek, 1998]

A model for generating BINCSPPs

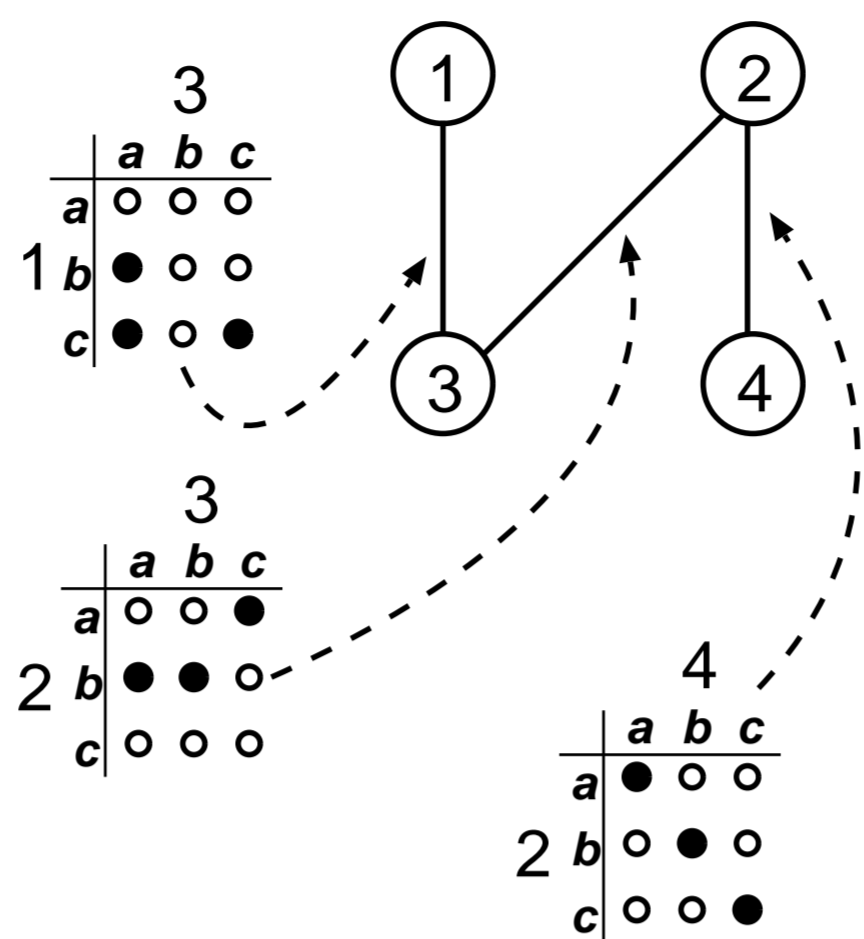
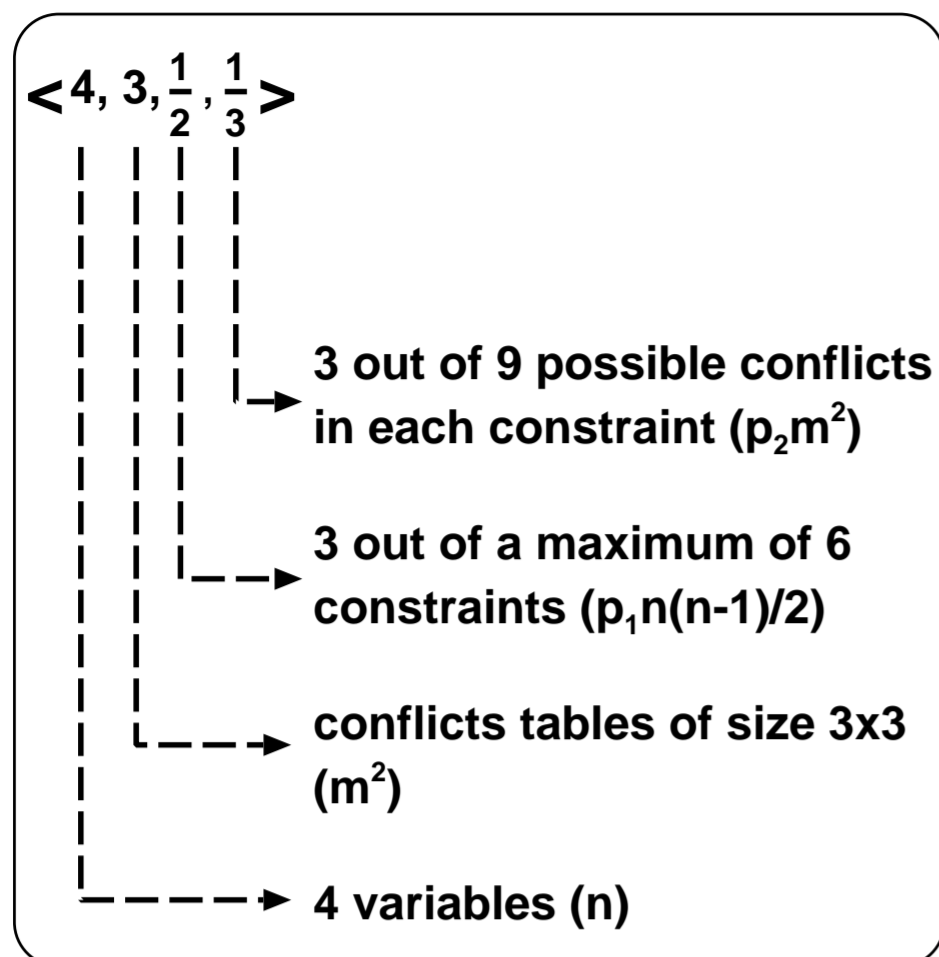
Parameters

- ① Number of variables (n)
- ② Domain size of each variable (m)
- ③ Density of the constraint network (p_1), between 0 and 1
- ④ Average tightness of a constraint (p_2), between 0 and 1

Recipe

- ☞ We use Model B type of generating instances: given the four parameters $\langle n, m, p_1, p_2 \rangle$ calculate the number of constraints and conflicts that ought to be present and distribute these randomly to form a binary constraint satisfaction problem [Palmer, 1985]

Example: a very simple "random" instance



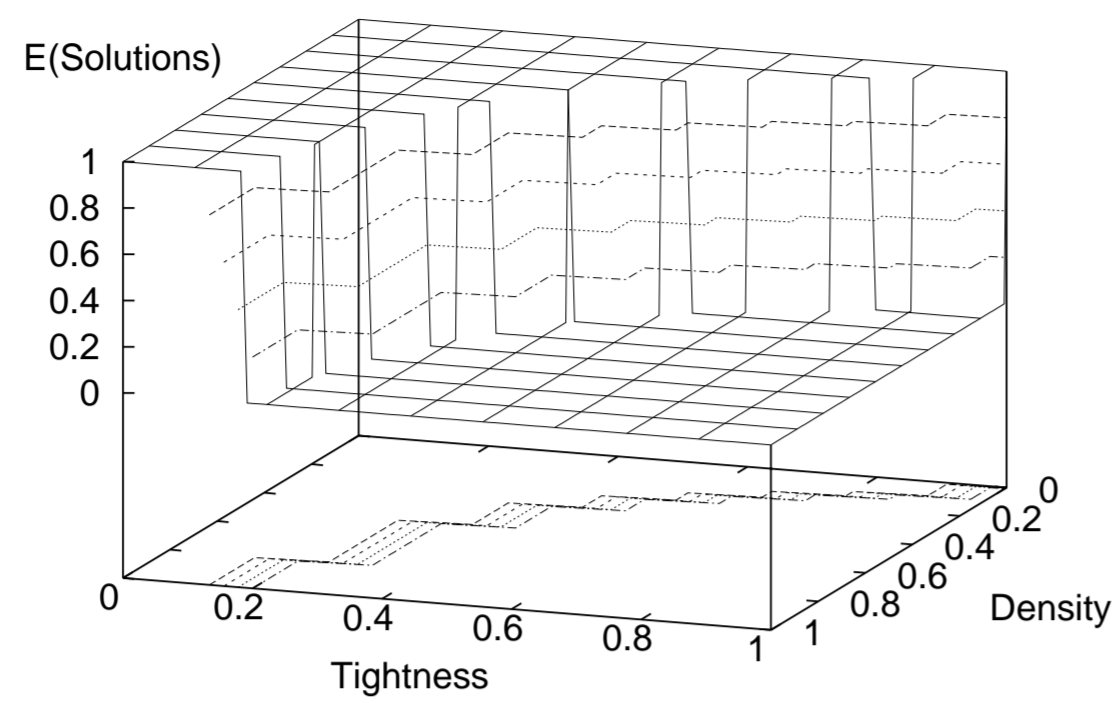
Difficult problem instances

☞ Conjecture by Smith: difficult problem instances have only one solution [Smith, 1994]

☞ Using this as an assumption we can estimate the values of the four parameters as

$$E(\text{Solutions}) = m^n (1 - p_2)^{\frac{n(n-1)p_1}{2}} = 1$$

The landscape of solvability



☞ The expected number of solutions with $E(\text{Solutions}) \leq 1$ for fixed $n = 15$ and $m = 15$ plotted against density and tightness

Problem generator RandomCsp

- ✓ Randomly generates BINCSPs
- ✓ Provides functionalities for solving and analysing problem instances
- ✓ Creates using Models A,B,C,D and E as used in the literature, especially in the field of Constraint Programming
- ✓ Can be used as stand-alone programs or as a C++ library
- ✓ Freely available under the GNU General Public License at <http://www.liacs.nl/~jvhemert/randomcsp>

Chronological backtracking (by Golomb & Baumert, 1965)

```
✓ bool Backtrack(solution[], current)
  if (current > number_of_variables) then
    return true;
  else
    foreach  $d \in D_{current}$  do
      solution[current] = d;
      if (consistent(solution, current)) then
        if (Backtrack(solution, current + 1)) then
          return true;
    return false;

✓ bool Consistent(solution[], current)
  for  $i = 1 \dots current - 1$  do
    constraint_checks++;
    if (conflict(current, i, solution[current], solution[i])) then
      return false;
  return true;
```

Forward checking with conflict-directed backjumping (by Prosser, 1993)

Forward checking

- ☞ Idea: avoid visiting inconsistent nodes by looking forward
- ✓ Instantiate a *current* variable
- ✓ Remove values incompatible with current instantiation from domains of uninstantiated variables
- ✓ Continue until all variables are instantiated (= solution) or until a domain is annihilated (= backjump & undo forward looking)

Backjumping

- ☞ Idea: improve speed by directly jumping to variables that cause dead-ends in the search
- ✓ While trying to instantiate a current variable remember which variables caused a conflict
- ✓ When no value can be found for our current variable jump to the variable farthest away from our remembered set

Conflict-directed backjumping

- ☞ Idea: further enhance backjumping by doing more bookkeeping
- ✓ Keep a set of conflicting variables at *each* variable
- ✓ When we need to do a backjump take the conflict set from the current variable to the variable we jump to

Microgenetic Iterative Method (by Dozier, 1994)

Principle

- ✓ One genetic operator that mutates one variable using a heuristic
- ✓ Small population size (< 10)
- ✓ Breakout Management System to escape local optima

Breakout Management System

- ✓ Used when no improvement has been made
- ✓ Updates the list of breakouts using constraint violations of the best individual
- ✓ A breakout consists of a pair of conflicting values and a weight
- ✓ The breakouts are used in the fitness function

Stepwise Adaptation of Weights (by Eiben et al., 1998)

Principle

- ✓ Fitness is weighted sum of violated constraints
- ✓ EA runs for T_p fitness evaluations and is then interrupted
- ✓ During interruption the weights are raised using constraint violations from the best individual

Features

- ✓ Size of population is one with preservative selection
- ✓ One genetic operator that mutates by swapping variables
- ✓ Order based representation, using a greedy algorithm as decoder

Measurements

- ✓ The ratio of successful runs (SR)
- ✓ The number of constraint checks performed on average (ACS)
- ✓ Every results is averaged over 25 instances
- ✓ A classic algorithm always succeeds thus we perform 1 run per instance
- ✓ An evolutionary algorithm is terminated after 100,000 fitness evaluations; we perform 10 independent runs per instance

Verification

- ✓ Our ACS results are not normally distributed according to the KS-Liliefors test
- ✓ We use the Wilcoxon Rank test to verify significance of our results

Comparison experiment

- ✓ number of variables $n = 15$
- ✓ domain size $m = 15$
- ✓ density $p_1 = \{0.1, 0.3, 0.5, 0.7, 0.9\}$
- ✓ tightness $p_2 = \{0.1, 0.3, 0.5, 0.7, 0.9\}$

density	algorithm	tightness									
		0.1		0.3		0.5		0.7		0.9	
0.1	BT	1.00	110	1.00	114	1.00	130	1.00	584	0.96	534
	FC-CBJ	1.00	1529	1.00	1431	1.00	1346	1.00	1254	0.96	1044
	MID	1.00	10	1.00	40	1.00	210	1.00	870	0.96	29230
	SAW	1.00	10	1.00	10	1.00	20	1.00	90	0.64	11590
0.3	BT	1.00	117	1.00	167	1.00	9169	0.68	150270	0.00	42702
	FC-CBJ	1.00	1395	1.00	1085	1.00	869	0.68	16750	0.00	20008
	MID	1.00	93	1.00	1550	1.00	10013	0.52	1004772	0.00	3100000
	SAW	1.00	31	1.00	62	1.00	1116	0.23	21281	0.00	3100000
0.5	BT	1.00	131	1.00	4351	1.00	103228	0.00	22218	0.00	4392
	FC-CBJ	1.00	1285	1.00	854	1.00	15444	0.00	6813	0.00	5208
	MID	1.00	520	1.00	9204	0.90	1393184	0.00	5200000	0.00	5200000
	SAW	1.00	52	1.00	416	0.74	557544	0.00	5200000	0.00	5200000
0.7	BT	1.00	152	1.00	12974	0.00	194909	0.00	6250	0.00	4300
	FC-CBJ	1.00	1173	1.00	1044	0.00	41851	0.00	4619	0.00	3982
	MID	1.00	1460	1.00	44092	0.00	7300000	0.00	7300000	0.00	7300000
	SAW	1.00	73	1.00	5329	0.00	7300000	0.00	7300000	0.00	7300000
0.9	BT	1.00	209	1.00	121187	0.00	76826	0.00	3265	0.00	2349
	FC-CBJ	1.00	1097	1.00	9454	0.00	24563	0.00	3561	0.00	3412
	MID	1.00	3102	1.00	764784	0.00	9400000	0.00	9400000	0.00	9400000
	SAW	1.00	94	1.00	361712	0.00	9400000	0.00	9400000	0.00	9400000

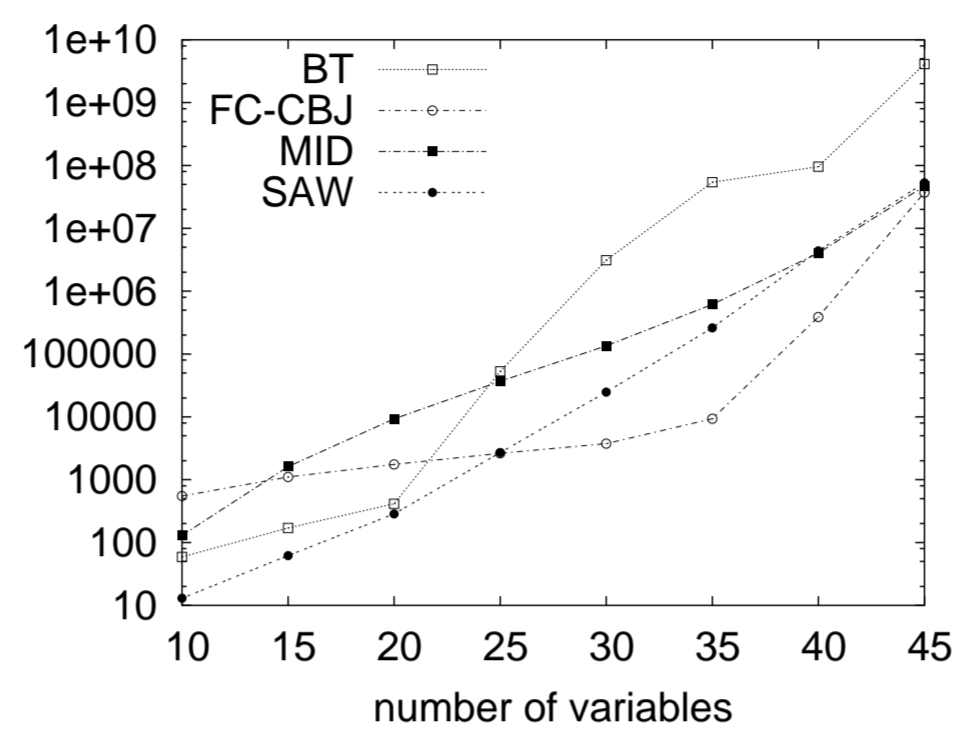
Scale-up experiment

- ✓ number of variables $n = \{10, 15, 20, 25, 30, 35, 40, 45\}$
- ✓ domain size $m = 15$
- ✓ density $p_1 = 0.3$
- ✓ tightness $p_2 = 0.3$

👉 All the instances are solvable

Scale-up results

n	BT	FC-CBJ	MID		SAW	
	ACS	ACS	SR	ACS	SR	ACS
10	5.90e1	5.49e2	1.00	1.30e2	1.00	1.30e1
15	1.71e2	1.10e3	1.00	1.61e3	1.00	6.20e1
20	4.14e2	1.75e3	1.00	9.29e3	1.00	2.85e2
25	5.31e4	2.63e3	1.00	3.69e4	1.00	2.70e3
30	3.10e6	3.75e3	1.00	1.35e5	1.00	2.47e4
35	5.43e7	9.32e3	1.00	6.16e5	1.00	2.61e5
40	9.57e7	3.88e5	1.00	4.04e6	1.00	4.37e6
45	4.10e9	3.70e7	0.44	4.70e7	0.24	5.30e7



Conclusions

- ✓ In the mushy region the performance of EAs is quite low
- ✓ A major problem for EAs is not being able to cope with unsolvable instances
- ✓ Evolutionary algorithms still have a lot of catching up to do if we wish them to compete with algorithms from the field of Constraint Programming

Future directives

- ✓ Stop using Model B for generating binary constraint satisfaction problems and start using Model E (in line with events in Constraint Programming)
- ✓ Explain the lack of performance around the mushy region
- ✓ Try to increase the number of variables above $n = 45$
- ✓ Do a similar comparison using dynamic CSPs
- ✓ Can evolutionary algorithms help guide classic algorithms in such a way we improve on the speed? (some preliminary results available)