Constraint Satisfaction Problems
and
Evolutionary Computation

Jano van Hemert
jvhemert@liacs.nl
http://www.liacs.nl/~jvhemert

LIACS
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
Contents

1 Theoretical introduction
   ✔ Constraint satisfaction
   ✔ Binary constraint satisfaction
   ✔ Randomly generated instances
   ✔ Using evolutionary computation

2 Practical introduction
   ✔ Things you get
   ✔ How to get it all up and running
   ✔ What’s in it for you
What is a constraint satisfaction problem?

**Definition 1** A Constraint Satisfaction Problem (CSP) is a tuple $(Z, D, C)$ 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.

Thus each $x_i \in Z$ has a corresponding discrete domain $D_i$ from which they can be instantiated, denoted as $(x_i, d_i)$, where $d_i \in D_i$. Every element $c \in C$ is a constraint over a subset of variables of $X$, it contains tuples of objects that are not allowed to be assigned simultaneously.

Abbreviation: Constraint Satisfaction Problem $\rightarrow$ CSP
So what is the problem?

Assign to each $x_i \in Z$ an object from $D_i$ such that no $c \in C$ is violated.

Extended objectives:

- ✔ Finding all possible instantiations of variables that do not violate a constraint
- ✔ Proving that there is no solution (object assignment) for a given problem
- ✔ Finding a partial solution with the most instantiated variables for an unsolvable problem instance
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

✎ In general all these problems belong to the class of NP-complete problems
Example: graph-$k$ colouring with $k = 3$

$$Z = \{ x_1, x_2, x_3, x_4, x_5 \}$$

$$D = \{ \text{red, blue, green} \}$$

$$C = \{ (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_2, x_4), (x_4, x_5) \},$$

where $<x_i \text{ colour}> \neq <x_j \text{ colour}>$

Solution: $\{ <x_1, \text{ red}>, <x_2, \text{ blue}>, <x_3, \text{ red}>, <x_4, \text{ green}>, <x_5, \text{ red} \}$
Definition 2 (Binary Constraint Satisfaction Problem) A Binary Constraint Satisfaction Problem (BINCSP) is a CSP where all constraints are associated with at most two variables. More precisely: Given the CSP \( (Z, D, C) \) the following must hold: \( \forall c \in C : |\hat{X}| \leq 2 \).

\( \triangledown \) This is not a restriction as every CSP can be transformed into a binary CSP (Tsang, 91)

\( \triangledown \) Multiple transformations may exist, where each transformation has its own impact on the efficiency of solving the problem (not in the scope of this summer school)

\( \triangledown \) Abbreviation: Binary Constraint Satisfaction Problem \( \rightarrow \) BINCSP
Example: transforming 4-Queens into a BINCSP
Why the need for BIN CSPs?

Idea: Generate random problem instances based on the BIN CSP model to do experiments.

Technique: by introducing parameters we will try to control the difficulty of a randomly generated problem instance.

Parameters:

1. Number of variables ($n$)
2. Domain size of each variable ($|D|$ or $m$)
3. Density of the constraint network ($p_1$ or $d$), between 0 and 1
4. Average tightness of a constraint ($p_2$ or $t$), between 0 and 1
Example: a very simple instance

$<4, 3, \frac{1}{2}, \frac{1}{3}> = <n, m, p_1, p_2>$

3 out of 9 possible conflicts in each constraint ($p_2 m^2$)

3 out of a maximum of 6 constraints ($p_1 \frac{1}{2} n(n-1)$)

conflicts tables of size 3x3 ($m^2$)

4 variables ($n$)
Difficult problem instances

Assumption of B. Smith: Difficult problem instances have only one solution

Using the assumption and a predictor for the expected number of solutions, we can estimate the values of the four parameters to identify difficult instances:

\[ E(\#\text{solutions}) = mn(1 - p_2) \frac{n^{(n-1)p_1}}{2} = 1 \]
The landscape of solvability

The expected number of solutions for fixed $n = 10, m = 10$
The other way around

☞ We can devise methods that generate instances in such a way that we know the parameters beforehand.

☞ Six methods exist in the literature: Models A–D, Model E, Model F.

☞ Model E works as follows, pick randomly two variables, then from each variable’s domain pick randomly an object. If no conflict exists between the two, create one. Model E repeats this process $p_e \binom{n}{2}|D|^2$ times, where $p_e$ can be used to set the conflict density, which has a direct influence on the difficulty.
Performance and difficulty

✔ We measure the percentage of instances where a solution is found ⇒ success rate

✔ We measure the average number of conflict checks performed

✔ We generate a test suite of instances using Model E by varying $p_e$ from 0.10 to 0.38 in steps of 0.02 where for each step 25 unique instances are created

✔ When testing evolutionary algorithms, we let an algorithm do 10 runs on one instance, each time with a different random seed
Some results
Solving CSPs with Evolutionary Algorithms

✔ **Representation** — *simple*

✔ **Initialisation** — *random object assignment*

✔ **Genetic operators**
  - Mutation — $\frac{1}{t}$
  - Crossover — *uniform*

✔ **Fitness** — *counting violated constraints*

✔ **Selection**
  - Parent selection — *linear ranked bias (bias = 2)*
  - Survivor selection — *replace worst*

✔ **Stop condition** — *solution found or 100,000 evaluations*
Representing the problem (or rather the solution)

Simple representation

Advantages are the use of simple genetic operators and easy evaluation of an individual
Representing the problem (or rather the solution)

More difficult, using a decoder

![Permutation diagram]

\[
\begin{align*}
&\text{permutation of the variables } x_1 \ldots x_n \\
&\begin{array}{c}
7 \quad 4 \\ 9 \\ n \quad 3 \\
\vdots \quad \ldots \\
&1
\end{array}
\end{align*}
\]

\[
< x_1, 1 >, < x_2, 3 >, < x_3, 3 >, \ldots, < x_n, 2 >
\]

greedy decoder

Advantage is that it works much better, especially on easy to solve instances
Determining the quality of your solution

- Difficult because we are searching only for a no/yes question (solved/not solved)
- Common solution is to count the number of violated constraints, minimising this number to zero leads to a solution
- On the other hand this can easily get your algorithm stuck in a local minima, therefore you will need to guide its search somehow
- Ideas to do this exist and will be explained on request or similarity of proposal ;-
- Other difficulties for an evolutionary algorithm exists, such as symmetry and deception
The things you get, documentation

✔ The Online Guide to Constraint Programming by Roman Barták (HTML, 1998)

✔ Pages from the Handbook on Evolutionary Computation on Constraint Satisfaction by G. Eiben & Zs. Ruttkay (PS, 1996)

✔ Assorted papers to help you get ideas, and a list to even more papers (PS, 1991–2001)

✔ Full web site of RandomCsp, the library you may use, comes with complete manual and reference guide (HTML & PS, 2001)

✔ These slides (PS & PDF, 2001)
The things you get, for you to work with

✔ Set of problem instances that are currently used in empirical research
✔ RandomCsp library setup and ready to go
✔ Some results to compare with
✔ An example to show the basic usage of the library
✔ An experiments manager that takes care of doing all the experiments for you
It really is easy to use

```cpp
#include <static_csp.h>
#include <sstream>

int main (int argc, char * argv [])
{
    istream input ( argv[1] ); // Read in random seed
    int RandomSeed = 0; input >> RandomSeed; // Set random seed
    StaticCsp csp(argv[2]); // Read in CSP instance

    ValueT * solution = new ValueT [csp.GetNumberOfVariables() * sizeof(ValueT)]; // Create a random solution
    for (unsigned int i = 0; i < csp.GetNumberOfVariables(); i++)
    {
        solution[i] = (ValueT) (csp.GetDomainSize(i)*(rand()/(RAND_MAX+1.0))); // Output number of conflicts
        cost << csp.GetNumberOfConflicts(solution) << ";
    }
    cout << csp.GetNumberOfConflicts(solution) << ";"; // Output number of conflicts
    for (unsigned int i = 0; i < csp.GetNumberOfVariables(); i++)
    {
        cost << solution[i] << ";
    }
    cout << solution[i] << ";"; // Output solution
    for (unsigned int i = 0; i < csp.GetNumberOfVariables(); i++)
    {
        cost << solution[i] << " ";
    }
    cout << RandomSeed << " ", argv[2] << endl; // Output random seed and CSP filename
    return 0;
}
```
It really *is* easy to use

To run this example first do a `make` then start the experiment manager with the appropriate experiments:

```bash
./experiment.pl problemInstances/experiments
```

Output looks like this:

```
#conflicts, solution, random seed, problem file
```
Blatant advertisement

☞ Serious problem, serious work
☞ All the boring stuff has been done
☞ You just focus on creating a novel solving method
☞ Leaving you with plenty of fun time