

# NOPT042 Constraint programming: Tutorial 2 – Intro to CP

## What was in Lecture 2?

Local search algorithms

- generate and test
- hill climbing, min-conflicts + random walk, tabu search
- GSAT (local search for SAT), heuristics
- connectionist approach (GENET)
- simulated annealing
- intensification vs. diversification (exploit vs. explore)

## Constraint programming aka modeling

Discrete ('combinatorial', as opposed to 'continuous') optimization, constraint satisfaction

- a form of decision making, many everyday problems
- solve Sudoku
- schedule classes
- schedule trains
- coordinate multi-facility production
- logistics of product transportation
- ...

Assign values to variables subject to **constraints**, satisfy/optimize.

## We will learn to...

- solve complex problems "without even knowing how"
- state the problem in a high-level language, use a constraint solver to "automagically" solve it (magic explained in the lectures)
- techniques and tricks to build efficient constraint models
- best practices, testing and debugging

## Why constraint programming?

- the ‘holy grail’ of programming: tell the computer what you want, not how to do it
- an order of magnitude easier than programming algorithms
- huge engineering investment in constraint solvers, highly optimized,
- often faster than your own algorithm would be (especially in “mixed” NP-complete problems), heuristic approach
- easier for molecular biologists to learn to specify their problems in a formal language, than for programmers to learn molecular biology

## History and (folk) etymology

- prográphō ("I set forth as a public notice"), from pró ("towards") + gráphō ("I write")
- program of a political movement
- program of a concert, broadcast programming, tv program
- computer program (1940s) Independently:
- U.S. Army operational programs (1940s)
- "linear programming" (1946) Maximize  $\mathbf{c}^T \mathbf{x}$  (objective function) subject to  $A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0$  (constraints).
- integer programming (1964),
- logic programming (late 1960s)
- constraint logic programming (1987)
- constraint programming (early 1990s)
- Modeling (USA) vs. Modelling (Commonwealth)

## Why Picat?

From a high-level point of view, all constraint programming languages are quite similar. Picat is:

- modern, simple yet powerful
- easier syntax and better utils than SICStus Prolog
- more flexible than MiniZinc
- fast: won several CP competitions (see [Picat homepage](#))

## Recall the basics of constraint programming:

- **decision variables** vs. **parameters**
- **domains**
- **constraints**
- **solution**
- **constraint propagation**
- **search space**

- **decision** vs. **optimization**
- modeling strategies

See pages 27–34 from [Modeling and Solving AI Problems in Picat](#).

## Structure of a constraint program

Typically, the structure of a constraint program looks like this (code inspired by the [tutorial](#)):

```
import cp.

problem(Variables) =>
    declare_variables(Variables),
    post_constraints(Variables).

main =>
    problem(Variables),
    solve(Variables).
```

Instead of `cp` we can use another solver, e.g. `mip` or `sat`. If the problem is parametrized, then we can pass the `Parameters` as an argument, e.g.

```
problem(Variables, Parameters) or we can have a separate function
get_instance(Parameters) = Variables .
```

## Introductory examples

### Example: Chinese remainder

After an indecisive battle, general Han Xin wanted to know how many soldiers of his 42000-strong army remained. In order to prevent enemy spies hidden among his soldiers to learn the number, he decided to use modular algebra: He ordered his soldiers to form rows of 5 and 3 soldiers remained. Then rows of 7; 2 remained. Then rows of 9; 4 remained. Then rows of 11; 10 remained. Finally, rows of 13; 1 remained.

- What are the **parameters** of our problem?
- Identify the decision variables: type, domains (as small as possible).
- What are the constraints? (Are there some implicit constraints?)
- Is this a satisfaction or an optimization problem?
- Find a solution using Picat.
- Is the last condition (rows of 13) needed?

```
In [1]: %load_ext ipicat
```

Picat version 3.9

```
In [2]: %%picat -n chinese
import cp.
```

```
chinese(X) =>
    % variables
    X :: 0..42000,

    % constraints
    X mod 5 #= 3,
    X mod 7 #= 2,
    X mod 9 #= 4,

    X mod 11 #= 10.

%     X mod 11 #= 10,
%     X mod 13 #= 1.
```

```
In [3]: %%picat
main =>
    chinese(X),
    solve(X),
    println(X).
```

373

```
In [4]: %%picat
main =>
    chinese(X),
    solve_all(X) = Solutions,
    foreach (Solution in Solutions)
        println(Solution)
    end.
```

373  
3838  
7303  
10768  
14233  
17698  
21163  
24628  
28093  
31558  
35023  
38488  
41953

```
In [5]: !picat chinese-remainder/chinese-remainder
```

35023

```
In [6]: !picat chinese-remainder/chinese-remainder 5 [2,3] [1,2]
```

5

```
In [7]: !picat chinese-remainder/chinese-remainder 42000 [5,7,9,11] [3,2,4,10]
```

373  
3838  
7303  
10768  
14233  
17698  
21163  
24628  
28093  
31558  
35023  
38488  
41953

```
In [8]: !cat chinese-remainder/chinese-remainder.pi
```

```

import cp.

chinese(X, Parameters) =>
    % parameters
    [Max, Primes, Moduli] = Parameters,

    % here we could test input data
    if Primes.length != Moduli.length then
        throw(illegal_arguments)
    end,

    % variables
    X :: 0..Max,

    % constraints
    foreach(I in 1..Primes.length)
        X mod Primes[I] #= Moduli[I]
    end.

solve_and_output(Parameters) =>
    chinese(X, Parameters),
    solve_all(X) = Solutions,
    foreach (Solution in Solutions)
        println(Solution)
    end.

main =>
    Parameters = [42000, [5,7,9,11,13], [3,2,4,10,1]],
    solve_and_output(Parameters).

main(Parameters_as_Strings) =>
    Parameters = map(parse_term, Parameters_as_Strings),
    solve_and_output(Parameters).

```

## Example: SEND + MORE = MONEY

Solve the crypt-arithmetic puzzle (each letter represents a different base-10 digit, S and M are nonzero):

```

| SEND + MORE = MONEY

```

```
In [9]: !picat send-more-money/send-more-money
```

```
[9,5,6,7,1,0,8,2]
```

```
In [10]: !cat send-more-money/send-more-money.pi
```

```

import cp.

send_more_money(Digits) =>
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: 0..9,
    S #!= 0,
    M #!= 0,

    % digits are all different: naive
    % foreach(I in 1..Digits.length, J in I+1..Digits.length)
    %     Digits[I] #!= Digits[J]
    % end,

    % digits are all different: using a global constraint (much better propagation!)
    all_different(Digits),

    % arithmetic
    1000 * S + 100 * E + 10 * N + D
+   1000 * M + 100 * O + 10 * R + E
#= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.

main =>
    send_more_money(Digits),
    solve(Digits),
    println(Digits).

```

All constraint languages are somewhat similar, see the included models in `send-more-money/models-in-other-languages/`: one in the C++-based solver Gecode, two in SICStus Prolog, and two in the high-level modeling language MiniZinc:

```

In [11]: !ls send-more-money/models-in-other-languages/
          #!cat send-more-money/models-in-other-languages/*

```

```

send-more-money2.mzn  send-more-money.cpp  send-more-money.pl
send-more-money2.pl   send-more-money.mzn

```

## Exercises

### Exercise: Pythagorean triples

1. Generate all Pythagorean triples up to a given parameter, i.e. positive integers such that  $a^2 + b^2 = c^2$ , where  $a \leq b$  (an example of [symmetry breaking](#)).
2. Modify your program to accept the flag `-c` to output the number of solutions.

```

picat pythagorean 42
picat pythagorean -c 42

```

### Exercise: Send more carry bits

Write a better constraint model for the SEND+MORE=MONEY crypt-arithmetic puzzle based on carry bits.

Why is it better?

Some letters can be computed from other letters and invalidity of the constraint can be checked before all letters are known. (from R. Barták's tutorial in Prolog, see the code)

If we don't study the mistakes of the future, we're bound to repeat them for the first time. (Ken M)