Constraint Programming

Constraint Programming is a powerful framework for problem satisfaction and optimisation. As with similar formalisms like *Linear Programming*, the key skill consists in expressing a problem in such a way that efficient algorithms may be implemented.

Requirements

- You should read the materials **before** the lab session.

At the end of the lab session, you should be able to:

- model a simple problem using constraint programming;
- use a key-in-hand solver, implement a model and get the solution;
- explain the highlights of *depth-first search* solving methods.

Definition

A *constraint satisfaction problem* consists of:

- a finite set of variables $x_1, ...x_n$;
- a corresponding set of finite domains $d_1, ...d_n$;
- a finite set of constraints over subsets of variables, i.e. relations expressing *which attributions of variables to values conflict with each other.*

The objective is to find an instantiation of values to variables which satisfies all the expressed constraints.

Example:

A problem of two variables $x, y \in \{0, 1, 2\}$ subject to $x \neq y$ (the constraint) is a constraint satisfaction problem.

Notes:

- Unlike *Linear Programming*, there is no restriction on the nature of constraints as long as the solver is able to determine for any instantiation if it violates any of the constraints stated.[1]
- It is **not** possible to manipulate infinite domains—not to say continuous.
- In theory, the domains can be formed of *anything*. However, it is a common practice to associate these pieces of *anything* with finite sets of integers.

[1] If you can explain the constraint with words in proper English, you can probably state it using constraint programming.

Illustrative problems

1. Which numbers can we use to replace the following letters so that the addition stands true?

   Each letter can be associated to a variable taking values between 0 and 9. All variables are linked by the following arithmetic constraint $\mathsf{S} \times 1000 + \mathsf{E} \times 100 + \ldots = \ldots + \mathsf{E} \times 10 + \mathsf{Y}$.

```
      SEND
  +   MORE
  =  MONEY
```

2. Place 8 queens on a chess board so that no two queens attack each other.

   Eight variables q_i represent the position of the queen placed on the row labeled i. All variables must be different (not on the same column). Also, no two queens may be placed on the same ($\nearrow$ and $\searrow$) diagonal.

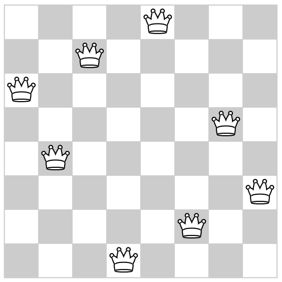
Figure 1: A solution to the 8-queen problem.

3. How to colour each region of the attached map, so that no two neighbouring regions hold the same colour? The problem consists in finding a colouring that minimises the number of colours used to paint the map.

   Each region is mapped to a colour (integer) taking values between 1 and n. A different ($\neq$) constraint is stated over all regions that are neighbouring.

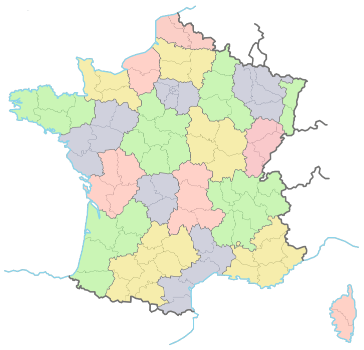
Figure 2: An (optimal?) colouring of administrative regions of France.

Note the mathematical theorem stating that at most four colours are necessary for a proper colouring of such maps (graphs).

We can use $n = 4$ for a start and choose to decrease n until no solution is found. ☞ *constraint satisfaction problem*

A more complicated—albeit general—approach would be to add an minimising criterion (optimisation) on the total number of colours used in a satisfying colouring. ☞ *constraint optimisation problem*

⏭ see Definition on page 4

The resolution process

Constraints as we state them do not necessarily apply to all variables which we use to define the problem.

Each constraint is associated with:

- its *scope*, i.e. the set of variables it applies on;
- its *arity*, i.e. the number of variables it applies on.

- $c_1 = a \neq b$:
  - scope: $\{a, b\}$
  - arity: 2
- $c_2 = b \neq c \mid a + c \leq 7$:
  - scope:$\{a, b, c\}$
  - arity: 3

A common resolution process implements a *depth-first search* algorithm in a tree that is built *on the fly*:

- We first choose an ordering over the variables. We shall name them $x_1, x_2, \cdots x_n$;
- The first branch of the first level of the tree maps to the instantiation of the first possible value of variable x_1;
  From this node, the first branch (of the second level) of the tree maps appends to the preceding *partial instantiation* the instantiation of the first possible value for variable x_2;
- At each node (a.k.a. partial instantiation), the algorithm checks the validity of all constraints whose scope is included in the set of already assigned variables:
  – if no conflict is raised, the search goes *deeper* and a new variable is assigned;
  – if a conflict is raised, the search goes *backward* and invalidates the assumptions already made. ☞ *backtracking, fig. 5*
- The algorithm stops when it reaches a leaf (a.k.a. full instantiation) which violates no constraint.

Take enough time to understand the process of depth-first search with figures 3 through to 6.

Notes:

- The choice of the variable and/or value order can have a significant impact on the size of the tree, i.e. on the resolution time. Some heuristics (e.g., choose first the variable with the smaller domain) may yield better performance.
- Do keep in mind that no tree is actually built in memory: the branching is made *on the fly* and the tree representation only illustrates the algorithm.

- $x_1, x_2, x_3 \in \{0, 1, 2, 3\}$
- $c_1 : x_1 \neq x_2$
- $c_2 : \sum x_i = 3$
- $c_3 : x_3 \geq 1$

Figure 3: Sample problem

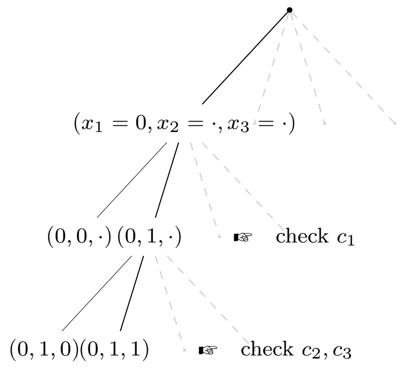

Figure 4: Illustration of the mapping between the tree-like view and partial instantiations.

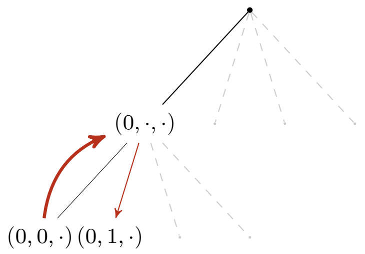

Figure 5: At partial instantiation $(0, 0, \cdot)$, constraint $c_1 : x_1 \neq x_2$ is violated: the algorithms *backtracks*.

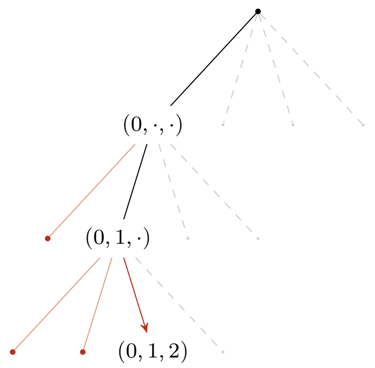

Figure 6: A solution is found that violates no constraint.

A word about arc-consistency

In the worst case, the tree is of an exponential size: efficient algorithms try to prune sub-trees *as early as possible* in the depth-first search process.

The basic backtracking algorithm we implemented on the toy problem shows several issues:

- the domain of x_3 should be limited to $\{1, 2, 3\}$ to avoid countless instantiations of value 0 to variable x_3 and subsequent backtracking.
- a similar domain reduction should be implemented on *binary constraints*[2]. If $c_1 : x_1 \neq x_2$, assigning value 0 to x_1 should remove 0 from the domain of x_2 in the sub-tree rooted in $(0, \cdot, \cdot)$.

[2] A binary constraint has a scope of 2 variables (arity 2).

A basic optimisation consists of *maintaining arc-consistency*, i.e. pruning the domains of variables which are not yet assigned but linked to assigned variables through binary constraints.

We can illustrate the benefits of this *forward-checking* approach with the 8-queen problem. When the first (red) queen is placed, we prune the domain of all the other queens (red dots). Then, instead of trying position 1 and 2, the second (green) queen is directly placed on the first value of its subsequent domain, i.e. position 3.

When the third (blue) queen is placed, the domain of the sixth queen is reduced a single value (i.e. position 4); however, the placement of the fourth (violet) queen eliminates all values of the domain of this queen. A backtrack can be triggered at that point.

Note that with this arc-consistency technique, the first backtrack occurs for instance $(1, 3, 5, 2, \cdot, \cdot, \cdot, \cdot)$. Without this optimisation, we would have already backtracked 7 times before reaching this partial instantiation, and would still need to make several attempts/backtracks for queens #5 and #6 before realising they lead to no solution.

There are many algorithms designed to maintain arc-consistency and detect irrelevant sub-trees as soon as possible.

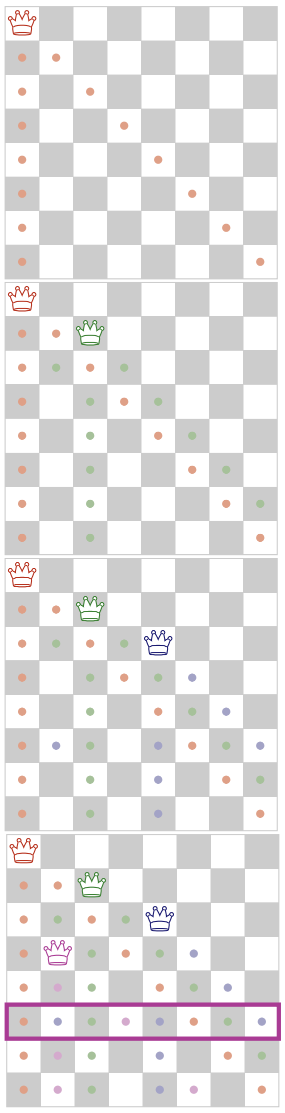
Figure 7: How to exploit arc-consistency information to prune the depth-first search process.

Branch & Bound

A *constraint optimisation problem* consists of:

- a finite set of variables $x_1, ...x_n$;
- a corresponding set of finite domains $d_1, ...d_n$;
- a finite set of positive functions over subsets of variables.

The positive functions may evaluate to $+\infty$ and be equivalent to constraints in the previous definition. The objective is to find an instantiation of values to variables which minimises $\sum f_i$.

The *Branch & Bound* algorithm is an adaptation of depth-first search for optimisation problems. As it parses through the tree, the algorithm shall compute and maintain:

- a *lower bound*, LB, partial evaluation at a node of the tree, i.e. $\sum f_i$ for all f_i whose scope is included into the current partial instantiation;
- an *upper bound*, UB, full evaluation of the better solution found.

At the beginning of the algorithm, $\mathsf{LB} = 0$ and $\mathsf{UB} = +\infty$.

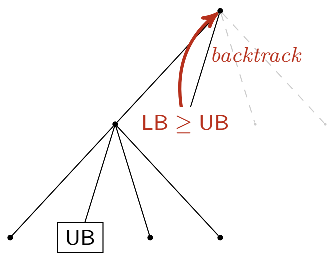

Figure 8: *Branch & Bound*: when a partial instantiation evaluates worse than UB, a backtrack is triggered.

Symmetries

Consider the n-queen problem. A chessboard that is rotated by 90°, 180° or 270° yields queens positions that are different yet equivalent.

If the former chessboard is a solution to our problem, the latter also is. Conversely, if the former violates any constraint, the latter does so as well. A mirrored chessboard follows the same logics.

Formally speaking, a symmetry may be a permutation on the variables ($q_1 \rightleftharpoons q_4$) and/or on the values ($1 \rightleftharpoons 4$) of the problem.

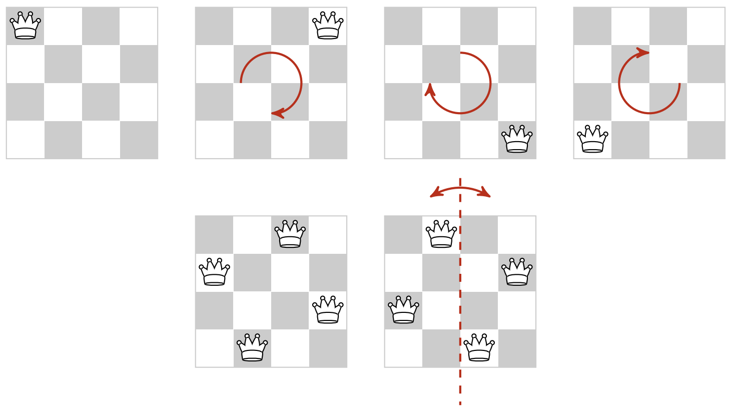

Figure 9: Symmetries in the n-queen problem

There are several ways to exploit the symmetries of a problem:

- *reformulate a problem* into a simpler one without symmetries. The tree to be searched is of a smaller size than the symmetrical one, leading to significantly better performance.
  As a matter of fact, it is important to find a good balance when writing models: avoid symmetries yet still be readable.

- *add constraints* to the problem so as to break the symmetries. The constraints may be added statically (before the search) or dynamically (during the search by the solver).
  Tip: when several variables x_i can be swapped in a problem, consider constraining them with an ordering: $x_1 \leq x_2 \leq \cdots x_n$.

- the solver may dynamically *use symmetry information during search*: if the depth-first search process hits a partial instantiation that is symmetrical to an instantiation which already triggered a backtrack, then it is of no use to go deeper: the solver backtracks.

The bottom line when considering symmetries is to only try to exploit them if detecting them costs less than the speedup you can get from exploiting them.

Exercices

- Two constraints may be missing in the suggested model of the "Send more money" problem. Add them to the model.

⏮ see page 2

- Suggest a different model of the problem. You may need to add *hidden* variables that do not appear in the definition of the problem, yet are involved in the constraints.

  ☞ *Think of how you compute additions by hand.*

- Find an empty sudoku grid on the Internet. Model the problem and keep your draft at hand: you will be able to implement it after the lab session.

- The golf tournament problem can be stated as follows:

  In their club, there are 32 golfers, each of whom play golf once a week, and always in groups of 4. They would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion.

  List all symmetries you can think of when modelling the problem.