Constraint Programming:

Solving CSP's

Eric MONFROY

IRIN, Université de Nantes

Objectives

- intuitive notion of CSP solving
- some more definitions about CSP’s
- constraint programming basic framework
- the different steps in constraint solving

Intuitive constraint solving

Constraint solving

Given a constraint c, the following problems can be studied :

- *satisfaction* : is the constraint c satisfiable ?
  (Is there a valuation of variables of c such that c is true ?)
- *solution* : if c is satisfiable, produce one, several, all solutions
- *optimization* : produce the/an optimal solution (concept to be defined)
- *simplification* : transform c into an *equivalent constraint* (i.e., with the same solution space)

We focus on the first two problems

Solver properties

- a solver is *complete* if it can always answer by yes or no for a CSP
- a solver is *correct* if it computes only solutions
- a solver is *reliable* (or *validated*) if it computes all the solutions of a problem

over real numbers : difficult to get completeness and correctness

Solving CSP

Theoretic : trivial, systematic exploration of the search space (look back) ! ! !

- *Generate and test* : generate an instantiation for all variables, and then test wheter constraints are satisfied or not
- *Backtracking* : incremental generation of instantiations.

  Test satisfiability of constraints whose variables are instanciated.

  In case of success : instanciate new variables.

  In case of failure : undo the most recent instanciation, and make a new instanciation.

  - *thrashing* : repeated failures caused by the same reasons
  - conflicting values are not memorized during backtracking
- *local consistency* : values that do not satisfy all constraints are removed from domain variables

Solving CSPs : look-back

Look back : variables are instanciated, and “instanciated” constraints are tested

- non-incremental version : *generate and test*
- incremental version : *backtracking*

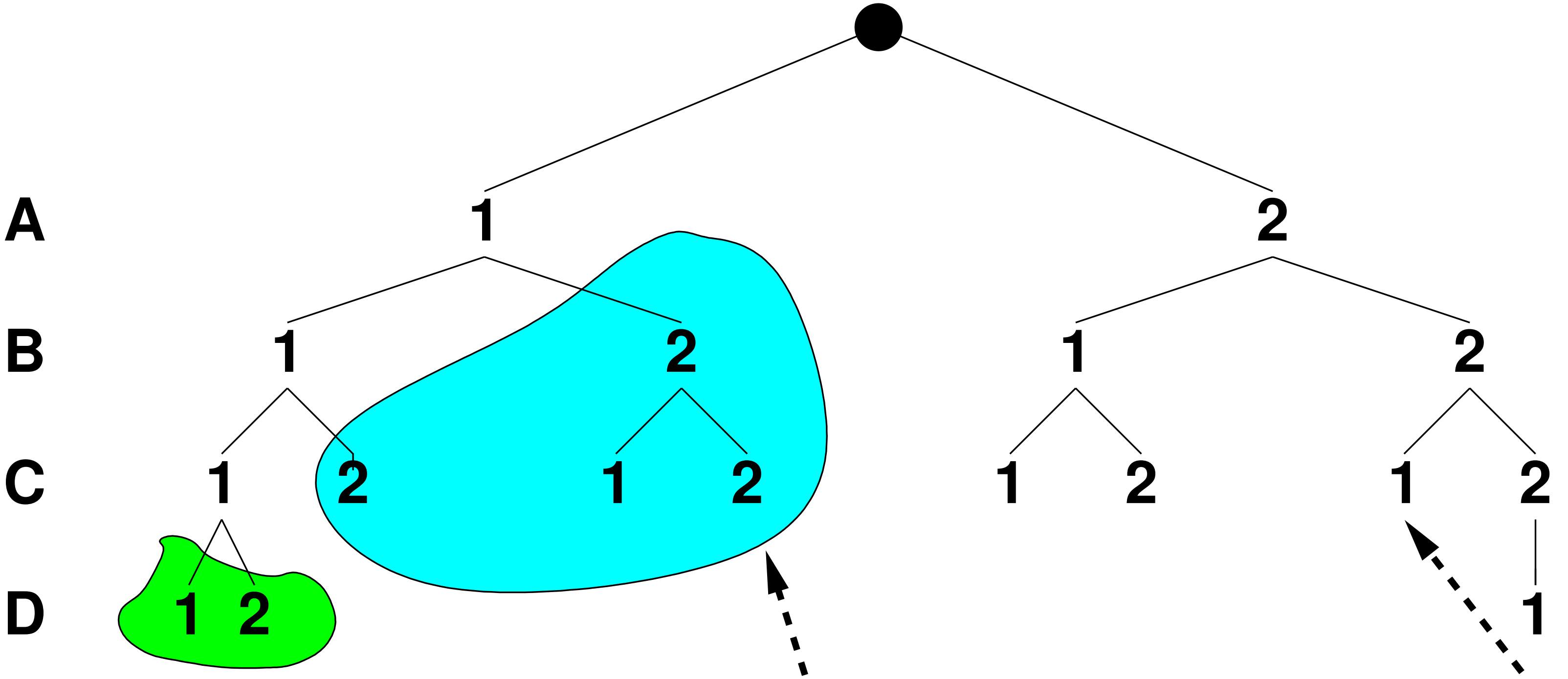
☺ Complete and correct

☹ Inefficient and costly

clever alternatives : *backjumping*, *backmarking*

Problem of the *look back*

$A \in \{1,2\}, B \in \{1,2\}, C \in \{1,2\}, D \in \{1,2\}$
$A > D \wedge B = C \wedge A = C$



thrashing (forget reasons for a conflict)

redundant tests of constraints

late detection of conflicts

Solving CSPs : CSP reduction

basic idea : from a given CSP, find an *equivalent* CSP with smaller domains (smaller search space, same solution space)

- consider each atomic constraint separately
- filter domains of variables and eliminate *inconsistent* values

☺ active use of the constraints. Many values vialoting constraints are removed

☹ Incomplete

→ **constraint propagation : replace a CSP by an equivalent and simpler one ; proceed by repeated reductions of domains/constraints**

Solving CSPs : example of CSP reduction

$$\{and(x, y, z), or(t, u, x);$$
$$x \in [0, 1], y \in [0, 1], z \in [1], t \in [0, 1], u \in [0, 1]\}$$

$$\equiv$$

$$\{and(x, y, z), or(t, u, x);$$
$$x \in [1], y \in [1], z \in [1], t \in [0, 1], u \in [0, 1]\}$$

reduction of x and y domains using the constraint $and(x, y, z)$ and the initial domains of x, y, and z

Solving CSPs : propagation and split

split : cut a CSP into sub-CSP's (and thus smaller)

basic idea : interleave propagation and split of CSP's

why ? from a smaller CSP, propagation can act again

1. constraint propagation
2. split
3. goto 1

☺ active use of the constraints

☺ complete

Solving CSPs : example

$$\{and(x,y,z), or(t,u,x);\ x \in [0,1], y \in [0,1], z \in [1], t \in [0,1], u \in [0,1]\}$$

$\equiv$ **(propagation : and)**

$$\{and(x,y,z), or(t,u,x);\ x \in [1], y \in [1], z \in [1], t \in [0,1], u \in [0,1]\}$$

$\equiv$ **(split t)**

$\{\ldots;\ x,y,z \in [1], t \in [0], u \in [0,1]\}$ **or** $\{\ldots;\ x,y,z \in [1], t \in [1], u \in [0,1]\}$

$\equiv$ **(propagation : or)** $\equiv$ **(no propagation)**

$\{\ldots;\ x,y,z \in [1], t \in [0], u \in [1]\}$ **or** $\{\ldots;\ x,y,z \in [1], t \in [1], u \in [0,1]\}$

$\equiv$ **(split u)**

$\{and(x,y,z), or(t,u,x);$
$x,y,z,t \in [1], u \in [1]\}$

or

$\{and(x,y,z), or(t,u,x);$
$x,y,z,t \in [1], u \in [0]\}$

Constraint solving

Projections

Given a sequence of variables $X = x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$

Consider :

- an element $d = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$
- and a sub-sequence $Y = x_{i_1}, \ldots, x_{i_l}$ of X

Let denote $d_{i_1}, \ldots, d_{i_l}$ by $d[Y]$.

$d[Y]$ is called the *projection* of d on Y

(note that $d[x_l] = d_l$)

Solution

Consider a CSP $\mathcal{P} = \{C_1, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n\}$.

$d_1, \ldots, d_n \in D_1 \times \ldots \times D_n$ is a *solution* of $\mathcal{P}$ if and only if :
for each C_l of $\mathcal{P}$ on Y (a sub-sequence of $x_1, \ldots, x_n$)

$$d[Y] \in C_l$$

Equivalence of CSP's

- two CSP's $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent* if they have the same solution space
- two CSP's $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent w.r.t. the sequence of variables* X iff :

$$\{d[X] \mid d \text{ is a solution to } \mathcal{P}_1\} = \{d[X] \mid d \text{ is a solution to } \mathcal{P}_2\}$$

- a csp $\mathcal{P}$ is equivalent w.r.t. a sequence of variables X to a *union* of CSP's $\mathcal{P}_1, \ldots, \mathcal{P}_n$ if :

$$\{d[X] \mid d \text{ is a solution to } \mathcal{P}\} = \bigcup_{i=1}^{n} \{d[X] \mid d \text{ is a solution to } \mathcal{P}_i\}$$

Solved and failed CSP's

- a constraint C on $y_1, \ldots, y_l$ with domains $D_1, \ldots, D_n$ is *solved* if $C = D_1 \times \ldots \times D_n$
- a CSP $\{C_1, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n$ is *solved* if each C_i ($i \in [1..l]$) is solved and none of the D_j ($j \in [1..n]$) is empty
- a csp $\{C_1, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n$ is *failed* if one of the C_i is the false constraint (generaly noted $\perp$) or one of the D_j is empty.

Constraint solving framework

```
solve(CSP) :
        while not finished do
            pre-process
            constraint propagation
            if happy
              then finished=true
              else split
                  part-of search
            endif
        endwhile
```

where part-of search consists in calls to the solve function

Remark : part-of search is one of the mechanisms defining search

Pre-process

transform constraints into a desired form :

- from which reductions (constraint propagation) can be performed (e.g., primitive constraints)
- from which reductions are stronger (e.g., dependency problem for reals)
- on which the solver is more efficient (redundancies, symetries, order of constraints, graph representation, . . .)

Pre-process example : decomposition

decomposing complexe constraints into primitive constraints (for which reductions can be performed) :

Example :

$3 * x + y + z * t = 6$

becomes

$3 * x = \alpha_1 \wedge \alpha_1 + y = \alpha_2 \wedge z * t = \alpha_3 \wedge \alpha_2 + \alpha_3 = 6$

☺ more easy to implement (no heavy symbolic manipulations)

☹ less reduction capacity (cf. alldiff and its decomposition)

Happy

Depends on the type of the desired solving :

- find a solution
- find all solutions
- find the optimal solution (global optimum)
- find a good solution (local optimum)
- find that there is no solution (insatisfiable CSP)
- find a “good” simplification (normal form to generate solutions, good approximation of solution)

Split

- split a CSP into smaller CSP’s s.t.
  the union of smaller CSP’s is equivalent to the initial one
- why ? propagation can act again on smaller CSP’s
- to obtain a complete solver
- two types of split :
  - split a domain (most common)
  - split a constraint

Constraint spliting

- replace a constraint by “smaller” constraints
- example : disjunction

  replace $\{C_1, \ldots, C_{i,1} \vee C_{i,2}, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n\}$

  by the two CSP’s (union of CSP’s)

  $\{C_1, \ldots, C_{i,1}, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n\}$,

  $\{C_1, \ldots, C_{i,2}, \ldots, C_l;\ x_1 \in D_1, \ldots, x_n \in D_n\}$

Domain spliting

- replace a domain by a union of “smaller” domains
- general form : (bisection : 2 CSP’s, split in the middle)

  replace $\{\mathcal{C};\ \ldots, x_i \in D_i, \ldots\}$

  by the CSP’s (union of CSP’s)

  $\{\mathcal{C};\ \ldots, x_i \in D_{i_1}, \ldots\}$, . . . , $\{\mathcal{C};\ \ldots, x_i \in D_{i_m}, \ldots\}$
  with $\bigcup_{j=1}^{m} D_{i_j} = D_i$
  (better if pairwise disjoint $D_{i_j} \cap D_{i_k} = \emptyset$ for all j and k)
- examples : labeling (enumeration)

  replace $\{\mathcal{C};\ \ldots, x_i \in D_i, \ldots\}$

  by the two CSP’s (union of CSP’s)

  $\{\mathcal{C};\ \ldots, x_i \in \{d_i\}, \ldots\}, \{\mathcal{C};\ \ldots, x_i \in D_i \setminus \{d_i\}, \ldots\}$

Spliting strategies

- theory : not important
- practice : very important for efficiency
- strategies based on :
  - the variable to be split
  - where to split (e.g., bisection)
  - which value for labeling
  - the constraint to be split
- examples :
  - most constrained variable (that appears the most often)
  - largest domain first (variable with the largest domain)
  - largest/smallest/middle value of a domain

Part-of search

- part of the search mechanism
  (according with propagation and split)
  → exploration of the search space
- practice : very important for efficiency
- manage sub-CSP's
- select the CSP's to explore w.r.t. the desired type of solving (one, all, optim, . . .)

Part-of search

Numerous techniques :

- bactracking
- intelligent backtracking
  - backjumping
  - backmarking
- branch and bound (optimization)
- branch and infer
- when combined with constraint propagation
  - forward checking
  - partial look ahead
  - full look ahead

Backtracking

- if no propagation : bactracking a la Prolog
- with propagation :

  depending on the propagation, can lead to :
  - forward checking
  - partial look ahead
  - full look ahead
- in all cases, give a search tree s.t.
  - nodes are dynamically generated (split)
  - a node = a CSP
  - leaves are failed or solved CSP's

Constraint propagation

- replace a CSP by a CSP which is :
  - equivalent (same set of solutions)
  - “smaller” (domains are reduced)
  - “simpler” (constraints are reduced)
- constraint propagation mechanism :

  repeatedly reduce domains or constraints
- can be seen as a fixed point of application of reduction functions
  - reduction function to reduce domains or constraints
  - can be seen as an abstraction of the constraints by reduction functions

Constraint propagation : reducing constraints

- Generally :
  - adding new (redundant) constraints
  - simplifying constraints (e.g., arithmetic simplification)
- example : transitivity

  Reduction function :

  $$x < y, y < z \quad \rightarrow \quad x < y, y < z, x < z$$

  the CSP $\{\ldots, x < y, \ldots, y < z, \ldots;\ \mathcal{D}\}$

  can be reduced to

  the CSP $\{\ldots, x < y, \ldots, y < z, \ldots,$ **x<z**$;\ \mathcal{D}\}$

Constraint propagation : reducing domains (1

- Generally :
  - reduce domains using constraint and domains
  - $\rightarrow$ reduce the search space
- generic domain reduction :
  - Given a constraint C over $x_1, \ldots, x_n$ with domains $D_1, \ldots, D_n$
  - select a variable x_i to be reduced
  - delete from D_i all values for x_i that do not participate in a solution of C

Constraint propagation : reducing domains (2

- example : linear equalities on integer
  - reduction function :

$$x < y, x \in [l_x..r_x], y \in [l_y..r_y]$$

$$\longrightarrow$$

$$x < y, x \in [l_x..min(r_x, r_y - 1)], y \in [max(l_y, l_x + 1)..r_y]$$

  - example of use

    the CSP

    $\{\ldots, x < y, \ldots;\ \ldots, x \in [10..20], \ldots, y \in [0..15]\}$

    - can be reduced to the CSP

    $\{\ldots, x < y, \ldots;\ \ldots, x \in [10..\mathbf{14}], \ldots, y \in [\mathbf{11}..15]\}$

Constraint propagation mechanism

- repeated application of reductions
- try to apply only useful reductions
- stop
  - when a *local consistency* notion is reached (e.g., arc, node, hyper-arc consitency)
  - or when reduction becomes inefficient (e.g., cycling weak reductions)
  - or when a domain is empty (failed CSP)