Foundations of constraint satisfaction

5

Roman Barták
Charles University in Prague

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

How to solve the constraint problems?

So far we have two methods:

search

- complete (finds a solution or proves its non-existence)
- too slow (exponential)
  explores "visibly" wrong valuations

consistency techniques

- usually incomplete (inconsistent values stay in domains)
- pretty fast (polynomial)

Share advantages of both approaches - **combine** them!

- label the variables step by step (backtracking)
- maintain consistency after assigning a value

Do not forget about **traditional solving techniques**!

Linear equality solvers, simplex …

such techniques can be integrated to global constraints!

Core search procedure - depth-first search

The basic constraint satisfaction technology:

- label the variables step by step
  the variables are marked by numbers and labelled in a given order
- ensure consistency after variable assignment

A skeleton of search procedure

```
procedure Labelling(G)
    return LBL(G,1)
end Labelling

procedure LBL(G,cv)
    if cv>|nodes(G)| then return nodes(G)
    for each value V from D_cv do
        if consistent(G,cv) then
            R ← LBL(G,cv+1)
            if R ≠ fail then return R
        end if
    end for
    return fail
end LBL
```

A „hook" for consistency procedure

Look back techniques

"Maintain" **consistency among the already labelled variables**.

„look back" = look to already labelled variables

What's result of consistency maintenance among labelled variables?

a conflict (and/or its source - a violated constraint)

Backtracking is the basic method of look back.

Backward consistency checks

```
procedure AC-BT(G,cv)
    Q ← {(V_i,V_cv) in arcs(G),i<cv}     % arcs to labelled variables.
    consistent ← true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        consistent ← not REVISE(V_k,V_m)
    end while
    return consistent
end AC-BT
```

When a value is deleted, the domain is empty

Backjumping & comp. uses information about violated constraints.

Forward checking

It is better to prevent failures than to detect them only!

Consistency techniques can remove incompatible values for future (=not yet labelled) variables.

Forward checking ensures consistency between the currently labelled variables and the variables connected to it via constraints.

Forward consistency checks

```
procedure AC-FC(G,cv)
    Q ← {(V_i,V_cv) in arcs(G),i>cv}     % arcs to future variables
    consistent ← true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) then
            consistent ← not empty D_k
        end if
    end while
    return consistent
end AC-FC
```

Empty domain implies inconsistency

Partial look ahead

We can extend the consistency checks to more future variables!

The value assigned to the current variable can be propagated to all future variables.

Partial lookahead consistency checks

```
procedure DAC-LA(G,cv)
    for i=cv+1 to n do
        for each arc (V_i,V_j) in arcs(G) such that i>j & j≥cv do
            if REVISE(V_i,V_j) then
                if empty D_i then return fail
        end for
    end for
    return true
end DAC-LA
```

Notes:

In fact DAC is maintained (in the order reverse to the labelling order).
Partial Look Ahead or **DAC - Look Ahead**

It is not necessary to check consistency of arcs between the future variables and the past variables (different from the current variable)!

Full look ahead

Knowing more about far future is an advantage!

Instead of DAC we can use a full AC (e.g. AC-3).

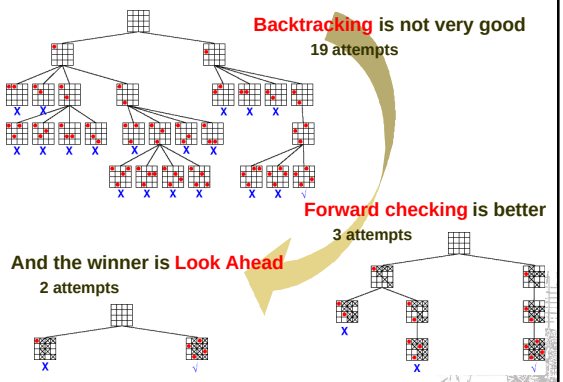
Full look ahead consistency checks

```
procedure AC3-LA(G,cv)
    Q ¬ {(Vi,Vcv) in arcs(G),i>cv}          % start with arcs going to cv
    consistent ¬ true
    while not Q empty & consistent do
        select and delete any arc (Vk,Vm) from Q
        if REVISE(Vk,Vm) then
            Q ¬ Q È {(Vi,Vk) | (Vi,Vk) in arcs(G),i ¹ k,i ¹ m,i>cv}
            consistent ¬ not empty Dk
        end if
    end while
    return consistent
end AC3-LA
```

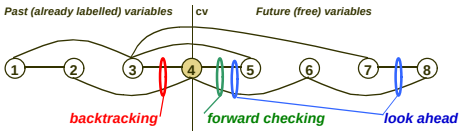
Notes:

- The arcs going to the current variable are checked exactly once.
- The arcs to past variables are not checked at all.
- It is possible to use other than AC-3 algorithms (e.g. AC-4)

Comparison of solving methods (4 queens)



Constraint propagation at glance



- Propagating through more constraints remove more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the step.
- Forward Checking does no increase complexity of backtracking, the constraint is just checked earlier in FC (BT tests it later).
- When using AC-4 in LA, the initialisation is done just once.
- Consistency can be ensured before starting search
  Algorithm MAC (Maintaining Arc Consistency)
  AC is checked before search and after each assignment
- It is possible to use stronger consistency techniques (e.g. use them once before starting search).

Variable ordering

Variable ordering in labelling influence significantly efficiency of solvers (e.g. in tree-structured CSP).

What variable ordering should be chosen in general?

FIRST-FAIL principle

„select the variable whose instantiation will lead to failure“

it is better to tackle failures earlier, they can be become even harder

- prefer the variables with smaller domain (dynamic order)
  a smaller number of choices ~ lower probability of success
  the dynamic order is appropriate only when new information appears during solving (e.g., in look ahead algorithms)

„solve the hard cases first, they may become even harder later“

- prefer the most constrained variables
  it is more complicated to label such variables (it is possible to assume complexity of satisfaction of the constraints)
  this heuristic is used when there is an equal size of the domains
- prefer the variables with more constraints to past variables
  a static heuristic that is useful for look-back techniques

Value ordering

Order of values in labelling influence significantly efficiency (if we choose the right value each time, no backtrack is necessary).

What value order for the variable should be chosen in general?

SUCCEED FIRST principle

„prefer the values belonging to the solution“

if no value is part of the solution then we have to check all values

if there is a value from the solution then it is better to find it soon

SUCCEED FIRST does not go against FIRST-FAIL !

- prefer the values with more supporters
  this information can be found in AC-4
- prefer the value leading to less domain reduction
  this information can be computed using singleton consistency
- prefer the value simplifying the problem
  solve approximation of the problem (e.g. a tree)

Generic heuristics are usually too complex for computation.

It is better to use problem-driven heuristics that propose the value!

Constraint optimisation

So far we have looked for feasible assignments only.

In many cases the users require optimal assignments where optimality is defined by an objective function.

Definition: Constraint Satisfaction Optimisation Problem (CSOP) consists of the standard CSP P and an objective function *f* mapping feasible solutions of P to numbers.

Solution to CSOP is a solution of P minimising / maximising the value of the objective function *f*.

To find a solution of CSOP we need in general to explore all the feasible valuations. Thus, the techniques capable to provide all the solutions of CSP are used.

Branch and bound

Branch and bound is perhaps the most widely used optimisation technique based on cutting sub-trees where there is no optimal (better) solution.

It is based on the **heuristic function** *h* that approximates the objective function.

a sound heuristic for minimisation satisfies h(x)£f(x)
[in case of maximisation f(x)£h(x)]

a function closer to the objective function is better

During search, the sub-tree is cut if

- there is no feasible solution in the sub-tree
- there is no optimal solution in the sub-tree
  bound £ h(x), where *bound* is max. value of feasible solution

How to get the bound?

It could be an objective value of the best solution so far.

BB and constraint satisfaction

Objective function can be modelled as a constraint

looking for the "optimal value" of *v*, s.t. v = f(x)

- first solution is found without any bound on *v*
- next solutions must be better then so far best (*v*<Bound)
- repeat until no more feasible solution exist

Algorithm Branch & Bound

```
procedure BB-Min(Variables, V, Constraints)
    Bound ¬ sup
    NewSolution ¬ fail
    repeat
        Solution ¬ NewSolution
        NewSolution ¬ Solve(Variables,Constraints È {V<Bound})
        Bound ¬ value of V in NewSolution (if any)
    until NewSolution = fail
    return Solution
end BB-Min
```


Some notes on branch and bound

Heuristic *h* is hidden in **propagation through the constraint** v = f(x).

Efficiency is dependent on:

- **a good heuristic** (good propagation of the objective function)
- **a good first feasible solution** (a good bound)
  the initial bound can be given by the user to filter bad valuations

The optimal solution can be found fast
proof of optimality can be long (exploring of the rest part of tree)

The optimality is often not required, **a good enough solution is OK.**

- BB can stop when reach a given limit of the objective function

Speed-up of BB: **both lower and upper bounds are used**

```
repeat
    TempBound ¬ (UBound+LBound) / 2
    NewSolution ¬ Solve(Variables,Constraints È {V£TempBound})
    if NewSolution=fail then
        LBound ¬ TempBound+1
    else
        UBound ¬ TempBound
until LBound = UBound
```


A motivation - robot dressing problem

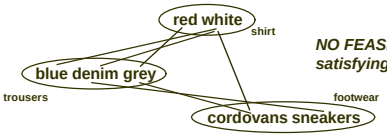
Dress a robot using minimal wardrobe and fashion rules.

Variables and domains:

shirt: {red, white}
footwear: {cordovans, sneakers}
trousers: {blue, denim, grey}

Constraints:

shirt x trousers: red-grey, white-blue, white-denim
footwear x trousers: sneakers-denim, cordovans-grey
shirt x footwear: white-cordovans



NO FEASIBLE SOLUTION satisfying all the constraints

We call the problems where no feasible solution exists **over-constrained problems**.

First solution to the robot dressing problem

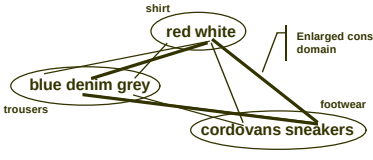
There is no feasible valuation but we need to dress robot!

1) buy new wardrobe
   enlarge the domain of some variable
2) less elegant wardrobe
   enlarge the domain of some constraint
3) no matching of shoes and shirt
   remove some constraint
4) do not wear shoes
   remove some variable

Domain is defined by a unary constraint

All combinations are assumed feasible

Delete the constraint bounding the variable



Partial constraint satisfaction

First let us define a **problem space** as a partially ordered set of CSPs (PS,£), where $P_1 £ P_2$ iff the solution set of P_2 is a subset of the solution set of P_1.

The problem space can be obtained by weakening the original problem.

Partial Constraint Satisfaction Problem (PCSP) is a quadruple
áP_I,(PS,£),M_I,(N,S)ñ

- P is the original problem
- (PS,£) is a problem space containing P
- M is a metric on the problem space defining the problem distance
  M(P,P') could be a number of different solutions of P a P'
  or the number of different tuples in the constraint domains
- N is a maximal allowed distance of the problems
- S is a sufficient distance of the problems (S<N)

Solution to PCSP is a problem P' and its solution such that P' ÎPS and M(P,P')<N. **A sufficient solution** is a solution s.t. M(P,P') £ S.
The optimal solution is a solution with the minimal distance to P.

Partial constraint satisfaction in practice

When solving PCSP we do not explicitly generate the new problems
- an evaluation function *g* is used instead; it assigns a numeric value to each (even partial) valuation
- the goal is to find assignments minimising/maximising g

PCSP is a generalisation of CSOP:

g(x) = f(x), if the valuation x is a solution to CSP

g(x) = ¥, otherwise

PCSP is used to solve:
- over-constrained problems
- too complicated problems
- problems using given resources (e.g. time)
- problems in real time (anytime algorithms)

PSCP can be solved using local search, branch and bound, or special propagation algorithms.

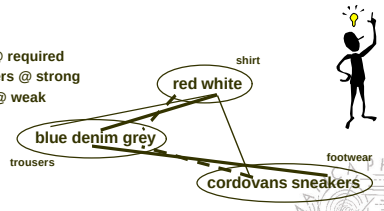
Second solution of the robot dressing problem

It is possible to assign a preference to each constraint to describe priorities of satisfaction of the constraints.

The preference describes a strict priority.

a stronger constraint is preferred to arbitrary number of weaker constraints

shirt x trousers @ required
footwear x trousers @ strong
shirt x footwear @ weak



Constraints marked by a preference make a hierarchy, thus we are speaking about constraint hierarchies.

Constraint hierarchies

Every constraint is labelled by a preference (the set of preferences is totally ordered)
- there is a special preference *required*, marking constraints that must be satisfied (hard constraints)
- the other constraints are preferential, their satisfaction is not required (soft constraints)

Constraint hierarchy H is a finite (multi)set of labelled constraints.

H_0 is a set of the required constraints (the label is removed)

H_1 is a set of the most preferred soft constraints

…

A solution to the hierarchy is an assignment satisfying all the required constraints and satisfying best the preferential constraints.

$S_{H,0} = \{s \mid \forall c \in H_0, c\,s \text{ holds}\}$

$S_H = \{s \mid s \in S_{H,0} \wedge \forall w \in S_{H,0}\ \varnothing \text{better}(w,s,H)\}$

Comparators

Comparing the assignments according to a given hierarchy.
- anti-reflexive, transitive relation that *respects the hierarchy*
- if any assignment satisfies all the constraints till the level k, then every better assignment must satisfy these constraints as well

Error function e(c,s) - how good the constraint is satisfied

predicate error function (satisfied/violated)

metric error function - distance from solution, e(X>5,{X/3}) = 2

Local comparators

compare the assignments using the constraint individually

locally_better(w,s,H) ° $k>0
"i<k "c∈H_i e(c,w)=e(c,s) & "c∈H_k e(c,w)£e(c,s) & $c∈$H_k$ e(c,w)<e(c,s)

Global comparators

aggregate the individual errors at the level via the function g

globally_better(w,s,H) ° $k>0 "i<k g($H_i$,w)=g($H_i$,s) & g($H_k$,w)<g($H_k$,s)

weighted-sum, least-squares, and worst-case methods …

Why should we use CP?

Close to real-life (combinatorial) problems
- everyone uses constraints to specify problem properties
- real-life restriction can be naturally described using constraints

A declarative character
- concentrate on problem description rather than on solving

Co-operative problem solving
- unified framework for integration of various solving techniques
- simple (search) and sophisticated (propagation) techniques

Semantically pure
- clean and elegant programming languages
- roots in logic programming

Applications
- CP is not another academic framework, it is already used in many applications

Final notes

Constraints
- arbitrary relations over the problem variables
- express partial local information in a declarative way

Solution technology
- search combined with constraint propagation
- local search

It is easy to state combinatorial problems in terms of CSP … but it is more complicated to design solvable models.

We still did not reach the Holy Grail of computer programming: *the user states the problem, the computer solves it.*

Constraint Programming is one of the closest approaches to the Holly Grail of programming!