

# **Introduction to Artificial Intelligence: Reasoning with Constraints**

Héctor Geffner

ICREA and Universitat Pompeu Fabra  
Barcelona, Spain

# Recall SAT

- SAT is the problem of determining whether a set of clauses is satisfiable, and if so, determining a satisfying valuation.
- Many problems can be mapped into SAT such as Planning, Scheduling, CSPs, Verification problems etc.
- SAT is an intractable problem (exponential in the worst case unless  $P=NP$ ) yet **very large SAT problems can be solved in practice**
- Best SAT (DP) algorithms not based on either pure case analysis (model theory) or resolution (proof theory), but a combination of both

# Davis and Putnam (DP) Procedure for SAT

- DP uses resolution but in restricted form called **unit resolution** in which **one parent clause** must be **unit clause**
- Unit resolution very efficient (linear) but **not complete** (e.g.,  $q \vee p$ ,  $\neg q \vee p$ ,  $q \vee \neg p$ ,  $\neg q \vee \neg p$ )
- When Unit Resolution gets stuck, DP picks undetermined Var, and **splits** the problem in two: one where Var is true, the other where it is false

DP(clauses)

Unit-resolution(clauses)

if Contradiction, Return False

else if all VARS determined, Return True

\* else pick Non-determined VAR X and

Return DP(clauses + X) OR DP(clauses + NEG X)

- DP method is sound and complete; currently very large SAT problems solved
- Criterion for **variable selection** is critical

# Constraints

- **Constraint Satisfaction Problems (CSPs)** is a useful generalization of SAT model
- $\text{CSP} = \langle \text{Variables}, \text{Domains}, \text{Constraints} \rangle$ 
  - A **solution** is an assignment of values to vars from domains satisfying all constraints
  - A CSP is **consistent** if it has one or more solutions

E.g., first CSP below is consistent; second is not

$$\langle \{X, Y\}, \{D_X, D_Y = [1..10]\}, \{X + Y > 10, X - Y > 7\} \rangle$$

$$\langle \{A, B, C\}, \{D_A, D_B, D_C = [a, b]\}, \{A \neq B, B \neq C, A \neq C\} \rangle$$

## Other Examples of CSPs

- N-queens
- Map coloring
- Stable marriage
- ⋮
- Job shop scheduling
- SAT
- Resource allocation
- . . .

# Algorithms for CSPs: Backtracking

Backtracking refers to DFS traversal of search tree where

- **state space:** nodes  $\sigma$  are **partial assignments**
  - **initial state:**  $\sigma_0$  is **empty assignment**
  - **branching** achieved by selecting unassigned variable  $X$  and extending  $\sigma$  with  $X = x_i$  for each  $x_i$  in domain  $D_X$  of  $X$
  - **dead-ends:** partial assignments  $\sigma$  that violate a constraint
  - **goals:** full (consistent) assignments
- Backtracking search is complete and uses linear memory
- Yet Backtracking is **blind** search, and doesn't scale up
- Need '**heuristic**'  $h$  to predict when partial assignment can lead to solution
- Davis and Putnam method for SAT is **Backtracking + Unit Resolution**

# Heuristics for CSPs: Detecting Inconsistency

- Determining **exactly** whether partial assignment can be extended to consistent solution is **intractable**
- **Constraint propagation algorithms** provide tractable sound but incomplete approximation
- **Analogy:** idea of **constraint propagation** is to make use of the **constraints** to guide the search for a **solution** in CSPs, very much as the idea of **heuristic search** is to make the **goal** guide search for solutions in State Models

# Constraint Propagation: the idea

Given partial assignment  $s = \{X = x, Y = y, \dots\}$

- **Domain filtering or reduction:** remove values of unassigned vars that cannot participate in a solution that extends  $s$
- **Report inconsistency:** when domain of variable becomes empty

**Key Question: How to do CP effectively?**

- many schemes proposed
- all enforce consistency over **relaxed** CSP that involves few variables and constraints only

# Consistency/Filtering Techniques

- Nodes in search tree regarded as **CSPs** where assigned vars  $X = x$  have domains  $D_X = \{x\}$
- Say domain  $D_X$  of variable  $X$  is **minimal** in CSP iff all  $x \in D_X$  participate in solutions to CSP
- Reducing domains to their min is intractable in general, but tractable over suitably **relaxed** CSPs:
  - For example, consider 'sub-CSP's from original CSP
    - involving at most a **single variable**
    - involving at most **two variables**
    - $\vdots$
    - involving at most  $i$  **variables**
  - For such 'relaxed' CSPs, only retain constraints from the original CSP involving selected variables

## i-Consistency: General Definition

- For a given  $i \leq n$ , consider the relaxed CSP corresponding to **each** subset of at most  $i$  variables in the problem
- For each such relaxed CSP, reduce domains eliminating values that cannot participate in a solution
- Keep iterating til fixed point is reached
- Result: is CSP that is **i-consistent**

## Special Cases: Node Consistency

$vars(c)$ : vars in constraint  $c \in Cs$

$|vars(c)|$ : number of vars in  $c \in Cs$

- A constraint  $c$  is **node consistent** if  $vars(c) = \{X\}$  implies  $\forall x \in D_X$   $X = x$  satisfies  $c$
- A CSP is **node consistent** if all  $c \in Cs$  are node consistent

**node consistency = 1-consistency**

## Special Cases: Arc Consistency

- A constraint  $c \in C_s$  is **arc-consistent** if  $vars(s) = \{X, Y\}$  implies
  - $\forall x \in D_X \exists y \in D_Y$  s.t.  $X = x \& Y = y$  satisfies  $c$
  - $\forall y \in D_y \exists x \in D_X$  s.t.  $X = x \& Y = y$  satisfies  $c$
- A CSP is **arc-consistent** if all  $c \in C_s$  are arc-consistent

**2-consistency = arc + node consistency**

## Achieving node and arc consistency

- Achieving NC is straightforward; for AC . . .
- Pick a constraint
- Check AC condition for variables involved
- Remove values that do not comply
- **keep iterating til fixed point reached**

E.g. for

$$X < Y \quad , \quad Y < Z$$
$$D_X = D_Y = D_Z = [1..5]$$

resulting domains are:

$$D_X = [1..3], D_Y = [2..4], D_Z = [3..5]$$

## AC-3: A Simple but effective Algorithm for AC

- For simplicity, assume all constraints **binary**
- Write  $X-Y$  iff there is a constraint between  $X$  and  $Y$
- AC-3 achieves AC over all the 'arcs' in CSP, and every time domain of variable  $X$  changes, arcs  $X-Z$  reconsidered
- Many other AC algorithms; in practice, none better than AC-3 over **general constraints**

AC-3 (..)

```
Achieve Node Consistency
Queue := {(X,Y),(Y,X) | for arcs X-Y}
while Queue is not empty
  (X,Y) := remove pair from Queue
  if REVISE(X,Y)
    Add to Queue pairs (Z,X) for arcs X-Z
```

REVISE(X,Y)

```
Delete := False
for x in D_X
  Supported := False
  for y in D_Y while !Supported
    if x,y satisfies constraint between X,Y
```

```
    Supported := True
  if !Supported
    delete x from D_x
    Delete := true
Return Delete
```

## Backtracking + Arc Consistency

- NC and AC proposed originally as **preprocessing** techniques; i.e., they were applied to original CSP, then resulting **reduced** CSP solved by backtracking
- Often better results by applying NC and AC to **every node** in search tree
- Resulting algorithm known as MAC, for Maintaining Arc Consistency (in MAC, also, after assignment  $V = v$  tried and failed, additional constraint  $V \neq v$  added to  $C_s$ )
- Consistency of higher order usually doesn't pay off; it reduces the search tree further **but** but overhead per node much higher

# Generalized Arc Consistency (GAC) and Global Constraints

- Arc consistency (2-consistency) prunes variable domains by considering **binary constraints only**
- **Non binary constraints** can always be expressed as sets of **binary constraints**, . . .
- Yet better results often obtained on **global constraint**

**DEF:** Say an  $n$ -ary constraint  $c$  is **generalized arc consistent** if the domains of all the vars involved in  $c$  are **minimal** with respect to  $c$  (i.e., for each value  $x_i$  of  $X_i$ , there are values  $x_1, x_2, \dots$ , for the other vars, that together satisfy  $c$ ).

- Achieving GAC is in general exponential in  $n = |vars(c)|$
- Yet for certain constraints, can be done (or approximated) efficiently  
. . .

# Bounds Consistency

- Suitable only for **arithmetic CSPs**, more precisely, for constraints involving numerals and symbols like  $>$ ,  $\leq$ ,  $+$ ,  $\dots$ , and where variables  $X$  take **integer values over finite ranges**  $D_X = [x_{min} \dots x_{max}]$
- Idea: consider consistency of **end points** only  $\dots$

**DEF:** constraint  $c$  is **bounds consistent** iff for each var  $X$  in  $vars(c)$ , for both  $X = x_{min}$  and  $X = x_{max}$ , there are **real values**  $y$  in  $[y_{min} \dots y_{max}]$  for the other variables  $Y$  in  $vars(s)$  that jointly satisfy  $c$

## Bounds Consistency: Example

$$X = 3Y + 5Z$$

- Domains  $D_X = [2 \dots 7]$ ,  $D_Y = [0 \dots 2]$ ,  $D_Z = [-1 \dots 2]$  **not GAC or BC**  
(Bounds Consistent)
- Domains  $D_X = [2 \dots 7]$ ,  $D_Y = [0 \dots 2]$ ,  $D_Z = [0 \dots 1]$  are BC but not GAC
- Domains  $D_X = \{3, 5, 6\}$ ,  $D_Y = [0 \dots 2]$ ,  $D_Z = [0 \dots 1]$  are GAC but not Intervals

# Achieving Bounds Consistency

- BC slightly weaker than GAC but can be computed **analytically without enumeration** by means of simple **propagation rules** obtained from the constraints
- The propagation rules operate on the interval **bounds**; e.g., rules for constraint  $X = Y + Z$

$$\begin{aligned}x_{max} &\leq y_{max} + z_{max} \\x_{min} &\geq y_{min} + z_{min} \\y_{max} &\leq x_{max} - z_{min} \\&\vdots\end{aligned}$$

and change in one bound **propagates** to changes in others

- Similar rules for all arithmetic constraints . . .

## Global Constraints: Specialized GAC

- Consider *alldiff* ( $[X_1, X_2, \dots, X_n]$ ) constraint, meaning that vars  $X_i$  must take different values
- Can be reduced to pairs of inequalities  $X_i \neq X_j, i \neq j$
- AC on reduced binary constraints, weaker than GAC over global constraint; e.g., for  $n = 3$  and  $D_1 = D_2 = D_3 = \{1, 2\}$ , domains AC but not GAC
- Yet GAC over **alldiff** constraints can be achieved efficiently; two main ideas:
  - For any subset of  $i$  vars, their domains must contain  $i$  different elements
  - Satisfying assignments correspond to maximal matchings over bipartite graph

# Constraint Programming Languages

- Extend (conventional) programming languages with
  - primitive arithmetic and global constraints (e.g., **alldiff**)
  - ways for defining new constraints, and
  - (bounds) consistency algorithms
- Some of these languages include
  - Eclipse: very natural extension of Prolog for CP
  - ILOG Solver (commercial): OO library in C++
  - Choco/Claire: a CP library on top of high level language
  - OPL/Ilog (commercial): nice high-level CP/LP modeling language
  - . . .

# Examples

Examples in Eclipse, from Marriot and Stuckey, Programming with Constraints, MIT Press, 1998

```
%%%
%%% Puzzle: send + more = money ...
%%%
smm(S,E,N,D,M,O,R,Y) :-
    [S,E,N,D,M,O,R,Y] :: [0..9],
    constrain([S,E,N,D,M,O,R,Y]),
    labeling([S,E,N,D,M,O,R,Y]).

constrain([S,E,N,D,M,O,R,Y]) :-
    S ## 0, M ## 0,
    alldifferent([S,E,N,D,M,O,R,Y]),
    1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y.
```

# Examples

```
%%%
%%% N-queens
%%%
goal(N, Queens) :-
    length(Queens, N),
    Queens :: [1..N],
    queens(Queens),
    labeling(Queens).

queens([]).
queens([X|Y]) :-
    safe(X,Y,1),
    queens(Y).

safe(_, [], _).
safe(X, [F|T], Nb) :-
    noattack(X,F,Nb),
    Newnb #= Nb + 1,
    safe(X,T,Newnb).

noattack(X,Y,Nb) :-
    Y + Nb ## X, X + Nb ## Y, X ## Y.

% goal for 8 queens
queen8(Queens) :- goal(8, Queens).
```

## More examples in Eclipse

```
%%% Version of stable marriage: partners have to like each other
%%%
likes(kim, maria).
likes(kim, nicole).
likes(kim, erika).
likes(peter, maria).
likes(bernd, nicole).
likes(bernd, maria).
likes(bernd, erika).
%
match(N,M,E) :-
    [N,M,E] :: [kim, peter,bernd], alldifferent([N,M,E]),
            likes(N, nicole), likes(M, maria), likes(E, erika).
%%
%% simple assignment problem: 4 workers wi and 4 products pi,
%% if worker w1 assigned to pi's, productivity 7,1,3,4 ..
%% find assignment of w's to p's to max productivity ...
assign([W1,W2,W3,W4]) :-
    [W1,W2,W3,W4] :: [1..4],
    alldifferent([W1,W2,W3,W4]),
    element(W1, [7,1,3,4], WP1),
    element(W2, [8,2,5,1], WP2),
    element(W3, [4,3,7,2], WP3),
    element(W4, [3,1,6,3], WP4),
    P #= WP1 + WP2 + WP3 + WP4,
    P #>= 19, labeling([W1,W2,W3,W4]).
```

# More Examples: Optimization

```
%%%  
%%% knapsack: first constraint is capacity, second is profit.  
%%% need to find Qty of each item W, P and C ..  
  
% satisfaction version ..  
knapsack(W,P,C) :-  
  [W,P,C] :: [0..9], 4*W + 3*P + 2*C #<= 9,  
  15*W + 10*P + 7*C #>= 30, labeling([W,P,C]).  
  
%%% yields solutions 0,1,3 ; 0,3,0; 1,1,1 ; 2,0,0 for W,P;  
  
% optimization version: maximize profit ..  
knapsack(W,P,C) :-  
  L #= 15*W + 10*P + 7*C,  
  maximize(([W,P,C] :: [0..9], 4*W + 3*P + 2*C #<= 9,  
  15*W + 10*P + 7*C #>= 30, labeling([W,P,C])), L).  
  
%%% yields single solution 1,1,1 with L=32
```

## Examples (Claire+Choco)

```
[queen0(n:integer, all:boolean)
-> let pb := choco/makeProblem("N queens on a N*N chessboard",n),
    // create variables
    queens := list{choco/makeIntVar(pb,"Q" /+ string!(i), 1, n) |
                    i in (1 .. n) } in
  (// diagonal constraints
   for i in (1 .. n)
     for j in (i + 1 .. n)
       let k := j - i in
         ( choco/post(pb, queens[i] != queens[j]),
           choco/post(pb, queens[i] != queens[j] + k),
           choco/post(pb, queens[j] != queens[i] + k) ),
   solveQueenPb(pb,n,all) )]

[solveQueenPb(pb:choco/Problem, n:integer, all:boolean) : void
-> let algo := choco/makeGlobalSearchSolver(all) in
  (choco/setMaxNbBk(algo,10000000),
   choco/attach(algo,pb),
   choco/run(algo),
   if (all & n <= 13)
     (if (choco/getNbSol(algo) != NBQUEENSOL[n])
        error("queen(~S,true) finds ~S solutions instead of ~S",
              n,choco/getNbSol(algo),NBQUEENSOL[n])),
     choco/discardProblem(pb) // v1.010
  )]
```

# Bibliography on Constraints

- Marriot and Stuckey, Programming with Constraints, MIT Press, 1998
- Rina Dechter. Constraint Processing. Morgan Kaufmann, 2003.
- P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press 1989.
- P. Van Hentenryck. OPL: Optimization Programming Language. MIT Press 1999.
- Papers by Rina Dechter, Peter Van Beek, J. Regin, J. F. Puget, Yveas Causeau and Francois Laburthe, etc.
- Conference CP (Constraint Programming), Journal Constraints, . . .