Constraint Satisfaction Problems

Tuomas Sandholm

Carnegie Mellon University

Computer Science Department

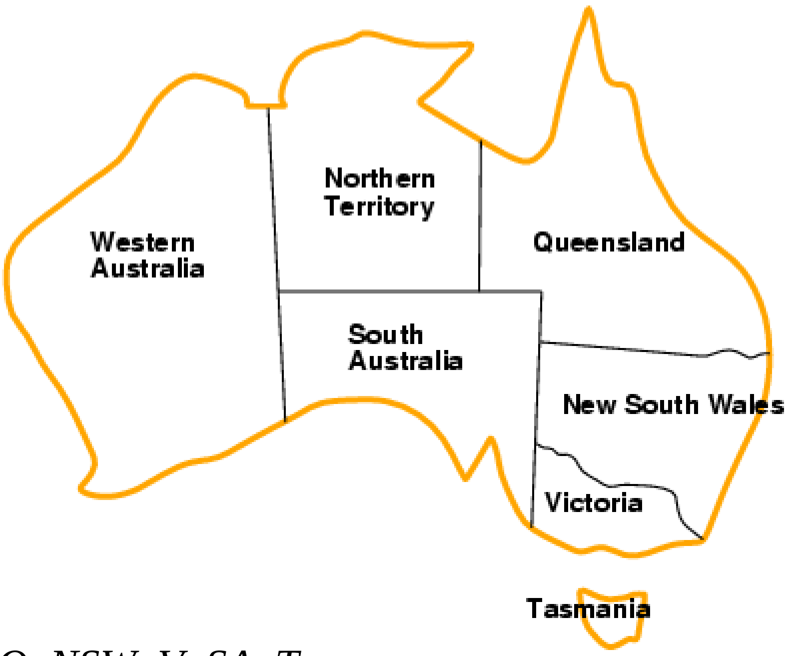
[Read Chapter 6 of Russell & Norvig]

Constraint satisfaction problems (CSPs)

- Standard search problem: state is a "black box“ – any data structure that supports successor function and goal test
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables

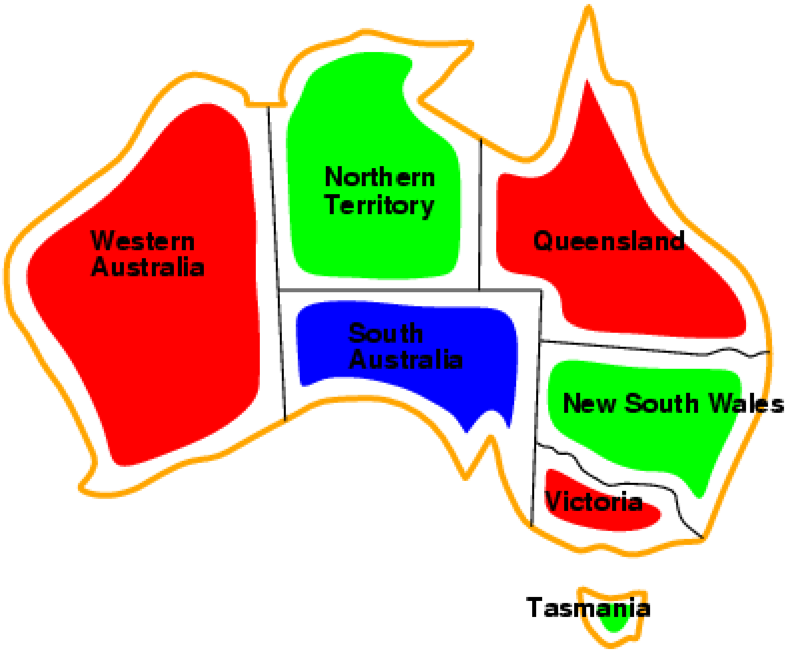
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms
- CSP:

Example: Map-Coloring



- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains D_i = {red,green,blue}
- Constraints: adjacent regions must have different colors
- e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue), (green,red), (green,blue),(blue,red),(blue,green)}

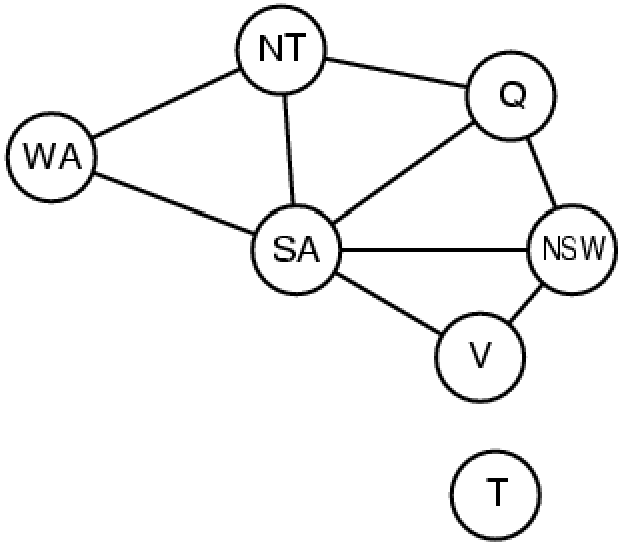
Example: Map-Coloring



- Solutions are complete and consistent assignments
- e.g., WA = red, NT = green, Q = red, NSW = green,V = red,SA = blue,T = green
-

Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints



Varieties of CSPs

- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* → $O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \le StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by LP

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

Backtracking search

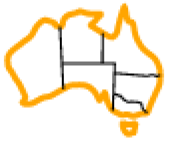
- Variable assignments are commutative, i.e.,
  [WA = red then NT = green] same as [NT = green then WA = red]

- => Only need to consider assignments to a single variable at each node

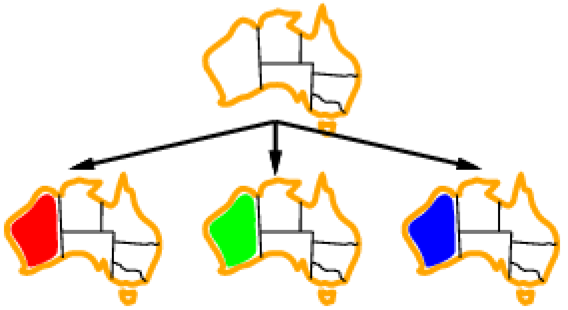
- Depth-first search for CSPs with single-variable assignments is called backtracking search

- Can solve *n*-queens for $n \approx 25$

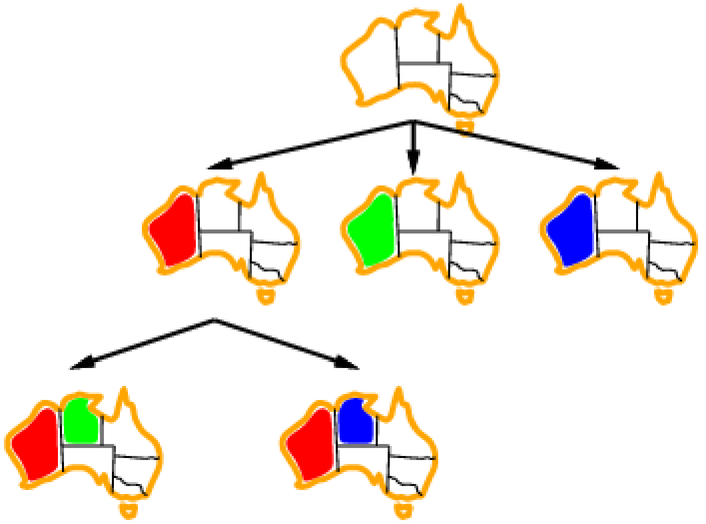
Backtracking example



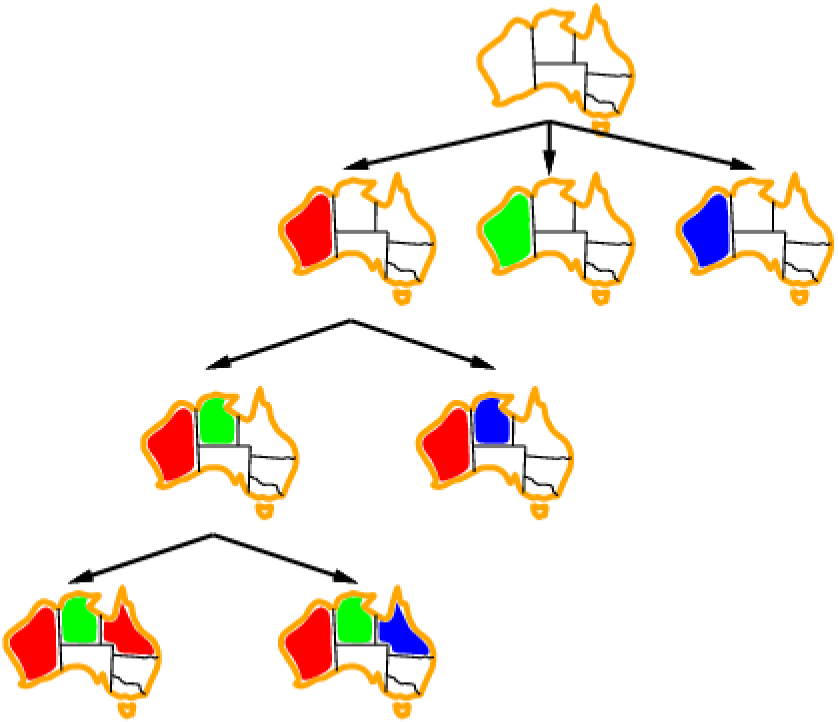
Backtracking example



Backtracking example



Backtracking example



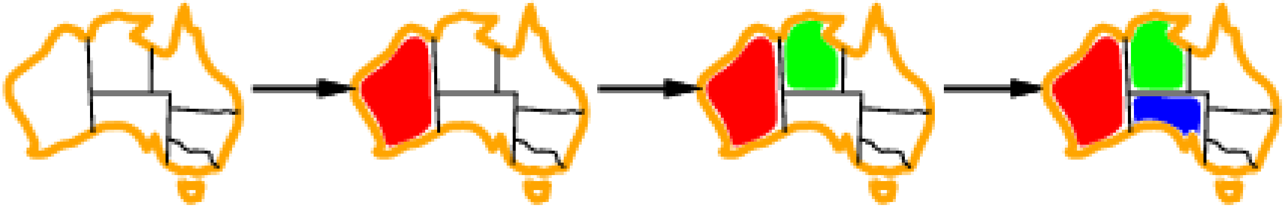
Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Most constrained variable

- Most constrained variable:

  choose the variable with the fewest legal values

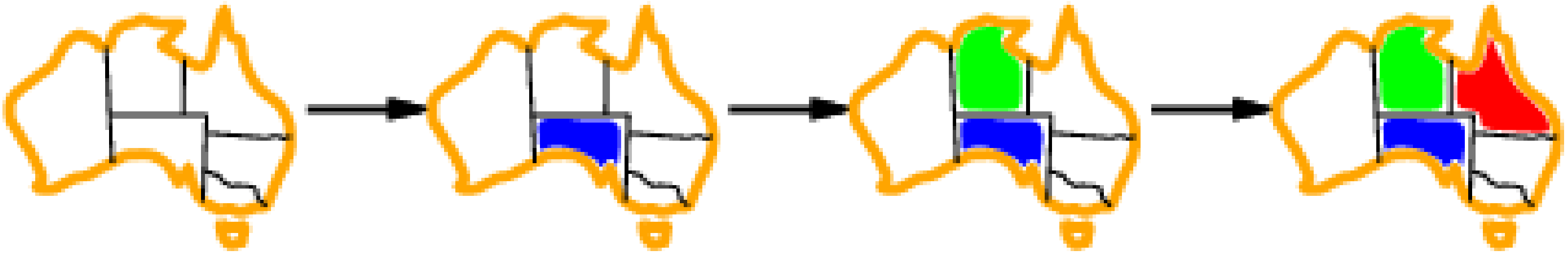


- a.k.a. minimum remaining values (MRV) heuristic

  –

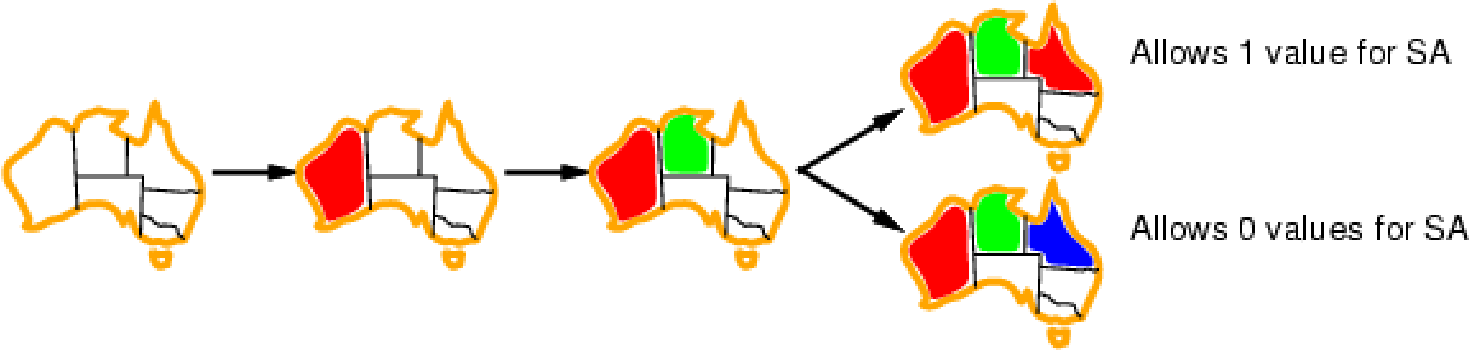
Most constraining variable

- A good idea is to use it as a tie-breaker among most constrained variables
- Most constraining variable:
  – choose the variable with the most constraints on remaining variables



Least constraining value

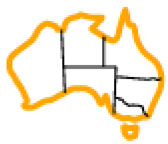
- Given a variable to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

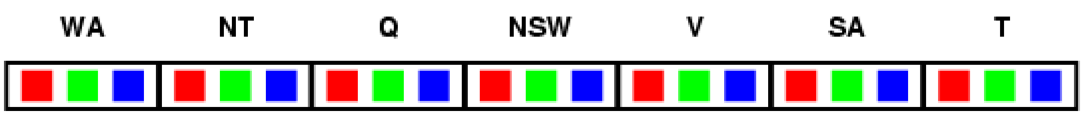


- Combining these heuristics makes 1000 queens feasible

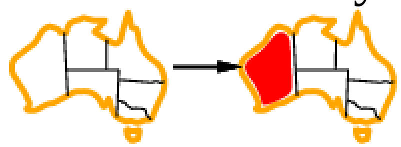
Forward checking

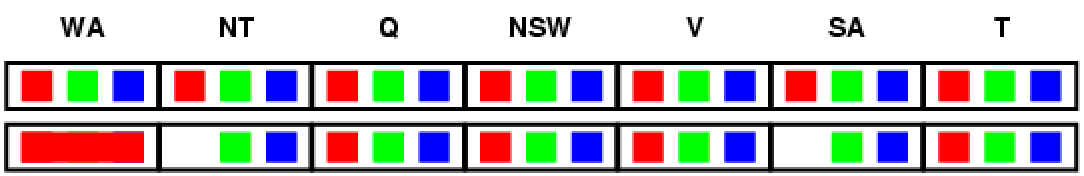
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -





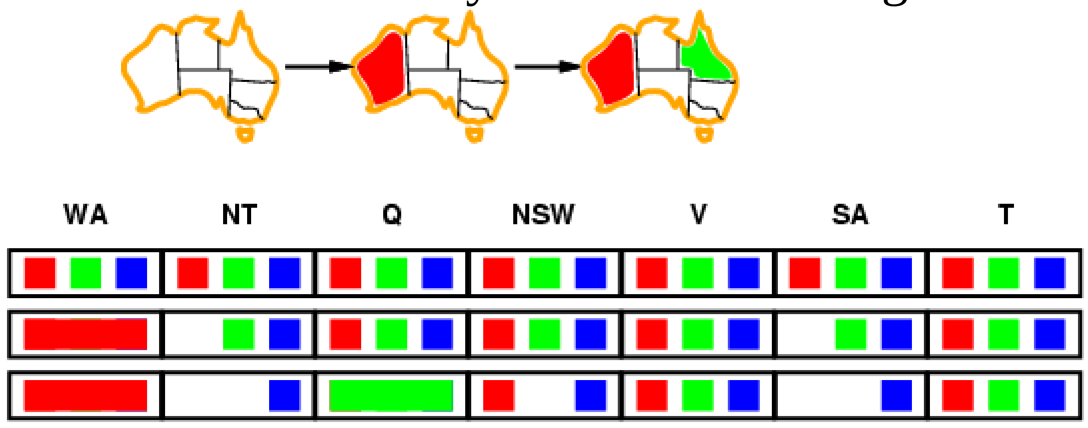
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  - 



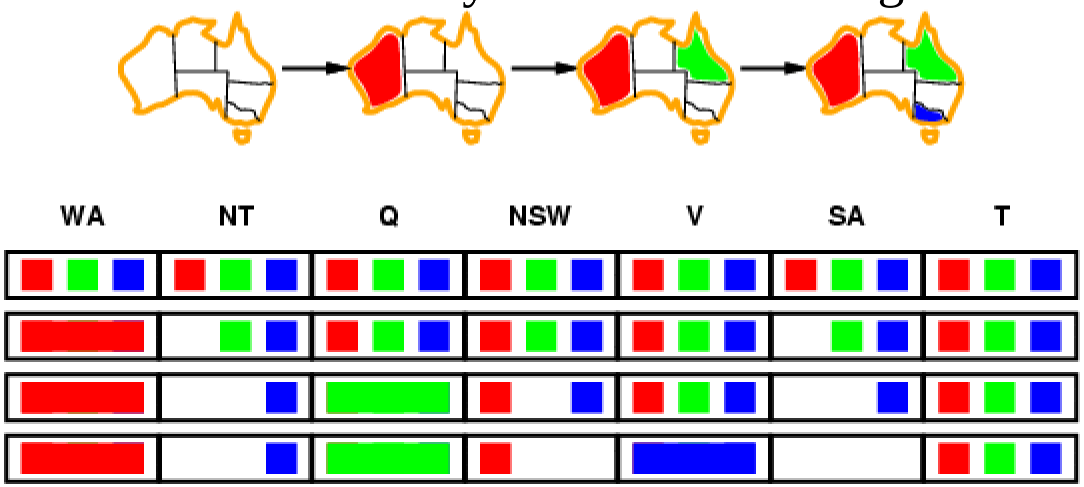
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -



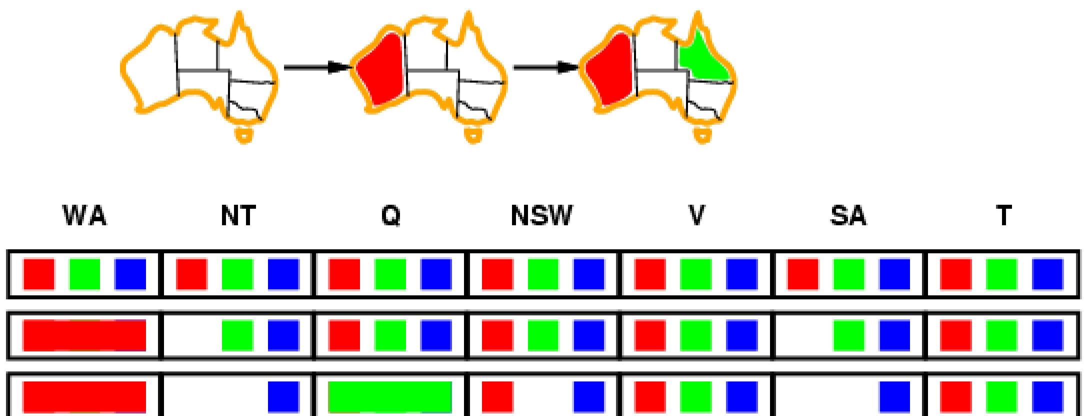
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -



Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Constraint propagation algorithms repeatedly enforce constraints locally…

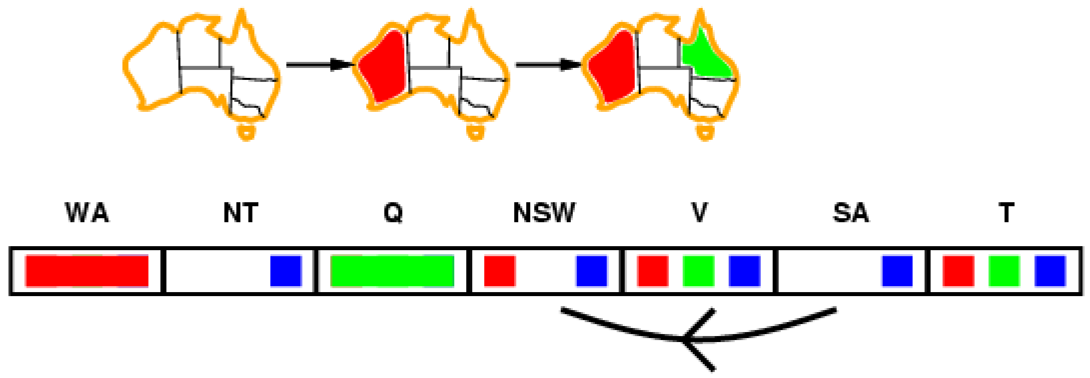
Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

  –

-

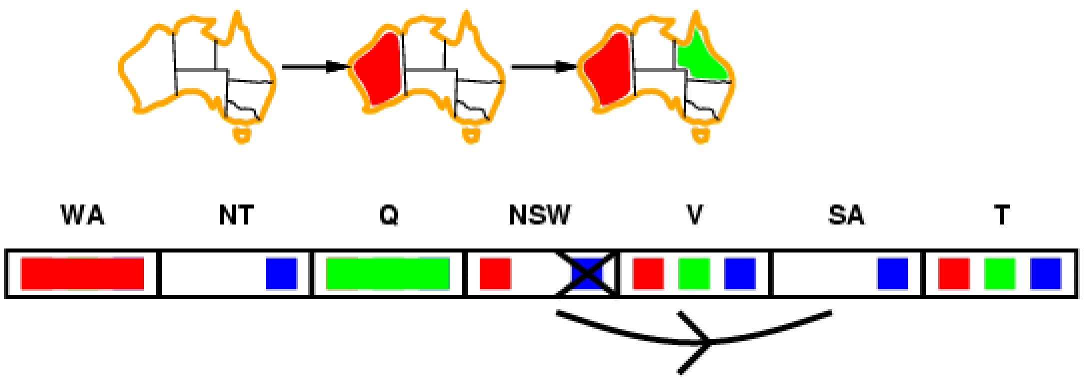


Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

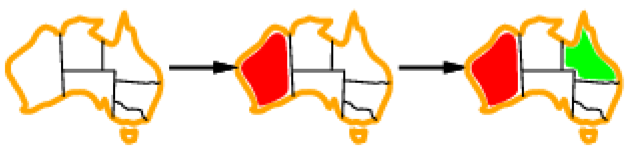
  –

- 

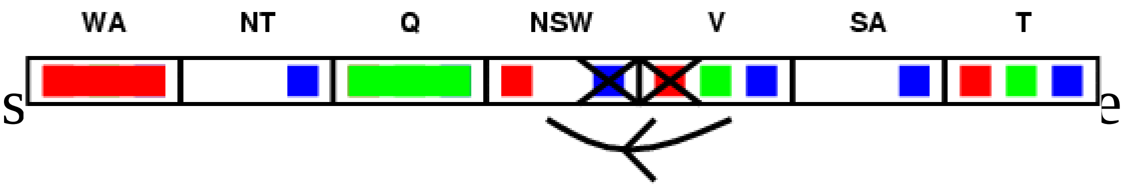
Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y



- If X los



ecked

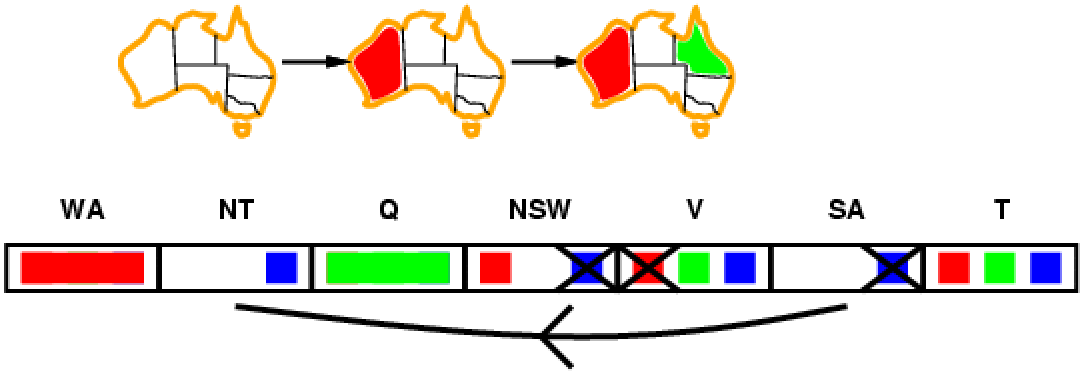
  –

-

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency algorithm AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function RM-INCONSISTENT-VALUES( Xi, Xj) returns true iff remove a value
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
         then delete x from DOMAIN[Xi];   removed ← true
   return removed
```

- Time complexity: O(#constraints |domain|^3)

Checking consistency of an arc is O(|domain|^2)

k-consistency

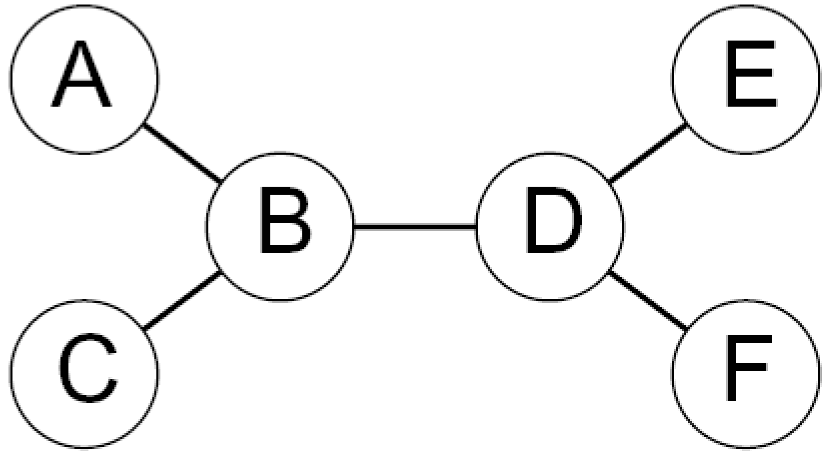
- A CSP is *k-consistent* if, for any set of k-1 variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable
- 1-consistency is node consistency
- 2-consistency is arc consistency
- For binary constraint networks, 3-consistency is the same as *path consistency*
- Getting k-consistency requires time and space exponential in k
- *Strong k-consistency* means k'-consistency for all k' from 1 to k
  - Once strong k-consistency for k=#variables has been obtained, solution can be constructed trivially
- Tradeoff between propagation and branching
- Practitioners usually use 2-consistency and less commonly 3-consistency

Other techniques for CSPs

- Global constraints
  - E.g., Alldiff
  - E.g., Atmost(10,P1,P2,P3), i.e., sum of the 3 vars ≤ 10
  - Special propagation algorithms
    - Bounds propagation
      - E.g., number of people on two flight D1 = [0, 165] and D2 = [0, 385]
      - Constraint that the total number of people has to be at least 420
      - Propagating bounds constraints yields D1 = [35, 165] and D2 = [255, 385]
    - …
- Symmetry breaking

Structured CSPs

Tree-structured CSPs



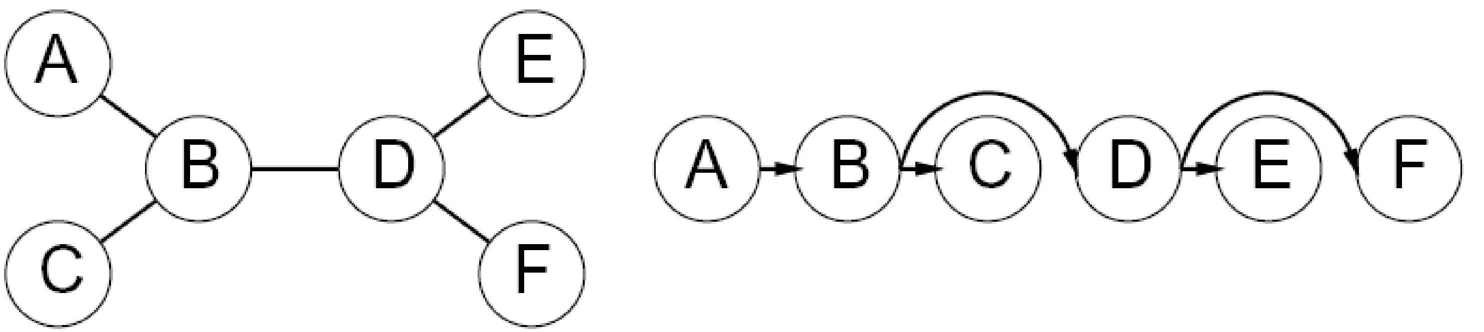
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions
and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

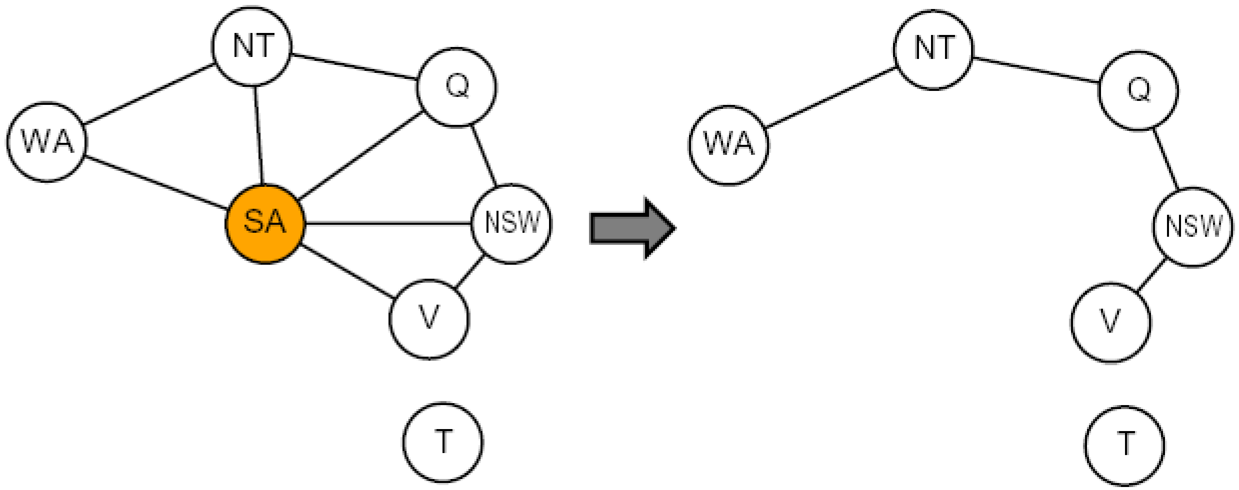


2. For j from n down to 2, apply $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Nearly tree-structured CSPs

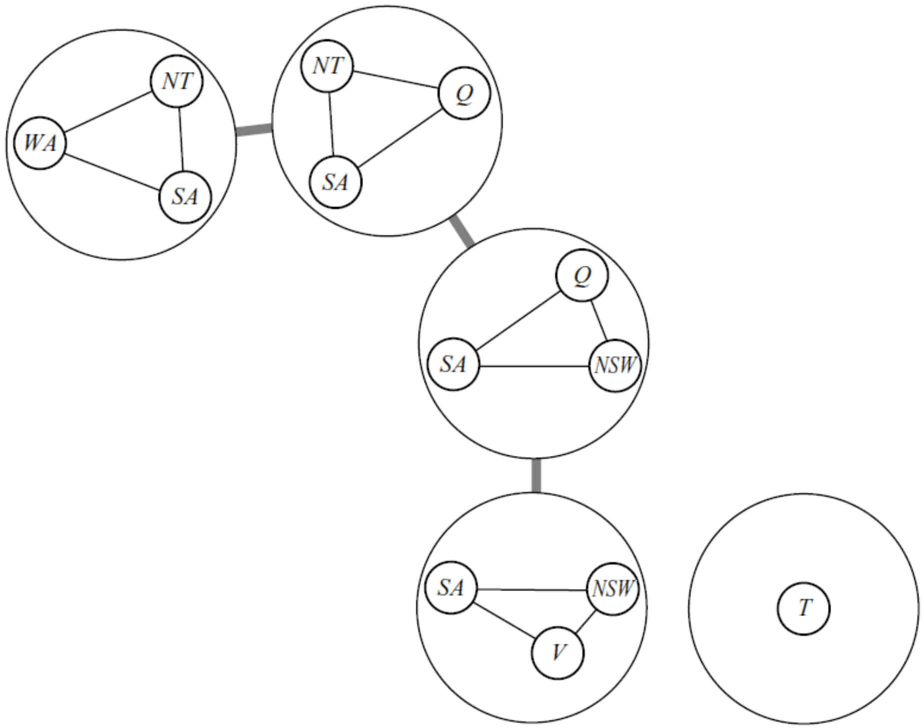
Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree (Finding the minimum cutset is NP-complete.)

Cutset size c $\Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Tree decomposition



- Every variable in original problem must appear in at least one subproblem
- If two variables are connected in the original problem, they must appear together (along with the constraint) in at least one subproblem
- If a variable occurs in two subproblems in the tree, it must appear in every subproblem on the path that connects the two

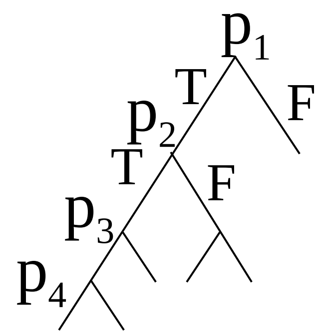
- Algorithm: solve for all solutions of each subproblem. Then, use the tree-structured algorithm, treating the subproblem solutions as variables for those subproblems.
- $O(nd^{w+1})$ where w is the *treewidth* (= one less than size of largest subproblem)
- Finding a tree decomposition of smallest treewidth is NP-complete, but good heuristic methods exists

An example CSP application: satisfiability

Davis-Putnam-Logemann-Loveland (DPLL) tree search algorithm

E.g. for 3SAT

$\exists ?\ \bar{p}$ s.t. $(p_1 \vee \neg p_3 \vee p_4) \wedge \overbrace{(\neg p_1 \vee p_2 \vee \neg p_3)}^{\text{clause}} \wedge \ldots$



Complete

Backtrack when some clause becomes empty

Unit propagation (for variable & value ordering): if some clause only has one literal left, assign that variable the value that satisfies the clause (never need to check the other branch)

Boolean Constraint Propagation (BCP): Iteratively apply unit propagation until there is no unit clause available

A helpful observation for the DPLL procedure

$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$ (Horn)

is equivalent to

$\neg(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \vee Q$ (Horn)

is equivalent to

$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$ (Horn clause)

Thrm. If a propositional theory consists only of Horn clauses (i.e., clauses that have at most one non-negated variable) and unit propagation does not result in an explicit contradiction (i.e., Pi and ¬Pi for some Pi), then the theory is satisfiable.

Proof. On the next page.

…so, Davis-Putnam algorithm does not need to branch on variables which only occur in Horn clauses

Proof of the thrm

Assume the theory is Horn, and that unit propagation has completed (without contradiction). We can remove all the clauses that were satisfied by the assignments that unit propagation made. From the unsatisfied clauses, we remove the variables that were assigned values by unit propagation. The remaining theory has the following two types of clauses that contain unassigned variables only:

$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$ and

$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n$

Each remaining clause has at least two variables (otherwise unit propagation would have applied to the clause). Therefore, each remaining clause has at least one negated variable. Therefore, we can satisfy all remaining clauses by assigning each remaining variable to *False*.

Variable ordering heuristic for DPLL [Crawford & Auton AAAI-93]

Heuristic: Pick a non-negated variable that occurs in a non-Horn (more than 1 non-negated variable) clause with a minimal number of non-negated variables.

Motivation: This is effectively a "most constrained first" heuristic if we view each non-Horn clause as a "variable" that has to be satisfied by setting one of its non-negated variables to *True*. In that view, the branching factor is the number of non-negated variables the clause contains.

Q: Why is branching constrained to non-negated variables?

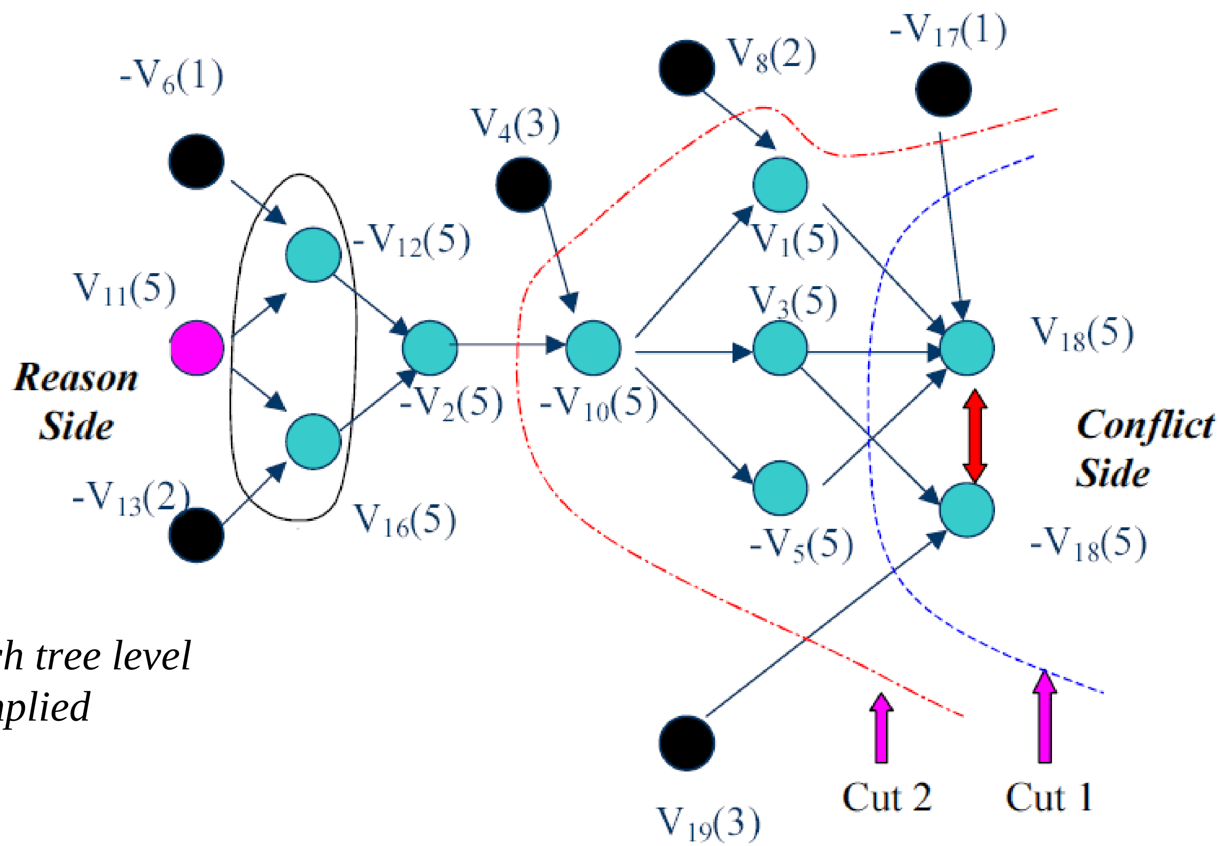
A: We can ignore any negated variables in the non-Horn clauses because

- whenever any one of the non-negated variables is set to *True* the clause becomes redundant (satisfied), and
- whenever all but one of the non-negated variables is set to *False* the clause becomes Horn.

Variable ordering heuristics can make several orders of magnitude difference in speed.

Constraint learning aka nogood learning aka clause learning

used by state-of-the-art SAT solvers (and CSP more generally)



Conflict graph

- *Nodes are literals*
- *Number in parens shows the search tree level where that node got decided or implied*

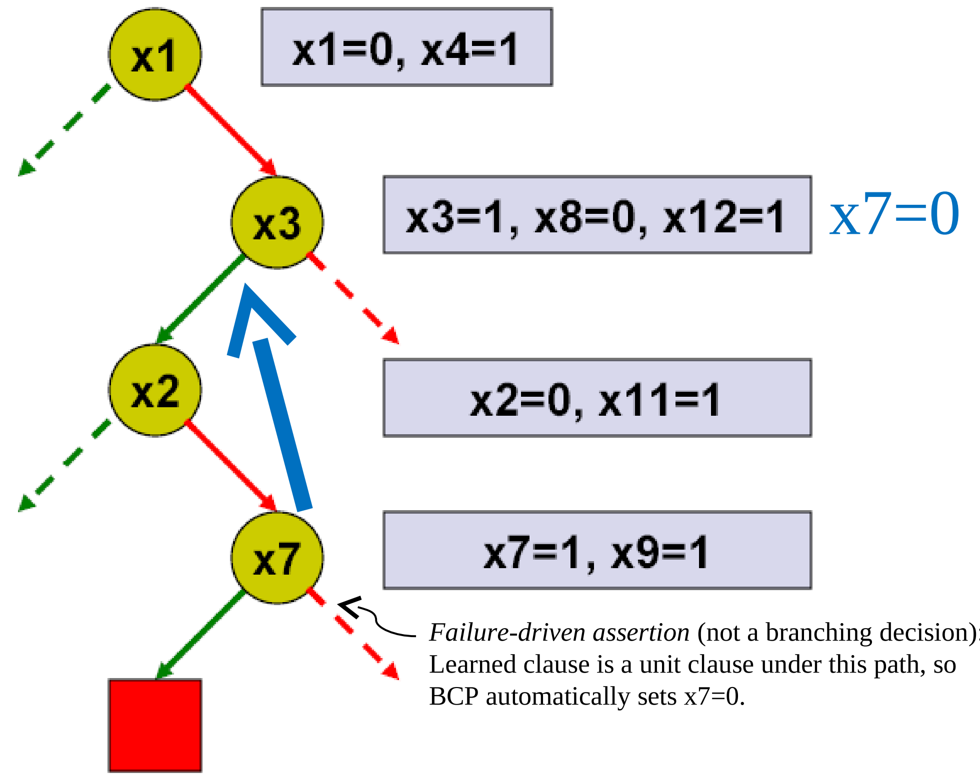
- Cut 2 gives the first-unique-implication-point (i.e., 1 UIP on reason side) constraint (v2 or –v4 or –v8 or v17 or -v19). That schemes performs well in practice.

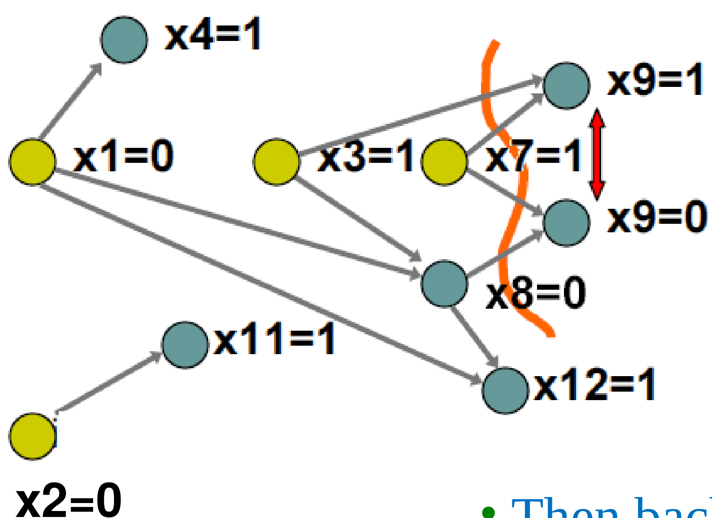
  Any cut would give a valid clause. Which cuts should we use? Should we delete some?
- The learned clauses apply to all other parts of the tree as well.

Conflict-directed backjumping

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

x3’+x7’+x8





x3=1∧x7=1∧x8=0 → conflict

Add conflict clause: x3’+x7’+x8

- Then backjump to the decision level of x3=1,
  keeping x3=1 (for now), and
  forcing the implied fact x7=0 for that x3=1 branch
- WHAT’S THE POINT? A: No need to just backtrack to x2

Classic readings on conflict-directed backjumping, clause learning, and heuristics for SAT

- "GRASP: A Search Algorithm for Propositional Satisfiability", Marques-Silva & Sakallah, *IEEE Trans. Computers, C-48, 5:506-521,*1999. (Conference version 1996.)
- ("Using CSP look-back techniques to solve real world SAT instances", Bayardo & Schrag*, Proc. AAAI, pp. 203-208, 1997)*
- "Chaff: Engineering an Efficient SAT Solver", Moskewicz, Madigan, Zhao, Zhang & Malik, 2001 (www.princeton.edu/~chaff/publication/DAC2001v56.pdf)
- "BerkMin: A Fast and Robust Sat-Solver", Goldberg & Novikov, *Proc. DATE 2002, pp. 142-149*
- See also slides at http://www.princeton.edu/~sharad/CMUSATSeminar.pdf

More on conflict-directed backjumping (CBJ)

- These are for general CSPs, not SAT specifically:
- Read Section 6.3.3. of Russell & Norvig for an easy description of conflict-directed backjumping for general CSP
- "Conflict-directed backjumping revisited" by Chen and van Beek, *Journal of AI Research*, 14, 53-81, 2001:
  - As the level of local consistency checking (lookahead) is increased, CBJ becomes less helpful
    - A dynamic variable ordering exists that makes CBJ redundant
  - Nevertheless, adding CBJ to backtracking search that maintains generalized arc consistency leads to orders of magnitude speed improvement experimentally
- "Generalized NoGoods in CSPs" by Katsirelos & Bacchus, *National Conference on Artificial Intelligence (AAAI-2005)* pages 390-396, 2005.
  - This paper generalizes the notion of nogoods, and shows that nogood learning (then) can speed up (even non-SAT) CSPs significantly

Random restarts

- Sometimes it makes sense to keep restarting the CSP/SAT algorithm, using randomization in variable ordering
  - Avoids the very long run times of unlucky variable ordering
  - On many problems, yields faster algorithms
  - Clauses learned can be carried over across restarts
  - Experiments suggest it does not help on optimization problems (e.g., [Sandholm *et al.* IJCAI-01, Management Science 2006])

Phase transitions in CSPs

"Order parameter" for 3SAT
[Mitchell, Selman, Levesque AAAI-92]

- β = #clauses / # variables
- This predicts
  - satisfiability
  - hardness of finding a model

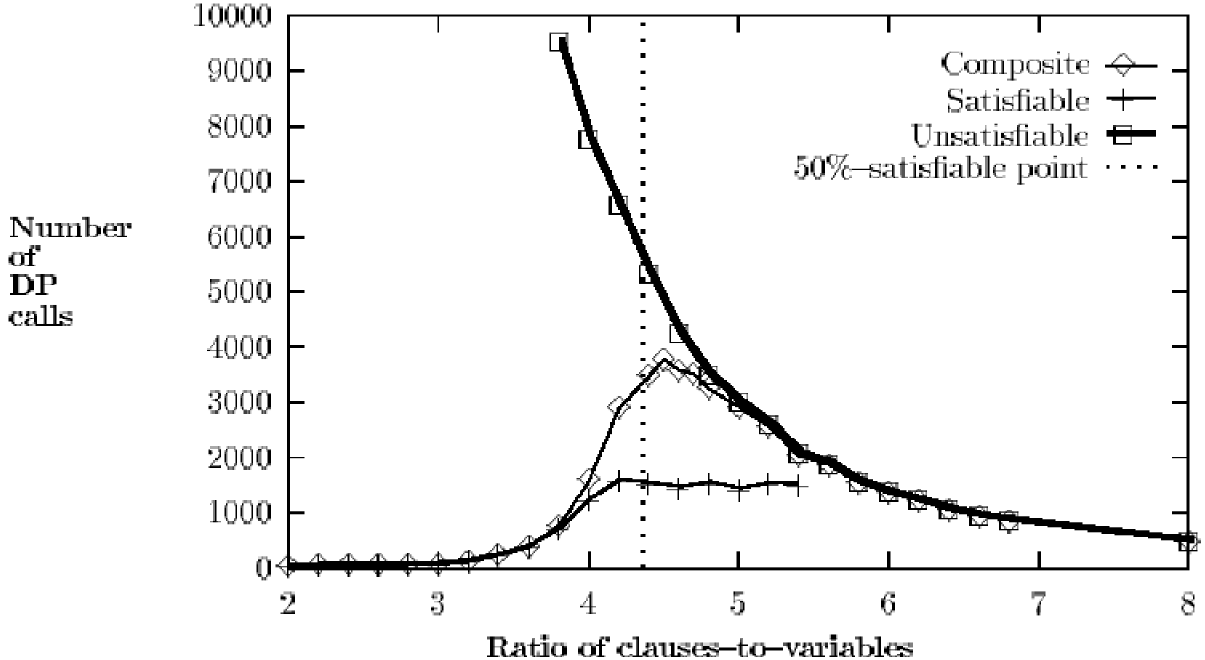
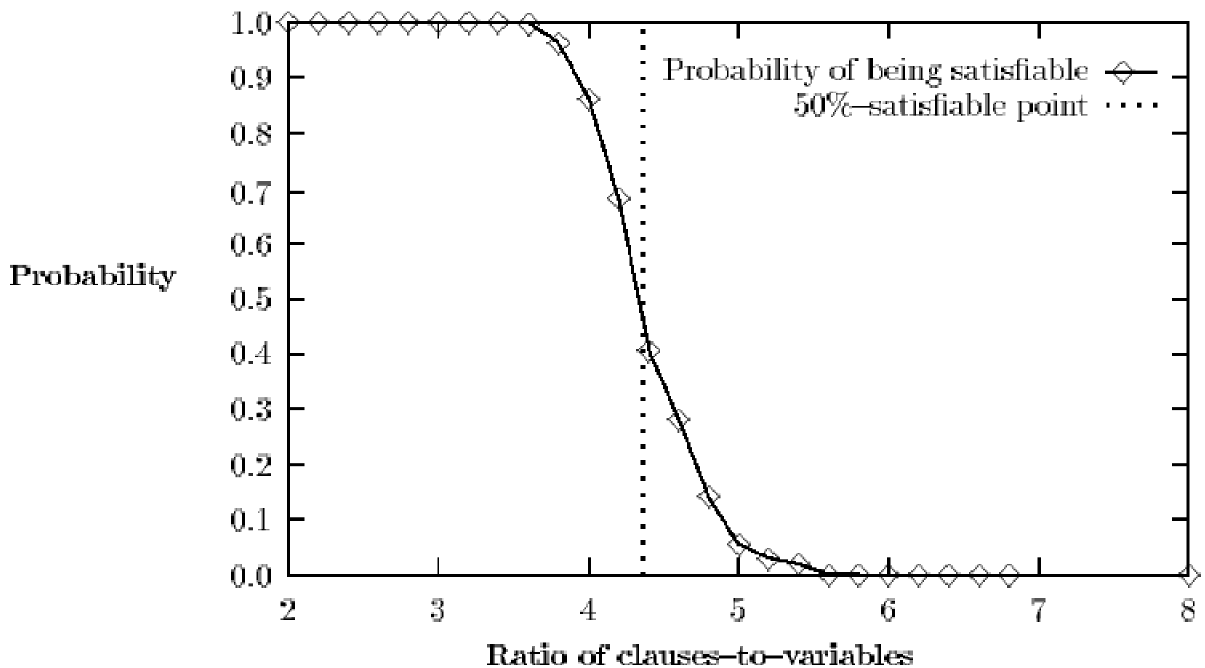


Figure 3: Median DP calls for 50–variable Random 3-SAT as a function of the ratio of clauses–to–variables.

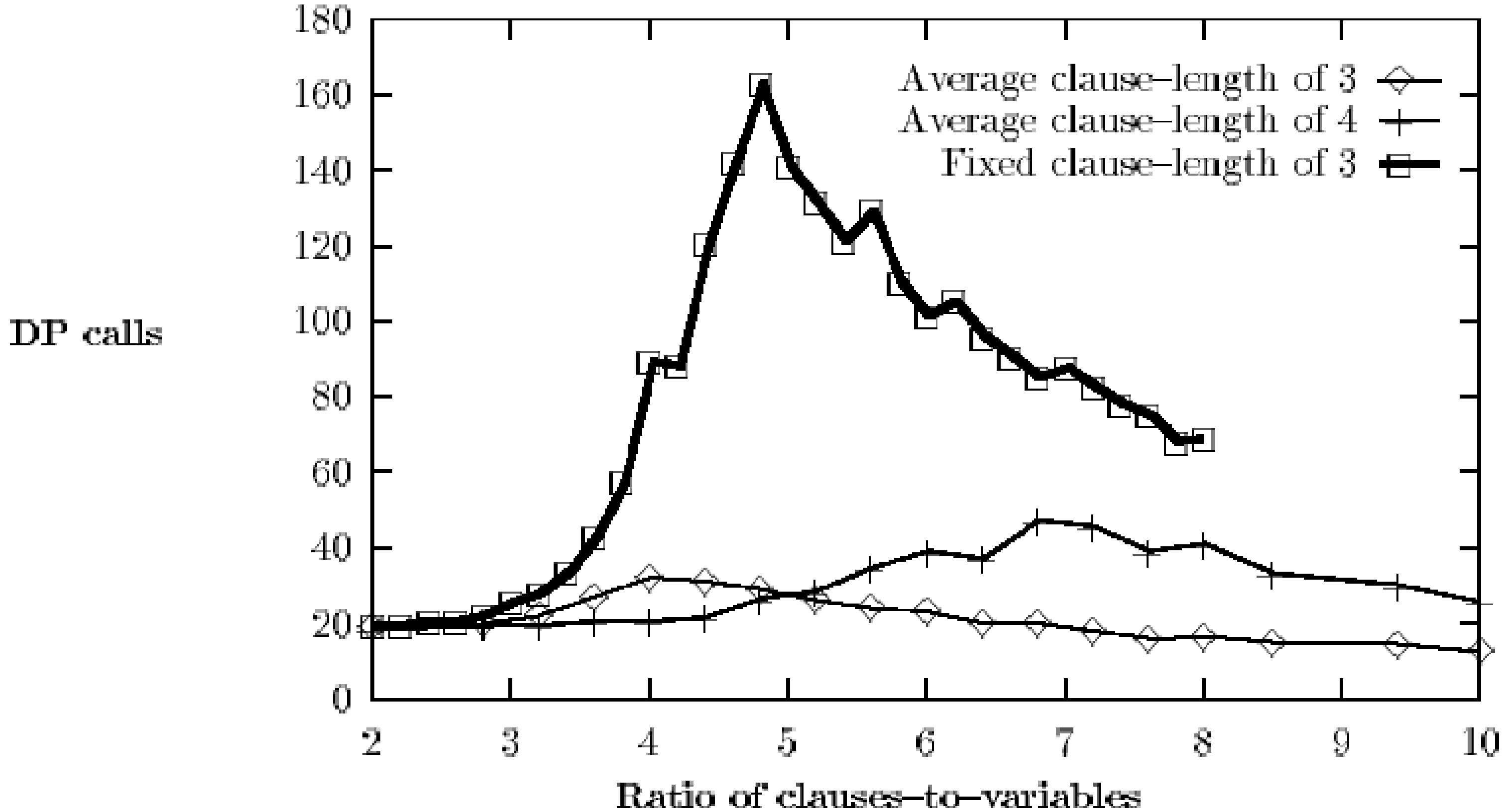


How would you capitalize on the phase transition in an algorithm?

Generality of the order parameter β

- The results seem quite general across model finding algorithms
- Other constraint satisfaction problems have order parameters as well

…but the complexity peak does not occur (at least not in the same place) under all ways of generating SAT instances



Iterative refinement algorithms for SAT

GSAT [Selman, Levesque, Mitchell AAAI-92]

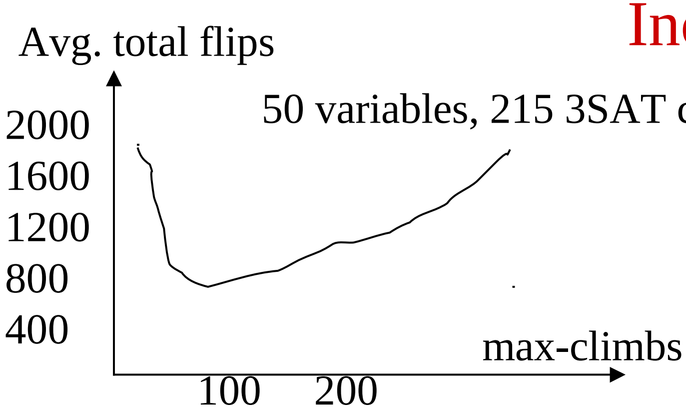
(= a local search algorithm for model finding)

function GSAT(*sentence, max-restarts, max-climbs*) **returns** a truth assignment or failure

  for $i \leftarrow 1$ **to** *max-restarts* **do**
    $A \leftarrow$ A randomly generated truth assignment
    for $j \leftarrow 1$ **to** *max-climbs* **do**
      if A satisfies *sentence* **then return** A
      A $\leftarrow$ a random choice of one of the best successors of A
    end
  end
  return failure

Figure 6.17 The GSAT algorithm for satisfiability testing. The successors of an assignment A are truth assignment with one symbol flipped. A "best assignment" is one that makes the most clauses true.

Incomplete (unless restart a lot)



Greediness is not essential as long as climbs and sideways moves are preferred over downward moves.

BREAKOUT algorithm [Morris AAAI-93]

Initialize all variables Pi randomly
UNTIL current state is a solution
    IF current state is not a local minimum
    THEN make any local change that reduces the total cost
    (i.e. flip one Pi)
    ELSE increase weights of all unsatisfied clause by one

Incomplete, but very efficient on large (easy) satisfiable problems.

Reason for incompleteness: the cost increase of the current local optimum spills over to other solutions because they share unsatisfied clauses.