CMSC 671
Fall 2010

Thu 9/16/10

Constraints Processing / Constraint Satisfaction Problem

Prof. Laura Zavala, laura.zavala@umbc.edu, ITE 373, 410-455-8775

Algorithms for CSPs

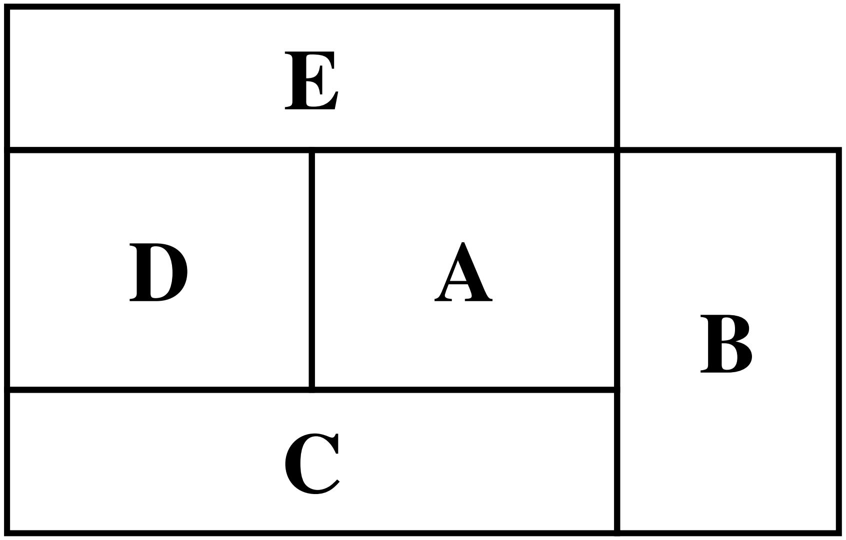
- Backtracking (systematic search)
- Constraint propagation (k-consistency)
- Variable and value ordering heuristics
- Intelligent backtracking

Constraint satisfaction - Overview

- Powerful problem-solving paradigm
  - View a problem as a **set of variables** to which we have to assign **values** that satisfy a number of **problem-specific constraints**.
  - Constraint programming, constraint satisfaction problems (CSPs), constraint logic programming…

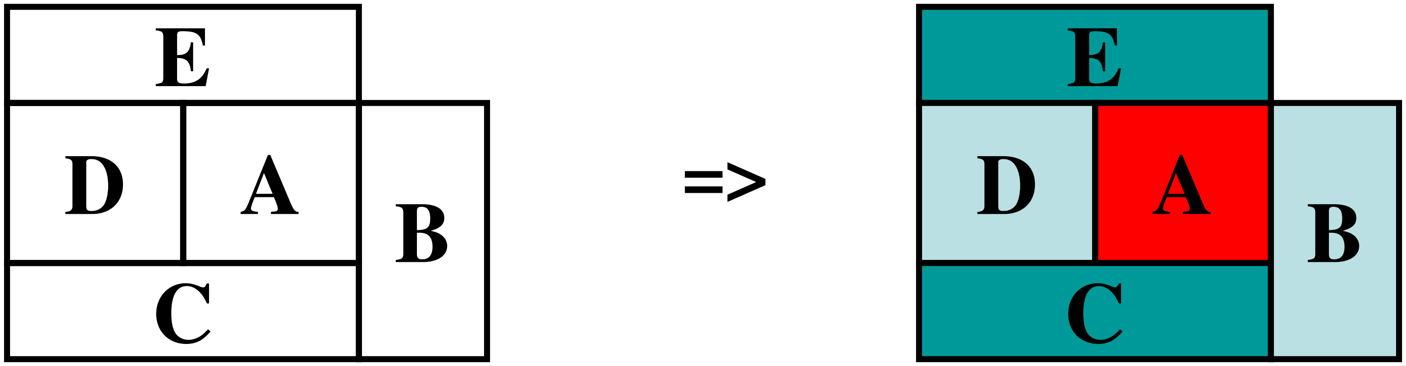
Informal example: Map coloring

- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.

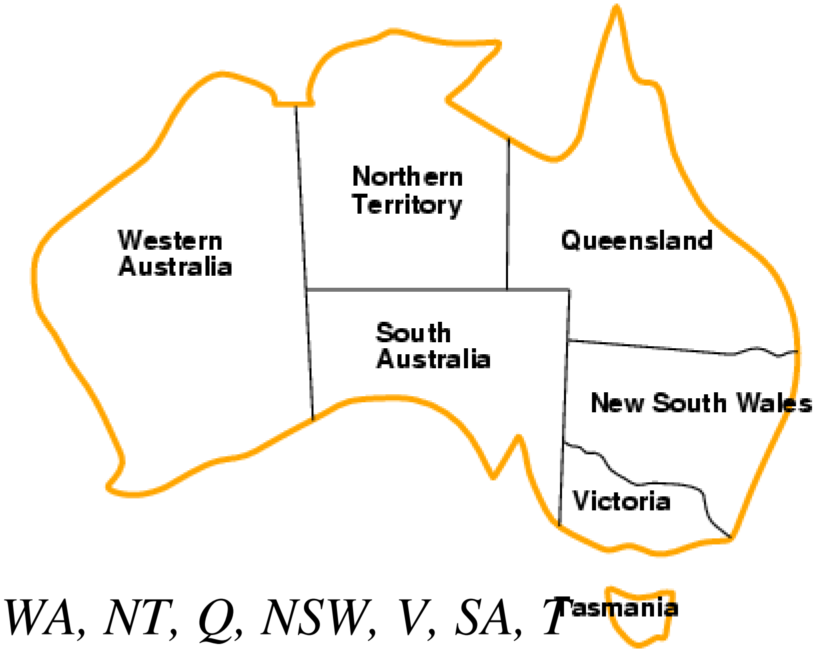


Map coloring II

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: A≠B, A≠C,A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
- One solution: A=red, B=green, C=blue, D=green, E=blue

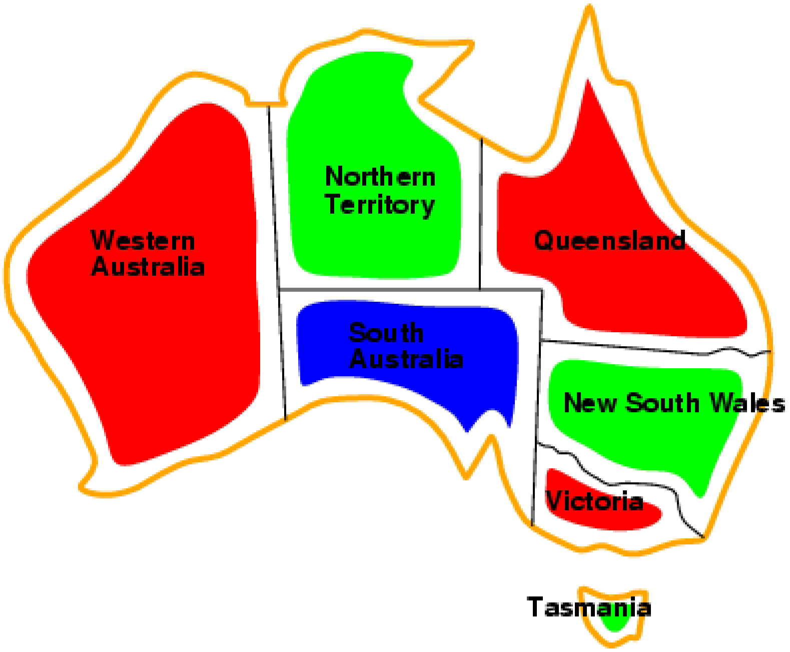


Map-Coloring - Australia



- 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)}

Map-Coloring - Australia



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

Why formulate (problems) using CSP?

- CSPs yield a natural representation for a wide variety of problems
- Easier to use an existing CSP-solving system than designing custom solution using another search technique

Informal definition of CSP

- CSP = Constraint Satisfaction Problem
- Given

  (1) a finite set of variables

  (2) each with a domain of possible values (often finite)

  (3) a set of constraints that limit the values the variables can take on
- A **solution** is an assignment of a value to each variable such that the constraints are all satisfied.
- Tasks might be to decide if a solution exists, to find a solution, to find all solutions, or to find the "best solution" according to some metric (objective function).

Example: SATisfiability

- Given a set of propositions containing variables, find an assignment of the variables to {false,true} that satisfies them.
- For example, the clauses:
  - $(A \vee B \vee \neg C) \wedge (\neg A \vee D)$
  - (equivalent to $(C \rightarrow A) \vee (B \wedge D \rightarrow A)$

  are satisfied by

  A = false

  B = true

  C = false

  D = false

Formal definition of a constraint network (CN)

A constraint network (CN) consists of

- a set of variables $X = \{x_1, x_2, \ldots x_n\}$
  - each with an associated domain of values $\{d_1, d_2, \ldots d_n\}$.
  - the domains are typically finite
- a set of constraints $\{c_1, c_2 \ldots c_m\}$ where
  - each constraint defines a predicate which is a relation over a particular subset of X.
  - e.g., C_i involves variables $\{X_{i1}, X_{i2}, \ldots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots D_{ik}$
- **Unary** constraint: only involves one variable
- **Binary** constraint: only involves two variables

Formal definition of a CN (cont.)

- Instantiations
  - An **instantiation** of a subset of variables S is an assignment of a value in its domain to each variable in S
  - An instantiation is **legal** iff it does not violate any constraints.
- A **solution** is an instantiation of all of the variables in the network.

Real-world problems

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design

Typical tasks for CSP

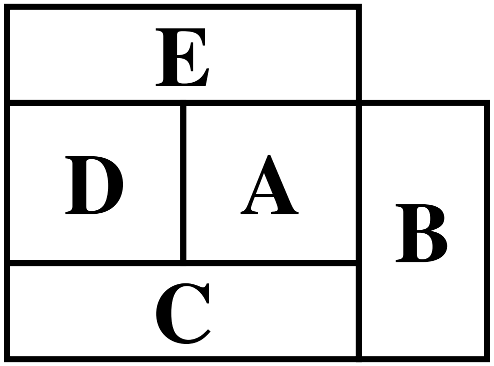
- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CN into an equivalent CN that is easier to solve.

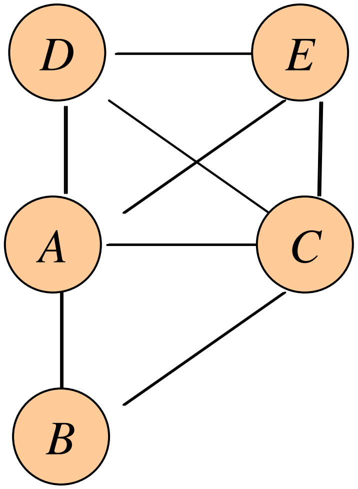
Binary CSP

- A **binary CSP** is a CSP in which all of the constraints are binary or unary.
- Any non-binary CSP can be converted into a binary CSP by introducing additional variables.

Binary CSP

- A binary CSP can be represented as a **constraint graph**, which has a node for each variable and an arc between two nodes if and only there is a constraint involving the two variables.





Example: Sudoku

	3		1
	1		4
3	4	1	2
		4	

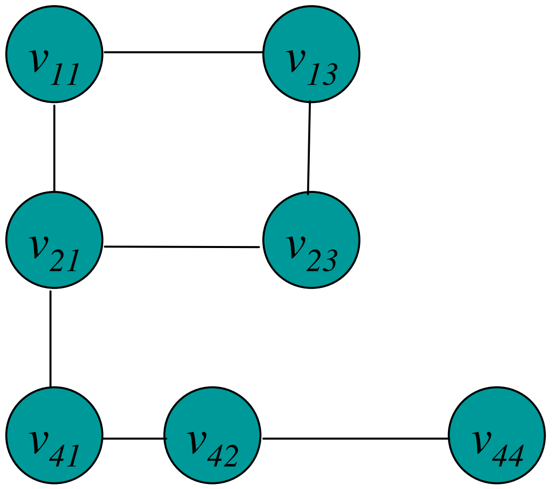
Running example: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

- Variables and their domains
  - v_{ij} is the value in the jth cell of the ith row
  - $D_{ij} = D = \{1, 2, 3, 4\}$
- Blocks:
  - $B_1 = \{11, 12, 21, 22\} \ldots B_4 = \{33, 34, 43, 44\}$
- Constraints (implicit/intensional)
  - $C^R : \forall i,\ \cup_j v_{ij} = D$ (every value appears in every row)
  - $C^C : \forall j,\ \cup_j v_{ij} = D$ (every value appears in every column)
  - $C^B : \forall k,\ \cup (v_{ij} \mid ij \in B_k) = D$ (every value appears in every block)
  - Alternative representation: pairwise inequality constraints:
    - $I^R : \forall i, j \neq j' : v_{ij} \neq v_{ij'}$ (no value appears twice in any row)
    - $I^C : \forall j, i \neq i' : v_{ij} \neq v_{i'j}$ (no value appears twice in any column)
    - $I^B : \forall k, ij \in B_k, i'j' \in B_k, ij \neq i'j' : v_{ij} \neq v_{i'j'}$ (no value appears twice in any block)
  - Advantage of the second representation: all binary constraints!

Sudoku constraint network

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

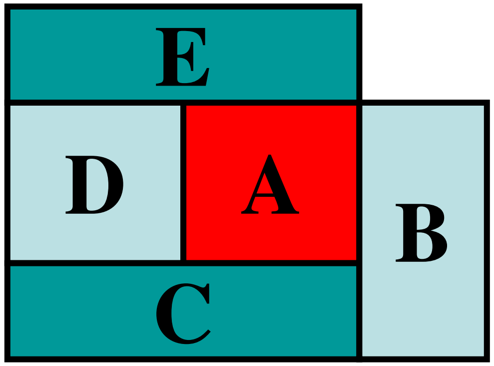


Solving constraint problems

- Systematic search
    - Generate and test
    - Backtracking
- Variable ordering heuristics
- Value ordering heuristics
- Constraint propagation (consistency)
- Backjumping and dependency-directed backtracking

Generate and test

- Try each possible combination until you find one that works:
    - green – red – green – red – green
    - green – red – green – red – blue
    - green – red – green – red – red
    - …
- Doesn't check constraints until all variables have been instantiated
- Very inefficient way to explore the space of possibilities

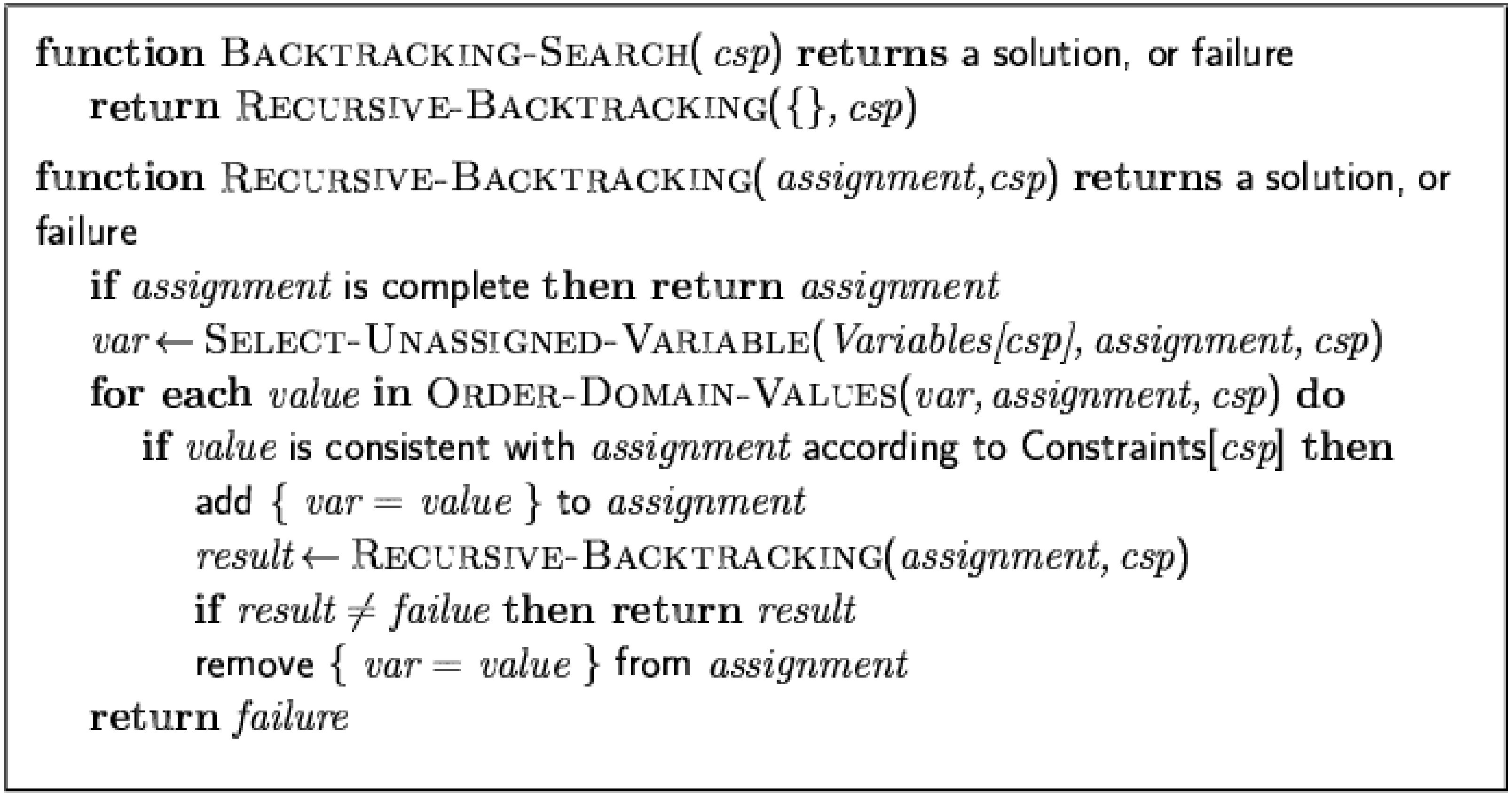


Backtracking

(a.k.a. depth-first search!)

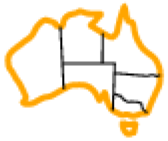
- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we've reached a dead end and need to backtrack to the previous variable
- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values

Backtracking search



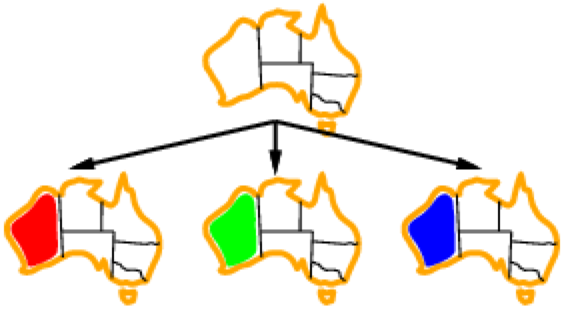
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

Backtracking example

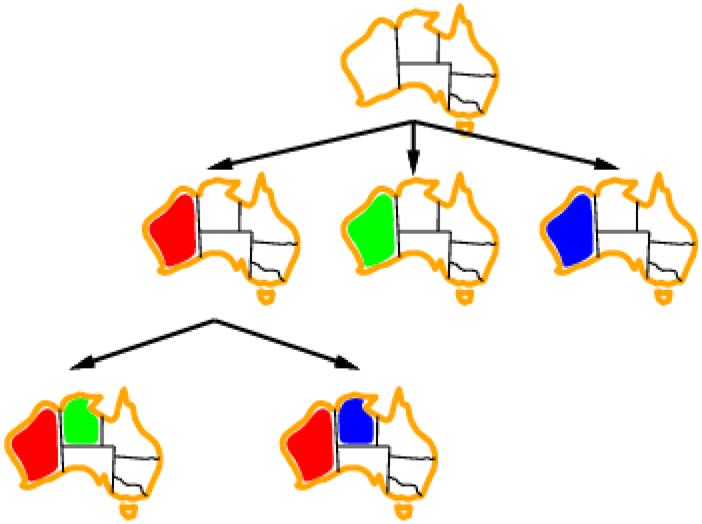


http://aima.eecs.berkeley.edu/slides-ppt/

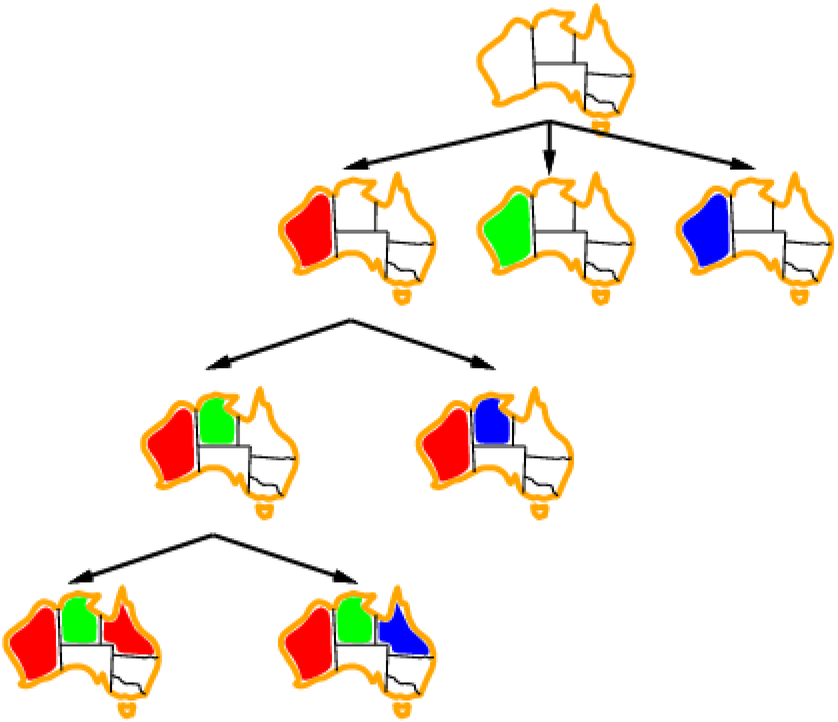
Backtracking example



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?

Problems with backtracking

- Inefficiency: can explore areas of the search space that aren't likely to succeed
  - Variable and value ordering can help
- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help

Variable and value ordering

- Minimum remaining values (variables)
  - fewest legal values
- Degree heuristic (variables)
  - largest number of constraints on other unassigned variables
  - reduces branching factor
- Least constraining value (values)
  - rules out the fewest choices for neighboring vars

Constraint Propagation

- Using the constraints to reduce the number of legal values for a variable, which in turn reduces the number of legal values for another variable, and so on.

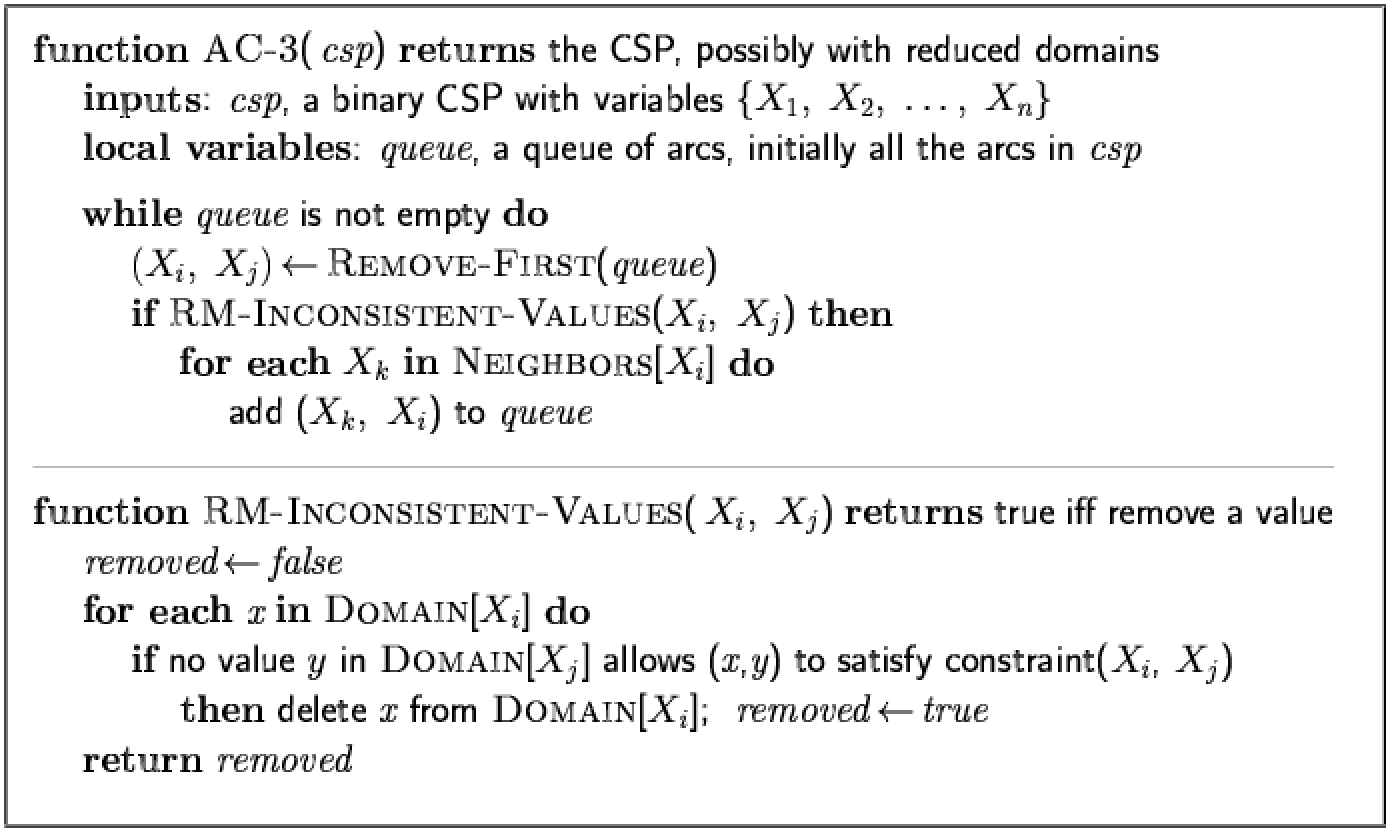
Consistency

- Node consistency
  - A node X is **node-consistent** if every value in the domain of X is consistent with X's unary constraints
  - A graph is node-consistent if all nodes are node-consistent

Consistency

- Arc consistency
  - An arc (X, Y) is **arc-consistent** if, for every value x of X, there is a value y for Y that satisfies the constraint represented by the arc.
  - A graph is arc-consistent if all arcs are arc-consistent.
- To create arc consistency, we perform **constraint propagation**: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs

Arc consistency algorithm AC-3



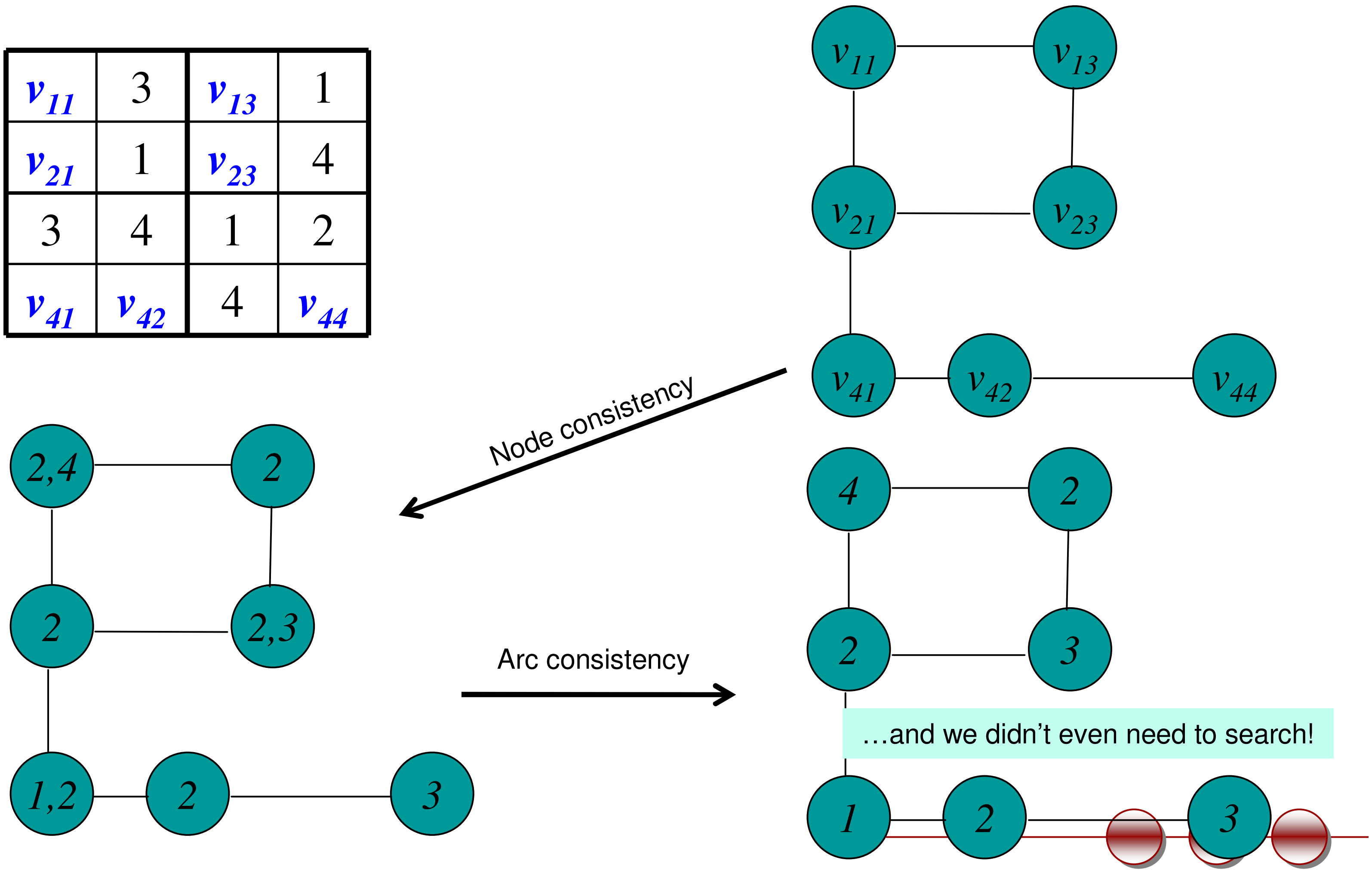
```
function AC-3( $csp$) returns the CSP, possibly with reduced domains
    inputs: $csp$, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
    local variables: $queue$, a queue of arcs, initially all the arcs in $csp$

    while $queue$ is not empty do
        $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
        if RM-INCONSISTENT-VALUES($X_i$, $X_j$) then
            for each $X_k$ in NEIGHBORS[$X_i$] do
                add ($X_k$, $X_i$) to $queue$
```

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

- Time complexity: $O(n^2d^3)$

Constraint propagation: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}



…and we didn't even need to search!

K-consistency

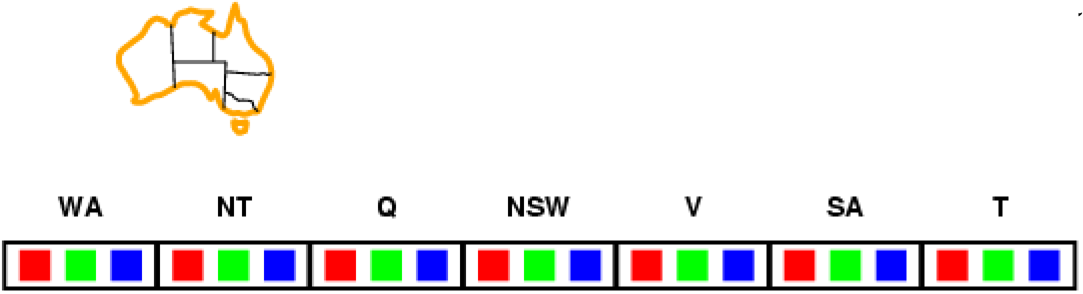
- K- consistency generalizes the notion of arc consistency to sets of more than two variables.
  - A graph is K-consistent if, for legal values of any K-1 variables in the graph, and for any Kth variable V_k, there is a legal value for V_k
- Strong K-consistency = J-consistency for all J<=K
- **Node consistency = strong 1-consistency**
- **Arc consistency = strong 2-consistency**
- Path consistency = strong 3-consistency

Why do we care?

1. If we have a CSP with N variables that is known to be **strongly N-consistent**, we can solve it **without backtracking**
2. For any CSP that is **strongly K-consistent**, if we find an **appropriate variable ordering** (one with "small enough" branching factor), we can solve the CSP **without backtracking**

Forward checking

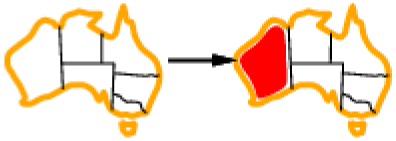
- Idea:
  - Interleaving search and inference of reductions in the domain of the variables
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦

Forward checking

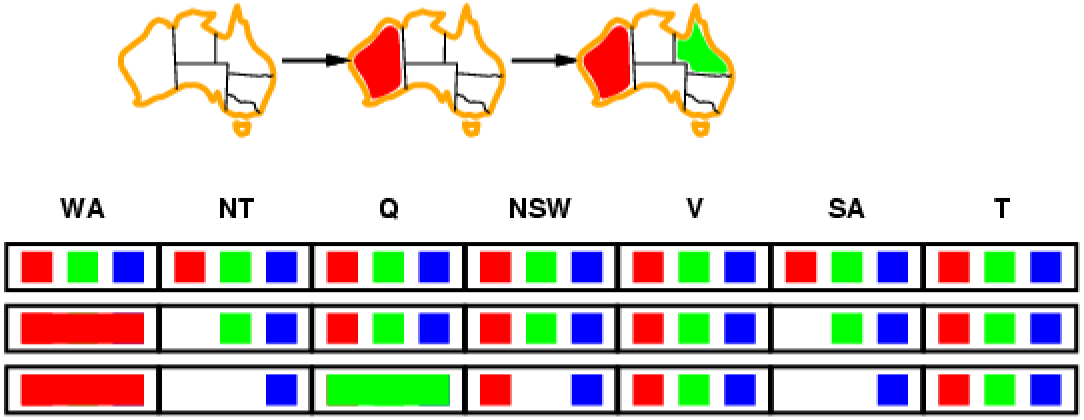
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

Forward checking

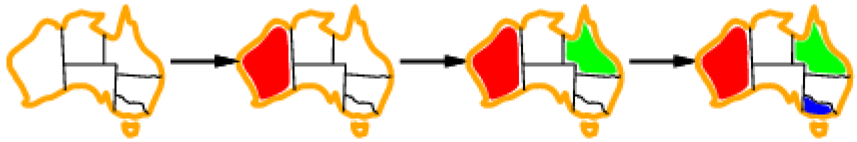
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

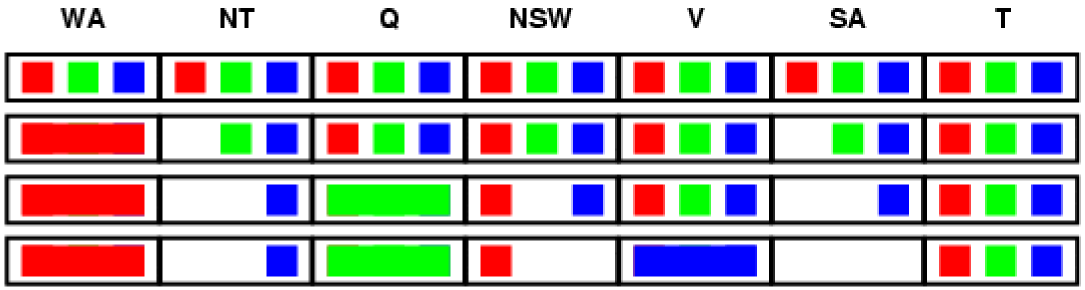


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

Forward checking

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

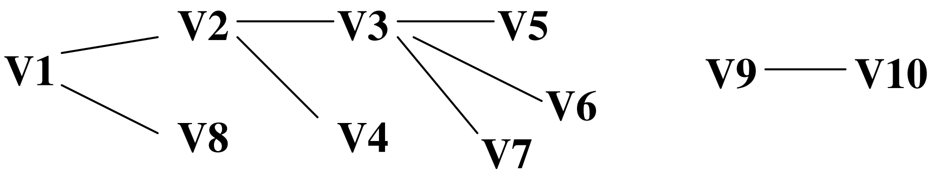




WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Tree-structured constraint graph

- A **constraint tree** rooted at V_1 satisfies the following property:
  - There exists an ordering V1, …, Vn such that every node has zero or one parents (i.e., each node only has constraints with at most one "earlier" node in the ordering)



  - Also known as an *ordered constraint graph with width 1*
- If this constraint tree is also node- and arc-consistent (a.k.a. *strongly 2-consistent*), then it can be solved without backtracking
- (More generally, if the ordered graph is strongly k-consistent, and has width w < k, then it can be solved without backtracking.)

Proof sketch for constraint trees

- Perform backtracking search in the order that satisfies the constraint tree condition
- Every node, when instantiated, is constrained only by at most one previous node
- Arc consistency tells us that there must be at least one legal instantiation in this case
  - (If there are no legal solutions, the arc consistency procedure will collapse the graph – some node will have no legal instantiations)
- Keep doing this for all *n* nodes, and you have a legal solution – without backtracking!

Backtrack-free CSPs: Proof sketch

- Given a strongly k-consistent OCG, G, with width w < k:
  - Instantiate variables in order, choosing values that are consistent with the constraints between V_i and its parents
  - Each variable has at most w parents, and k-consistency tells us we can find a legal value consistent with the values of those w parents
- *Unfortunately*, achieving k-consistency is hard (and can increase the width of the graph in the process!)
- *Fortunately*, 2-consistency is relatively easy to achieve, so constraint trees are easy to solve
- *Unfortunately*, many CGs have width greater than one (that is, no equivalent tree), so we still need to improve search

So what if we don’t have a tree?

- Answer #1: Try **interleaving** constraint propagation and backtracking
- Answer #2: Try using **variable-ordering** heuristics to improve search
- Answer #3: Try using **value-ordering** heuristics during variable instantiation
- Answer #4: See if **iterative repair** works better
- Answer #5: Try using **intelligent backtracking** methods

Intelligent backtracking

- **Backtracking search is chronological backtracking**
- **Backjumping**:
  - Jumps to the most recent assignment in the *conflict set*
  - if V_j fails, jump back to the variable V_i with greatest i such that the constraint (V_i, V_j) fails (i.e., most recently instantiated variable in conflict with V_i)

Intelligent backtracking

- **Backchecking**: keep track of incompatible value assignments computed during backjumping
- **Backmarking**: keep track of which variables led to the incompatible variable assignments for improved backchecking

Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- Variable selection: randomly select any conflicted variable

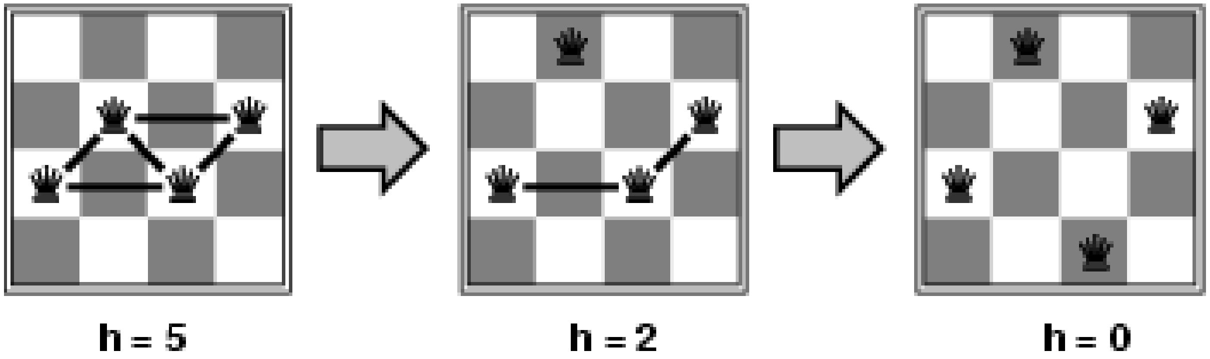
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with $h(n)$ = total number of violated constraints

Local search for CSPs

- Min-conflicts: Select new values that minimally conflict with the other variables
  - Use in conjunction with hill climbing or simulated annealing or…
- Local maxima strategies
  - Random restart
  - Random walk
  - Tabu search: don’t try recently attempted values

Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks



- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., n = 10,000,000)

Min-conflicts heuristic

- Iterative repair method
  1. Find some “reasonably good” initial solution
     - E.g., in N-queens problem, use greedy search through rows, putting each queen where it conflicts with the smallest number of previously placed queens, breaking ties *randomly*
  2. Find a variable in conflict (randomly)
  3. Select a new value that minimizes the number of constraint violations
     - O(N) time and space
  4. Repeat steps 2 and 3 until done
- Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution

Some challenges for constraint reasoning

- What if not all constraints can be satisfied?
  - Hard vs. soft constraints
  - Degree of constraint satisfaction
  - Cost of violating constraints
- What if constraints are of different forms?
  - Symbolic constraints
  - Numerical constraints [constraint solving]
  - Temporal constraints
  - Mixed constraints

Some challenges for constraint reasoning II

- What if constraints are represented intensionally?
  - Cost of evaluating constraints (time, memory, resources)
- What if constraints, variables, and/or values change over time?
  - Dynamic constraint networks
  - Temporal constraint networks
  - Constraint repair
- What if you have multiple agents or systems involved in constraint satisfaction?
  - Distributed CSPs
  - Localization techniques

Distributed Constraint Satisfaction

- Looks at solving CSP when there is a collection of agents, each of which controls a subset of the constraint variables.

- Active area of research; annual conferences and workshops.

Thanks for coming -- see you next Tuesday!