More Constraint Satisfaction

CMPSCI 383
October 13, 2011

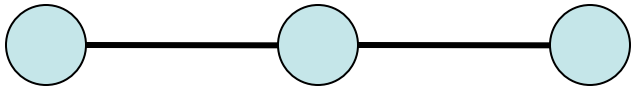
Today's lecture

- A review of CSPs
- Local search for CSPs
- Taking advantage of the structure of the CSP
- Some applications

What defines a CSP?

- In CSPs, states are defined by assignments of values to a set of variables $X_1 ... X_n$. Each variable X_i has a domain D_i of possible values.
- States are evaluated based on their consistency with a set of constraints $C_1 ... C_m$ over the values of the variables.
- A goal state is a complete assignment to all variables that satisfies all the constraints.

Local Consistency

- Node Consistency: satisfies all unary constraints
- Arc Consistency: satisfies all binary constraints
- Path Consistency: 
- n-consistency: for any consistent assignment to any set of n-1 variables, a consistent value can be found for any n-th variable.

Arc Consistency

X is arc-consistent with respect to Y if for every value in Dx there is a value in Dy that satisfies the binary constraint on arc (X,Y).

- Note:
  - Not symmetric in general
  - To make X ac with respect to Y, remove values from Dx
  - To make Y ac with respect to X, remove values from Dy

Arc Consistency (slightly different from the book)

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., 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) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

Standard Search Formulation

Let's start with the straightforward approach, then fix it

States are defined by the values assigned so far

- Initial state: the empty assignment { }
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment
  → fail if no legal assignments

- Goal test: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth *n* with *n* variables
   → use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. b = (n - ℓ)d at depth ℓ, hence n! · d^n leaves (d is domain size)

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

  → b = d and there are d^n leaves

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

- Backtracking search is the basic uninformed algorithm for CSPs

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

Backtracking Search (the book's)

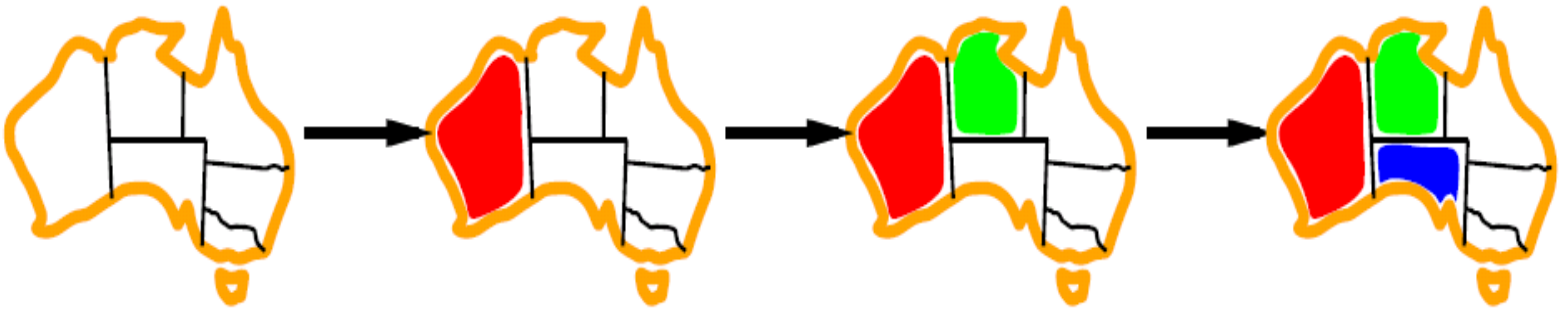
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Improving backtracking efficiency

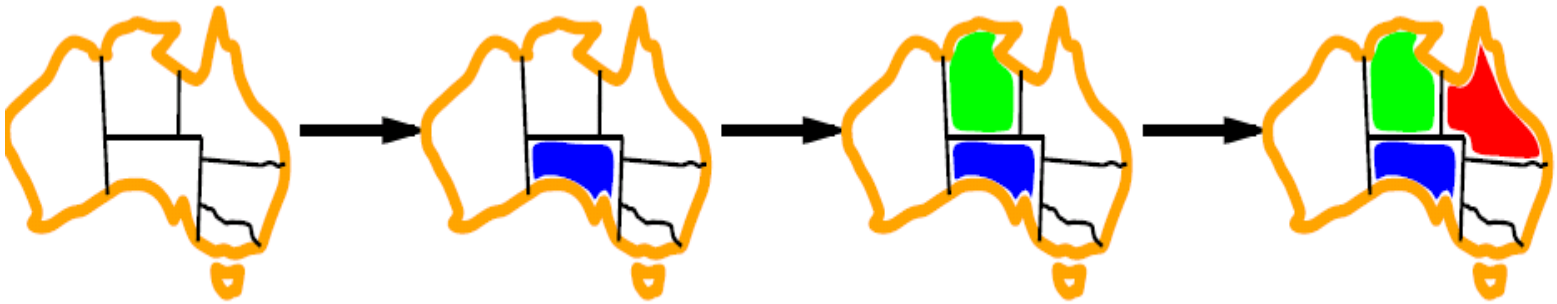
- Basic question: What next step should our search procedure take?
- Approaches
  - Minimum remaining values heuristic
  - Degree heuristic
  - Least-constraining value heuristic

Minimum remaining values (MRV) heuristic



- Select the *most constrained* variable (the variable with the smallest number of remaining values)

Degree heuristic



- Select the variable that is involved in the largest number of constraints with other unassigned variables
- The most constraining variable.
- A useful tie breaker (and guide to starting).
- In what order should its values be tried?

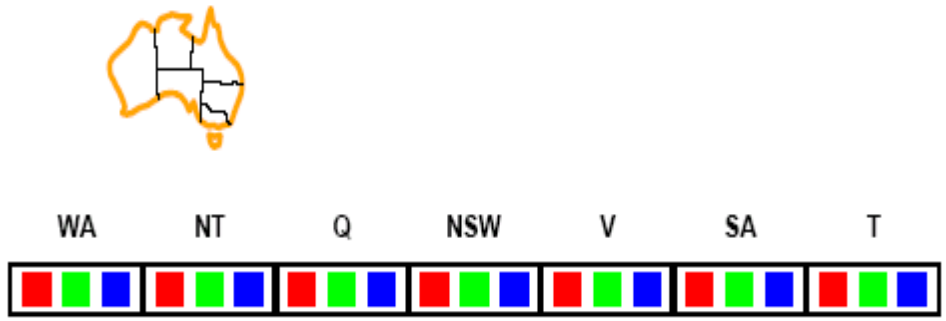
Least constraining value

- Given a variable, choose the *least constraining value* — the value that leaves the maximum flexibility for subsequent variable assignments.
- Combining these makes 1000 Queens possible.

Combining Search with Inference

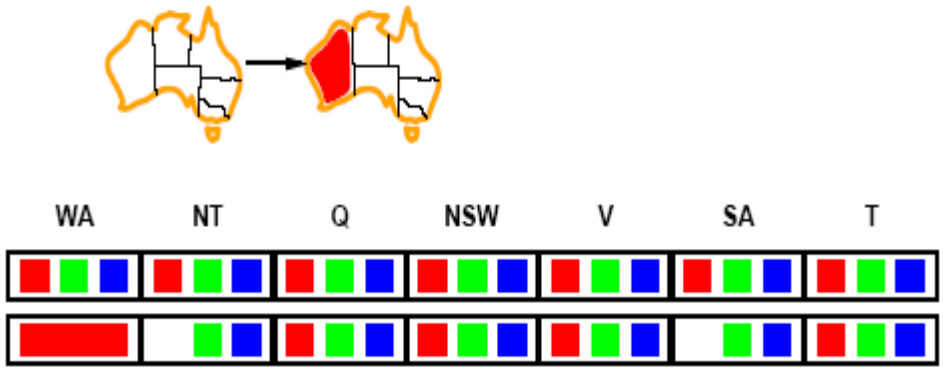
- Basic question — Can we provide better information to these heuristics?
- Forward checking
  - Precomputing information needed by MRV
  - Early stopping
- Constraint propagation
  - Arc consistency (2-consistency)
  - n-consistency

Forward checking



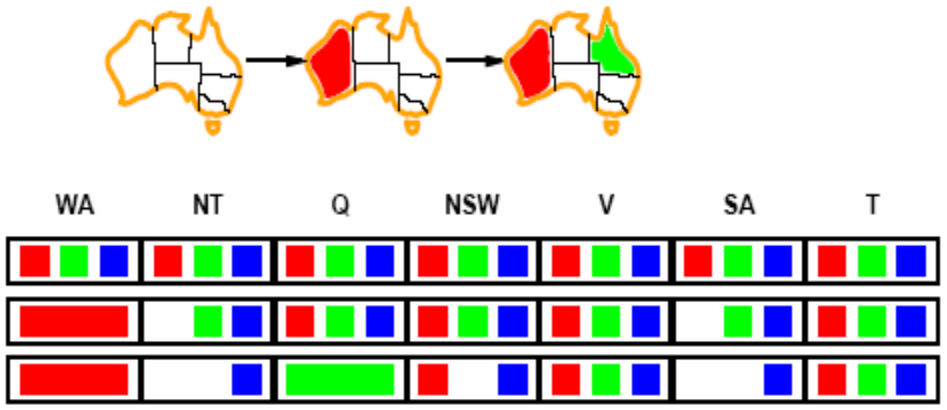
- Can we detect inevitable failure early?
  - *And avoid it later?*
- Yes — track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values.

Forward checking



- Assign *{WA=red}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be red*
  - *SA can no longer be red*

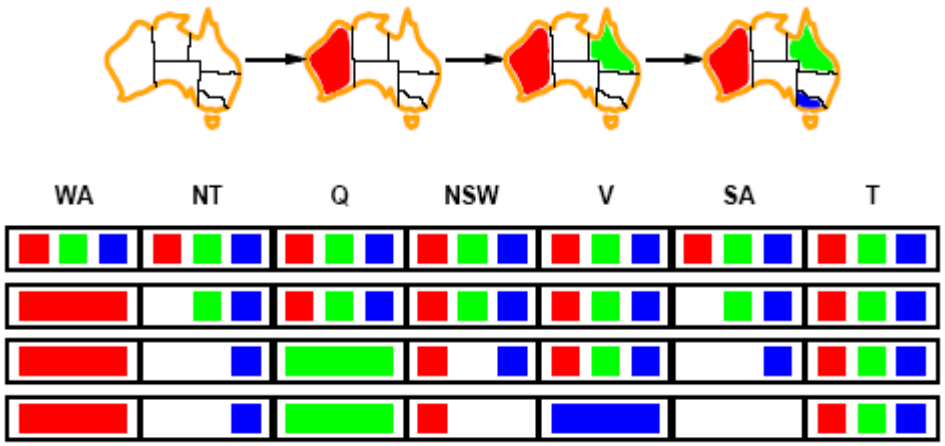
Forward checking



Note: not MRV choice

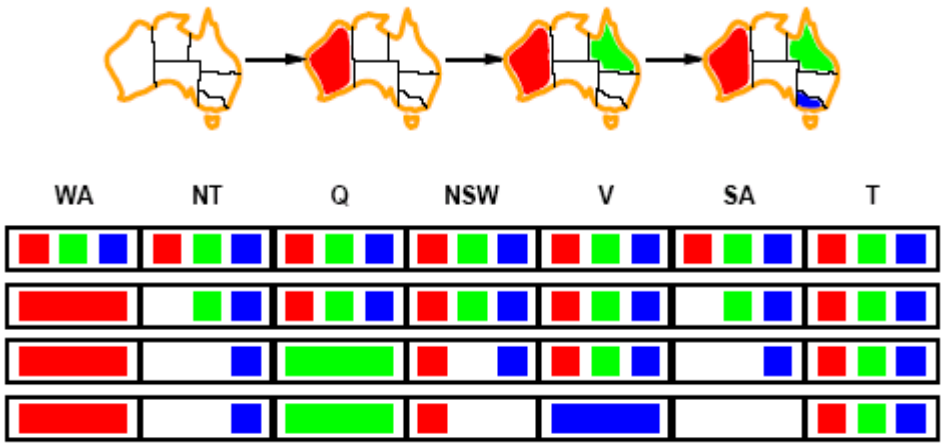
- Assign *{Q=green}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

Forward checking



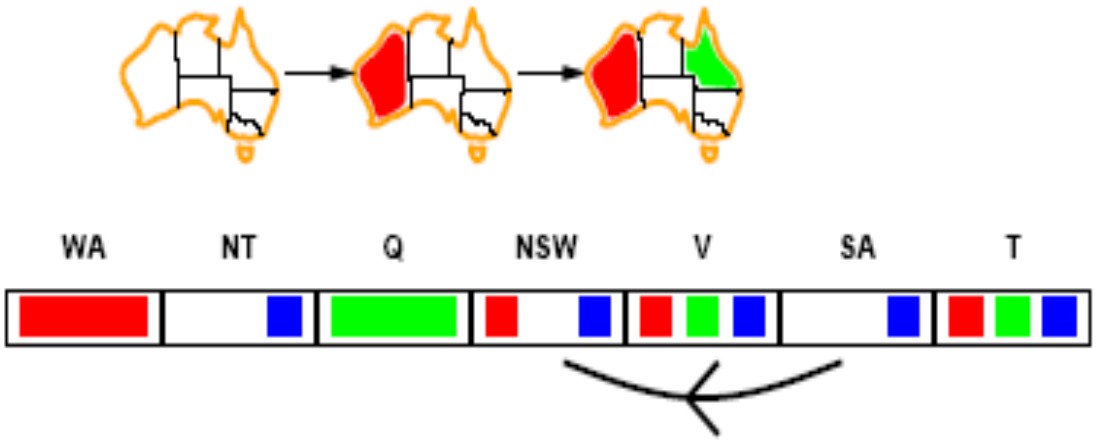
- If *V* is assigned *blue*
- Effects on other variables connected to WA
  - *SA is empty*
  - *NSW can no longer be blue*
- FC has detected a partial assignment that is *inconsistent* with the constraints.

Forward checking



- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC checking propagates information from assigned to unassigned variables but does not provide detection for all failures.
  - NT and SA cannot be blue!
- Makes each current variable assignment arc consistent, but does not look far enough ahead to detect all inconsistencies (as AC-3 would)

Arc consistency

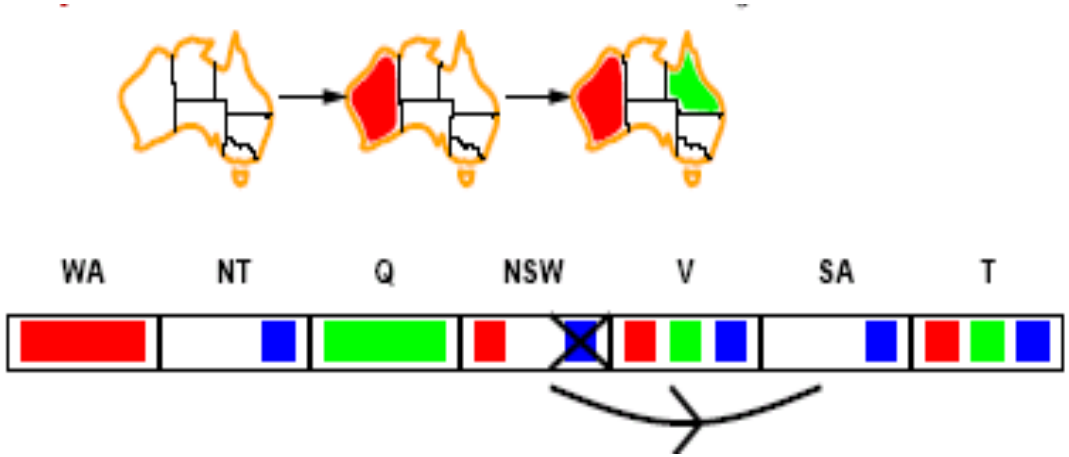


- *X* → *Y* is consistent iff

  for *every* value *x* of *X* there is some allowed *y*
- *SA* → *NSW* is consistent iff

  SA=blue and *NSW=red*

Arc consistency



- *X* → *Y* is consistent iff

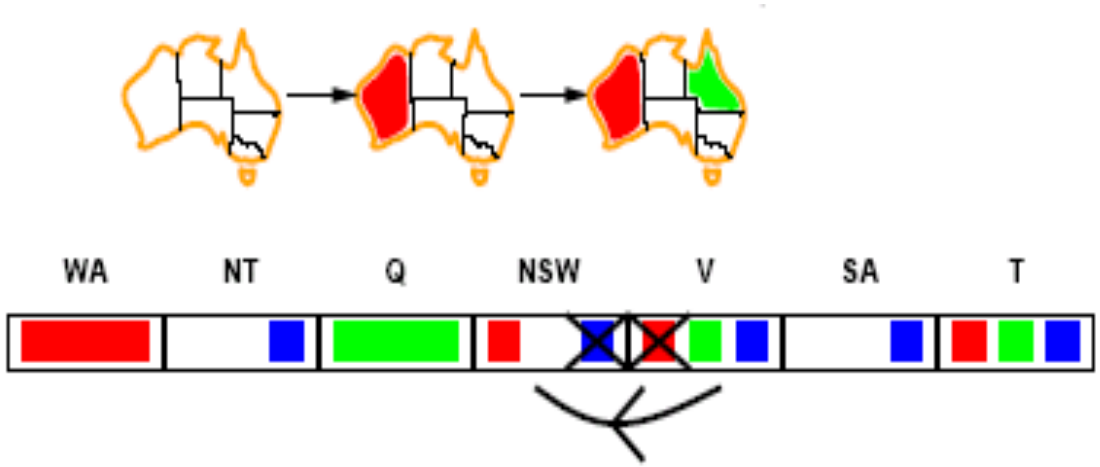
  for *every* value *x* of *X* there is some allowed *y*
- *NSW* → *SA* is consistent iff

  NSW=red and *SA=blue*

  NSW=blue and SA=???

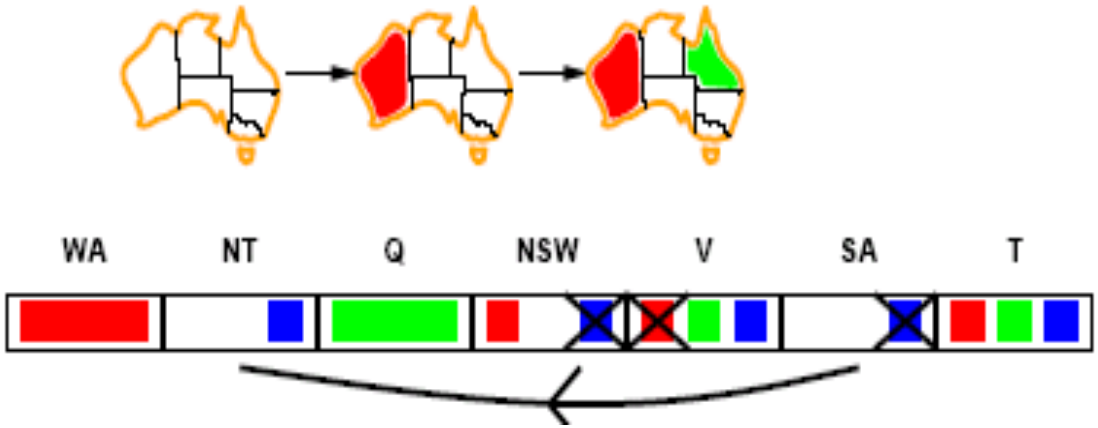
Arc can be made consistent by removing *blue* from *NSW*

Arc consistency



- Arc can be made consistent by removing *blue* from *NSW*
- Recheck *neighbours*
  - Remove red from *V*

Arc consistency



- Arc can be made consistent by removing *blue* from *NSW*
- Recheck *neighbours*
  - Remove red from *V*
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment.
  - Repeated until no inconsistency remains

n-consistency

- Arc consistency does not detect all inconsistencies:
  - Partial assignment *{WA=red, NSW=red}* is inconsistent.
- Stronger forms of propagation can be defined using the notion of n-consistency.
- A CSP is n-consistent if for any set of n-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any nth variable.
  - E.g. 1-consistency or node-consistency
  - E.g. 2-consistency or arc-consistency
  - E.g. 3-consistency or path-consistency

Further improvements

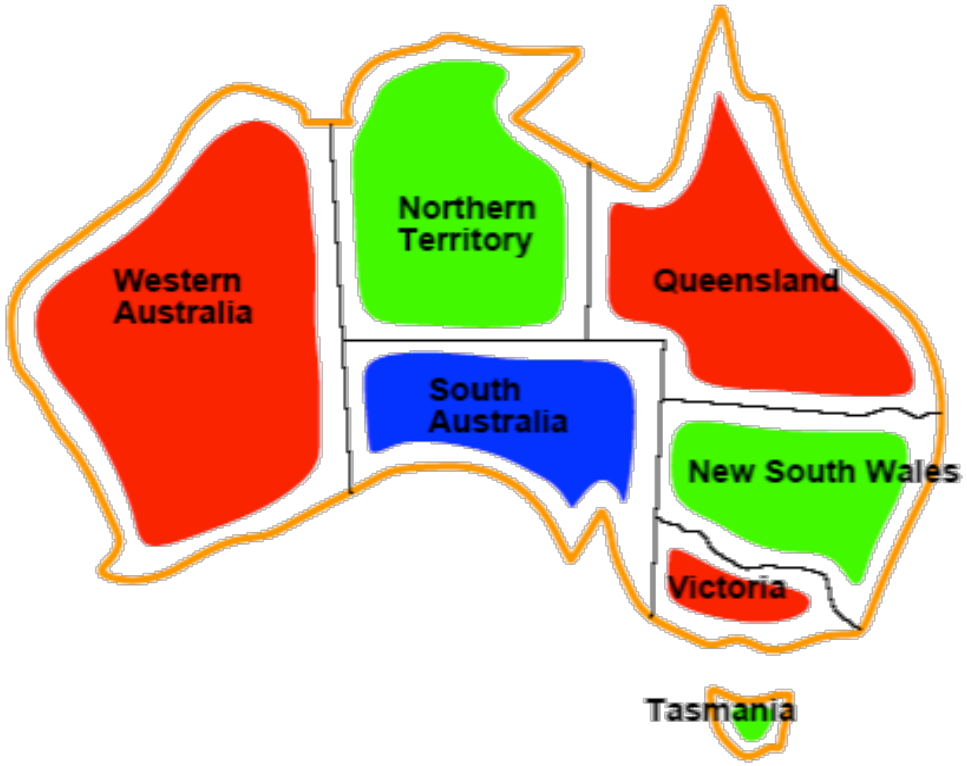
- Checking special constraints
  - Checking *Alldif*(…) constraint
  - Checking *Atmost*(…) constraint
    - *Bounds propagation for larger value domains*
- Intelligent backtracking
  - Standard form is chronological backtracking i.e. try different value for preceding variable.
  - More intelligent, backtrack to conflict set.
    - Set of variables that caused the failure or set of previously assigned variables that are connected to X by constraints.
    - Backjumping moves back to most recent element of the conflict set.
    - Forward checking can be used to determine conflict set.

Key Ideas so far

- Basic form of a CSP
- Different types of CSPs
- Types of constraints
- Consistent assignment
- Complete assignment
- Constraint graph
- Constraint propagation
- Backtracking search for CSPs
- Heuristics to improve backtracking search
  - MRV
  - Degree heuristic
  - Least-constraining value
- Interleaving search and inference
  - Forward checking
  - Arc consistency
  - Smart backtracking

Can this be a *local* search problem?

- Do we need the path to the solution or only the solution itself?
- Can we apply local search methods?
  - Hillclimbing
  - Simulated annealing
  - Genetic algorithms
- What's a state?



What are good heuristics for choosing moves in local search when solving CSPs?

Min-conflicts heuristic for local search

- To enable local search
  - 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

Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Actions: move queen in column
- Goal test: no attacks; $h(n) = 0$
- 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). Average of 50 steps for n = 1M.

Why can local search work well?

What's hard about CSPs?

Where the *Really* Hard Problems Are

Peter Cheeseman
RIACS*

Bob Kanefsky
Sterling Software

William M. Taylor
Sterling Software

Artificial Intelligence Research Branch
NASA Ames Research Center, Mail Stop 244-17
Moffett Field, CA 94035, USA
Email: <last-name>@ptolemy.arc.nasa.gov

Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI-91*, pages 331-337, San Mateo, CA, 1991. Morgan Kaufmann.

Abstract

It is well known that for many NP-complete problems, such as K-Sat, etc., typical cases are easy to solve; so that computationally hard cases must be rare (assuming $P \neq NP$). This paper shows that NP-complete problems can be summarized by at least one "order parameter", and that the hard problems occur at a critical value of such a parameter. This critical value separates two regions of characteristically different properties. For example

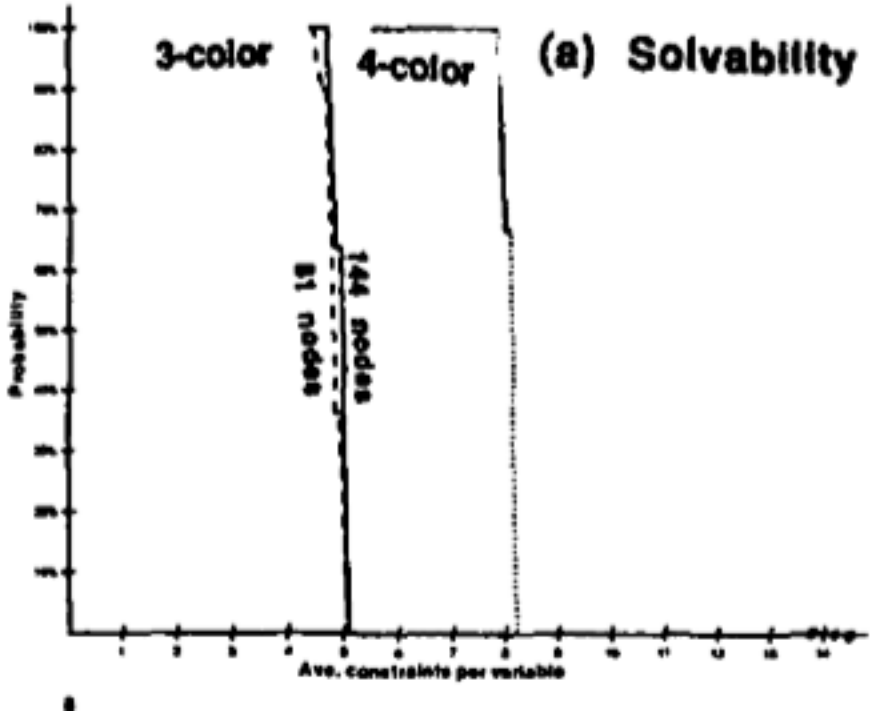
so says nothing about the difficulty of typical instances. However, this situation raises the question "where are the really hard instances of NP problems?". Can a subclass of problems be defined that is typically (exponentially) hard to solve, or do worst cases appear as rare "pathological cases" scattered unpredictably in the problem space?

In this paper we show that for many NP problems one or more "order parameters" can be defined, and hard instances occur around particular critical values of these order parameters. In addition, such critical values form a boundary that separates the space of problems into two regions. One region is underconstrained, so the density of solutions is high, thus making it relatively easy to find a solution. The other region is overconstrained and very unlikely to contain a solution. If there are solutions in this overconstrained region, then they have such

What's hard about CSPs?

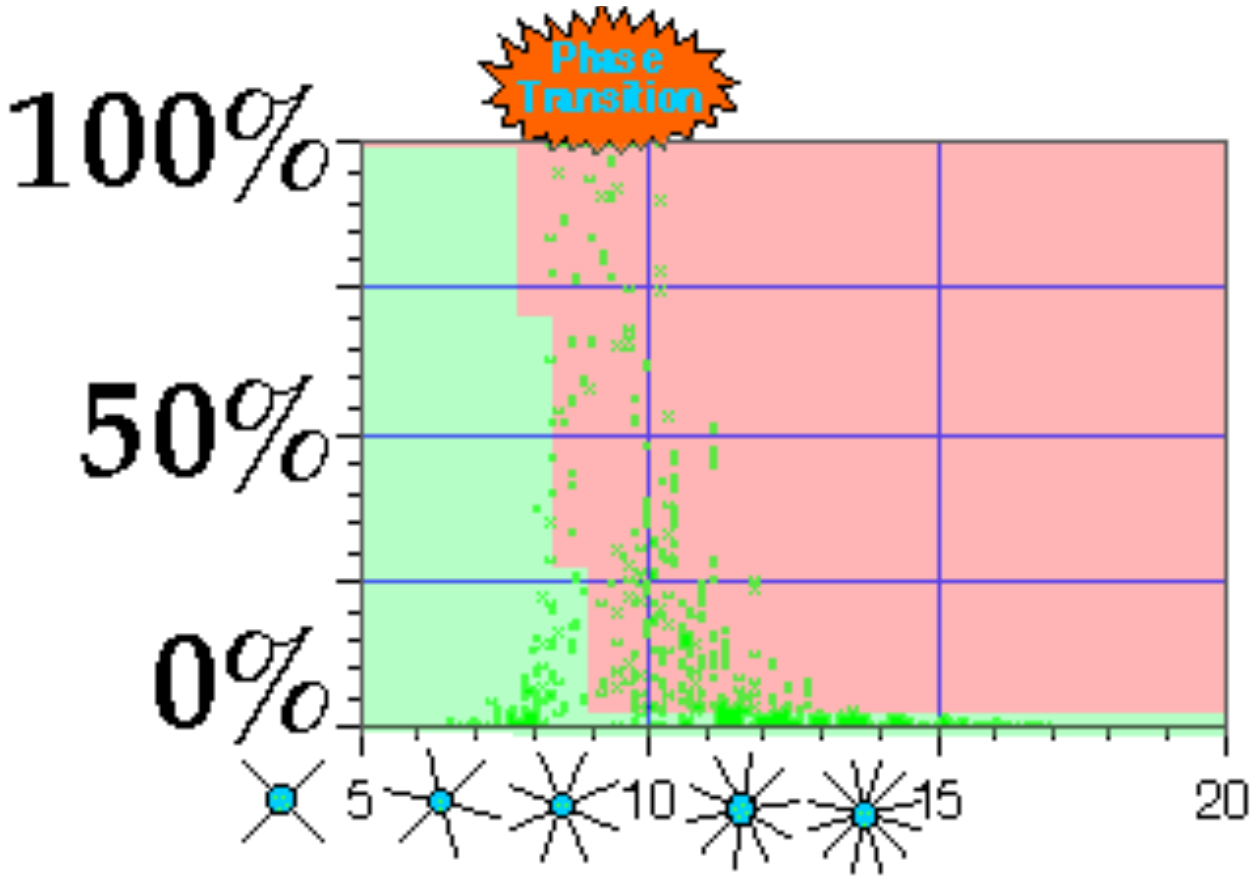
Graph coloring (color the graph or report impossible): NP complete but "almost always easy"

prob. of solution



avg. constraints per variable

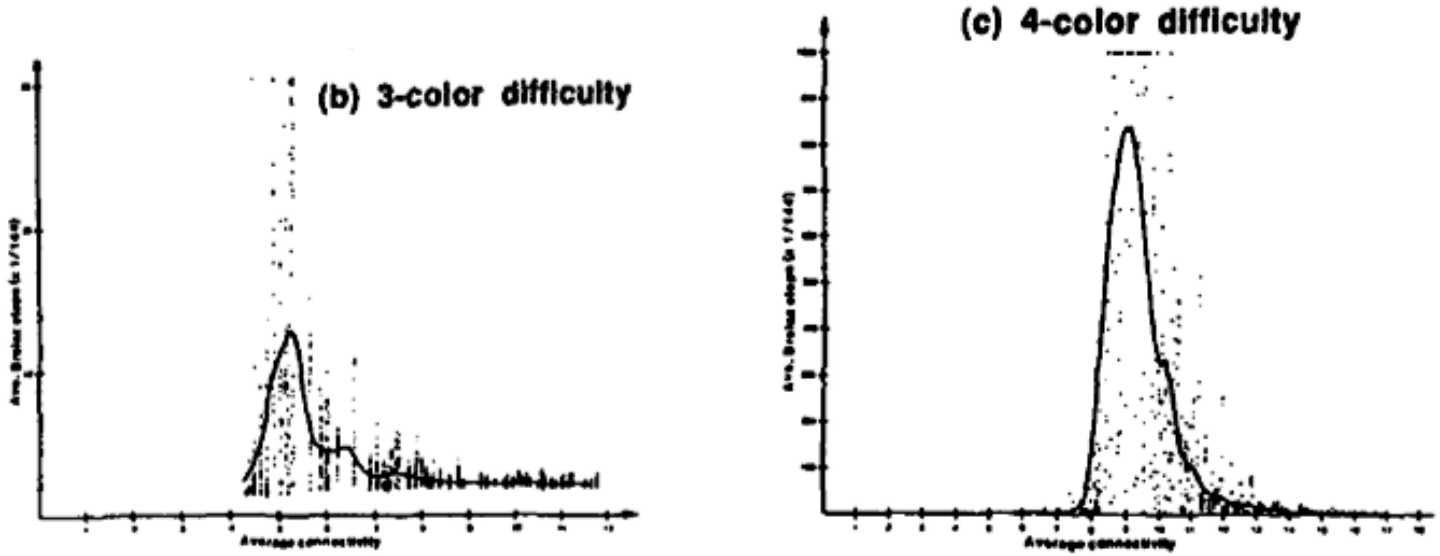
What's hard about CSPs?



What's hard about CSPs?

Graph coloring (color the graph or report impossible):
NP complete but "almost always easy"

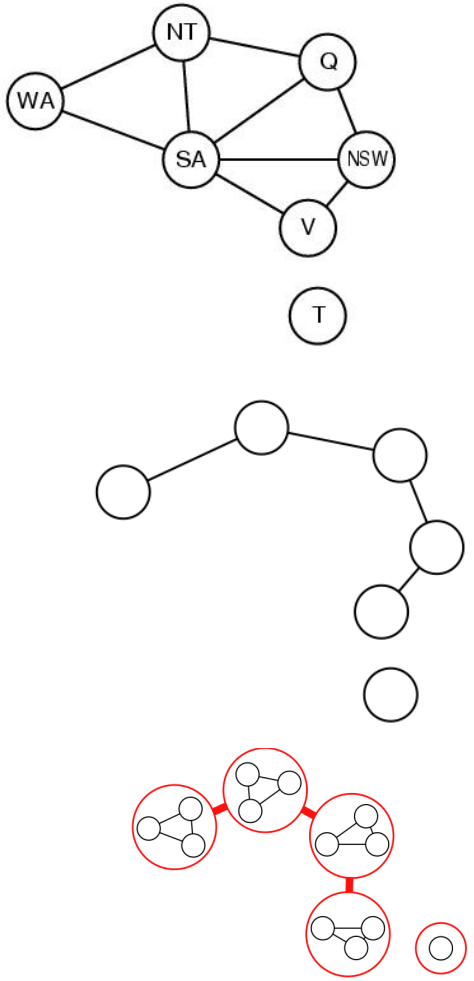
computational cost using backtrack algorithm (with MRV, breaking ties with degree heuristic)



avg. constraints per variable

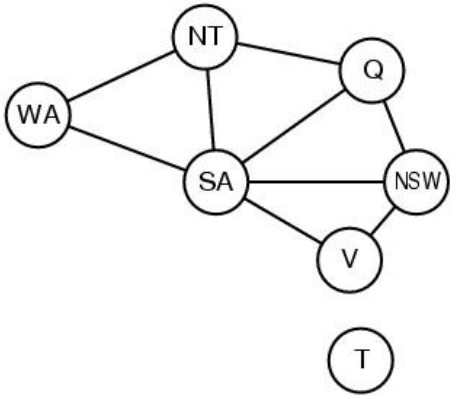
Exploiting the structure of CSPs

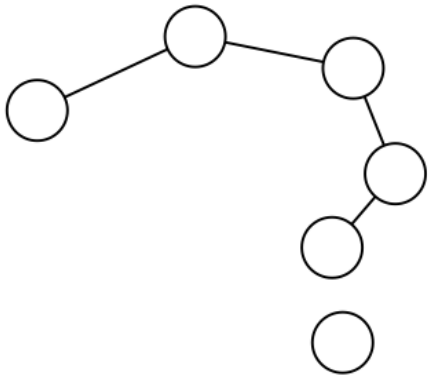
- Decompose into independent problems
- Tree-structured CSPs can be solved in linear time
- Reduce problems to tree-structured CSPs
  - Cycle cutset conditioning — Remove nodes to create trees
  - Tree decomposition — Decompose problem into a tree-structured set of subproblems



Cycle cutset conditioning

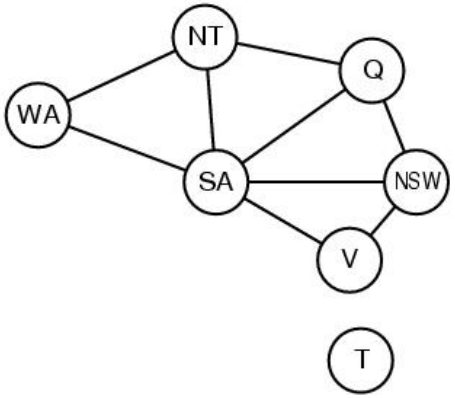
- Want to create a tree
  - What is a tree?
  - Why do we want to create one?
  - *Tree-structured CSPs solvable in linear time*
- Create a tree by deleting nodes
  - How can you delete nodes in CSPs?
  - *Set value and restrict domains*
- Does this always work well?
  - No, what can we do about that?
  - *Step through possible settings*
- What's the payoff?
  - *Big efficiency gains* $O(d^c \cdot (n-c)d^2)$

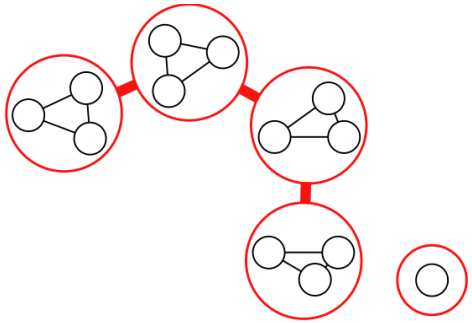




Tree decomposition

- Again, want to create a tree
  - What's another way of creating a tree?
  - *Merging nodes*
- What are the rules for doing this?
  - *Every variable in ≥1 subproblems*
  - *All connected variable pairs, and assoc. constraints, in ≥1 subproblems*
  - *If a variable appears in 2 subproblems, it must appear in all subproblems on the path connecting the two subproblems*
- Now, how can we solve this new problem?





Key Ideas for CSPs

- Basic form of a CSP
- Different types of CSPs
- Types of constraints
- Consistent assignment
- Complete assignment
- Constraint graph
- Constraint propagation
- Backtracking search for CSPs
- Heuristics to improve backtracking search
  - MRV
  - Degree heuristic
  - Least-constraining value
- Interleaving search and inference
  - Forward checking
  - Arc consistency
  - Smart backtracking
- Local search
  - Min-conflicts heuristic
- Using problem structure
  - Decomposing into independent subproblems
  - Turn into a tree structured problem
    - Cutset conditioning
    - Tree decomposition

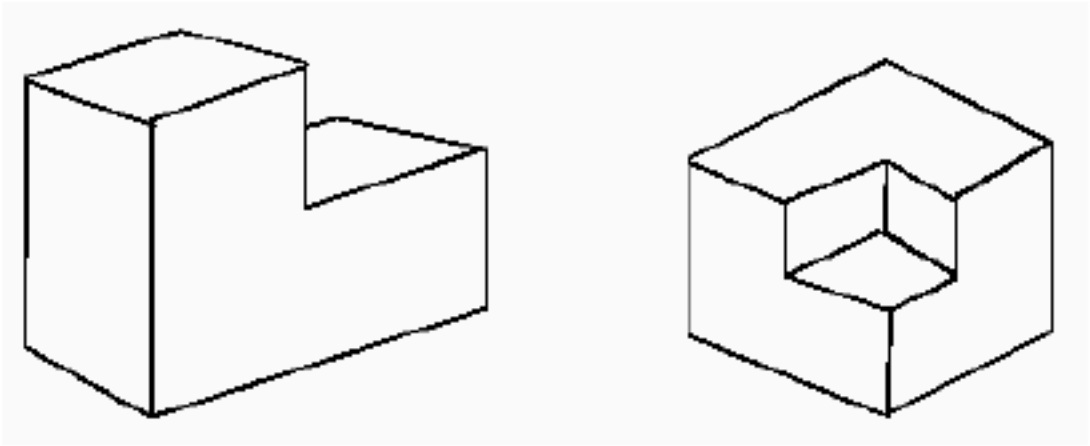
Thanks for Andrew Moore…

Note to other teachers and users of these slides. Andrew would be delighted if you found this source material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. PowerPoint originals are available. If you make use of a significant portion of these slides in your own lecture, please include this message, or the following link to the source repository of Andrew's tutorials: http://www.cs.cmu.edu/~awm/tutorials . Comments and corrections gratefully received.

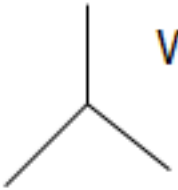
The Waltz algorithm

One of the earliest examples of a computation posed as a CSP.

The Waltz algorithm is for interpreting line drawings of solid polyhedra.



Look at all intersections.



What kind of intersection could this be? A concave intersection of three faces? Or an external convex intersection?

Adjacent intersections impose constraints on each other. Use CSP to find a unique set of labelings. Important step to "understanding" the image.

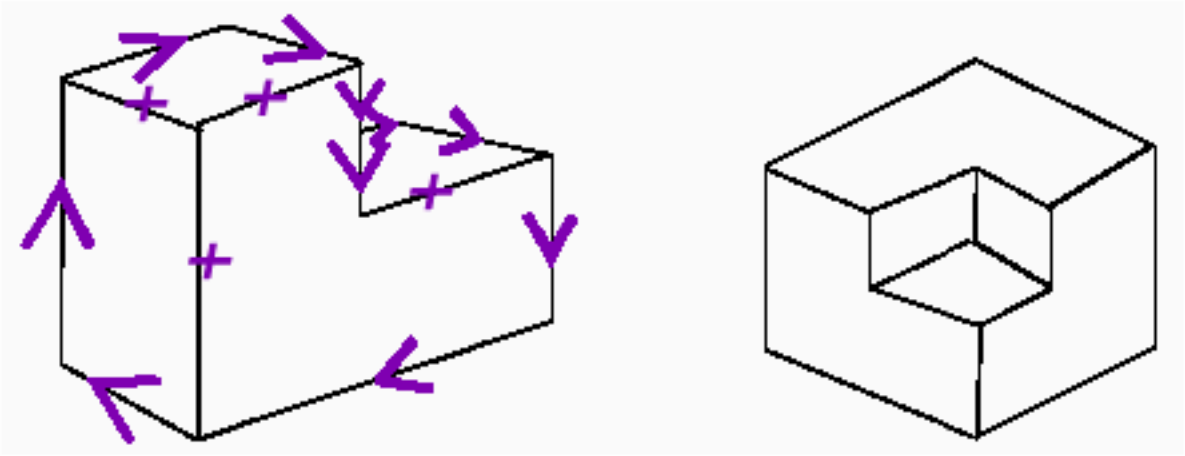
Waltz Alg. on simple scenes

Assume all objects:

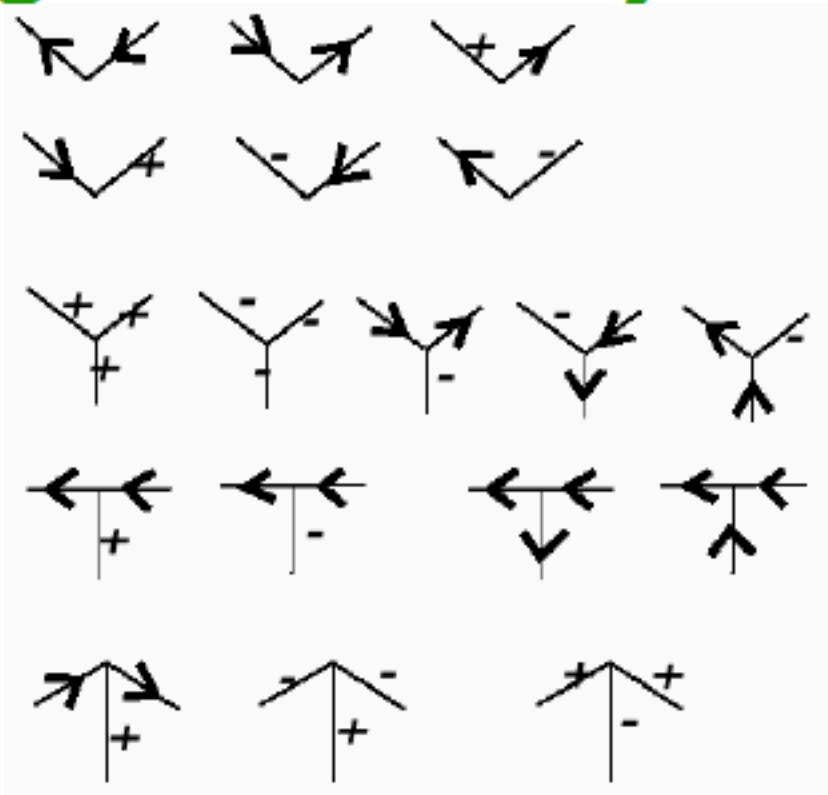
- Have no shadows or cracks
- Three-faced vertices
- "General position": no junctions change with small movements of the eye.

Then each line on image is one of the following:

- Boundary line (edge of an object) (<) with right hand of arrow denoting "solid" and left hand denoting "space"
- Interior convex edge (+)
- Interior concave edge (-)



18 legal kinds of junctions

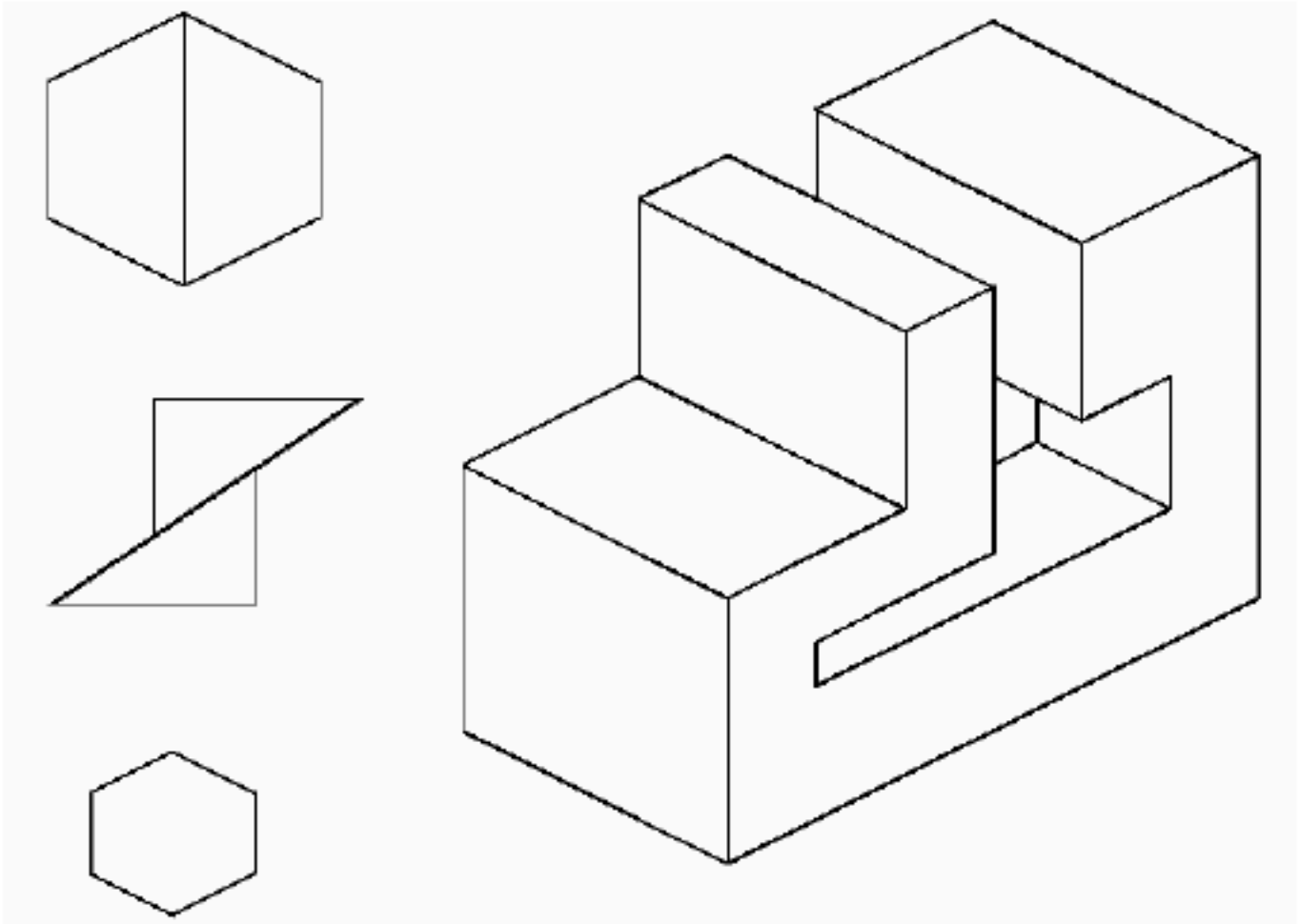


Given a representation of the diagram, label each junction in one of the above manners.

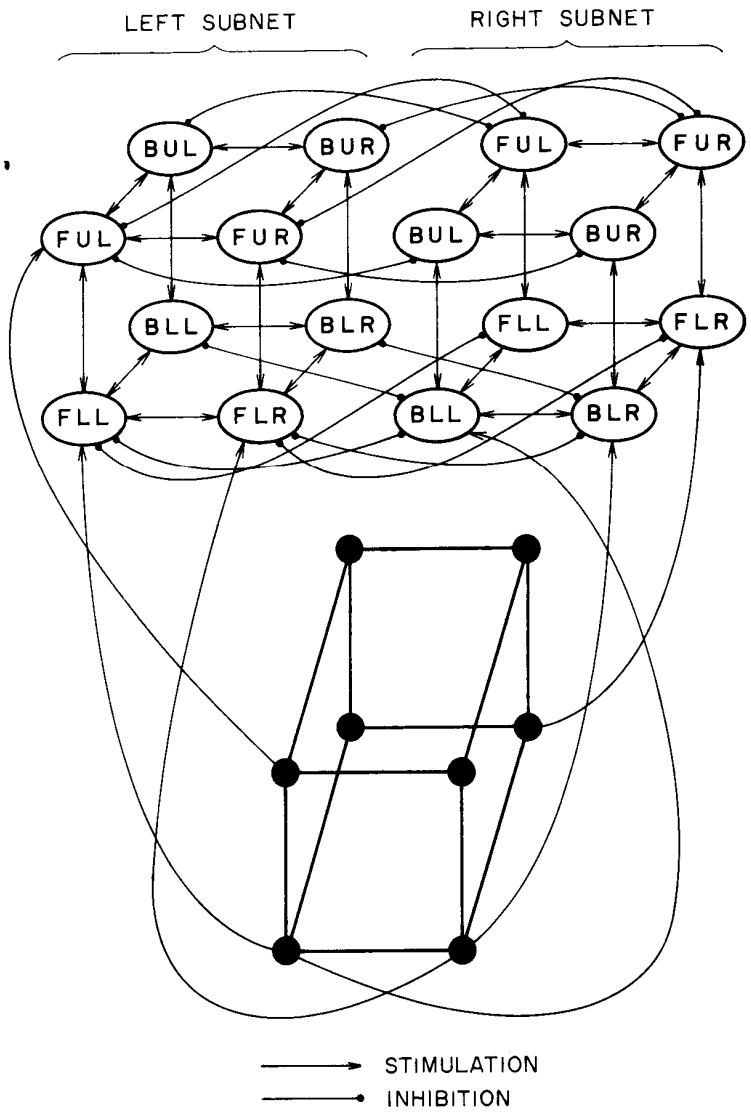
The junctions must be labeled so that lines are labeled consistently at both ends.

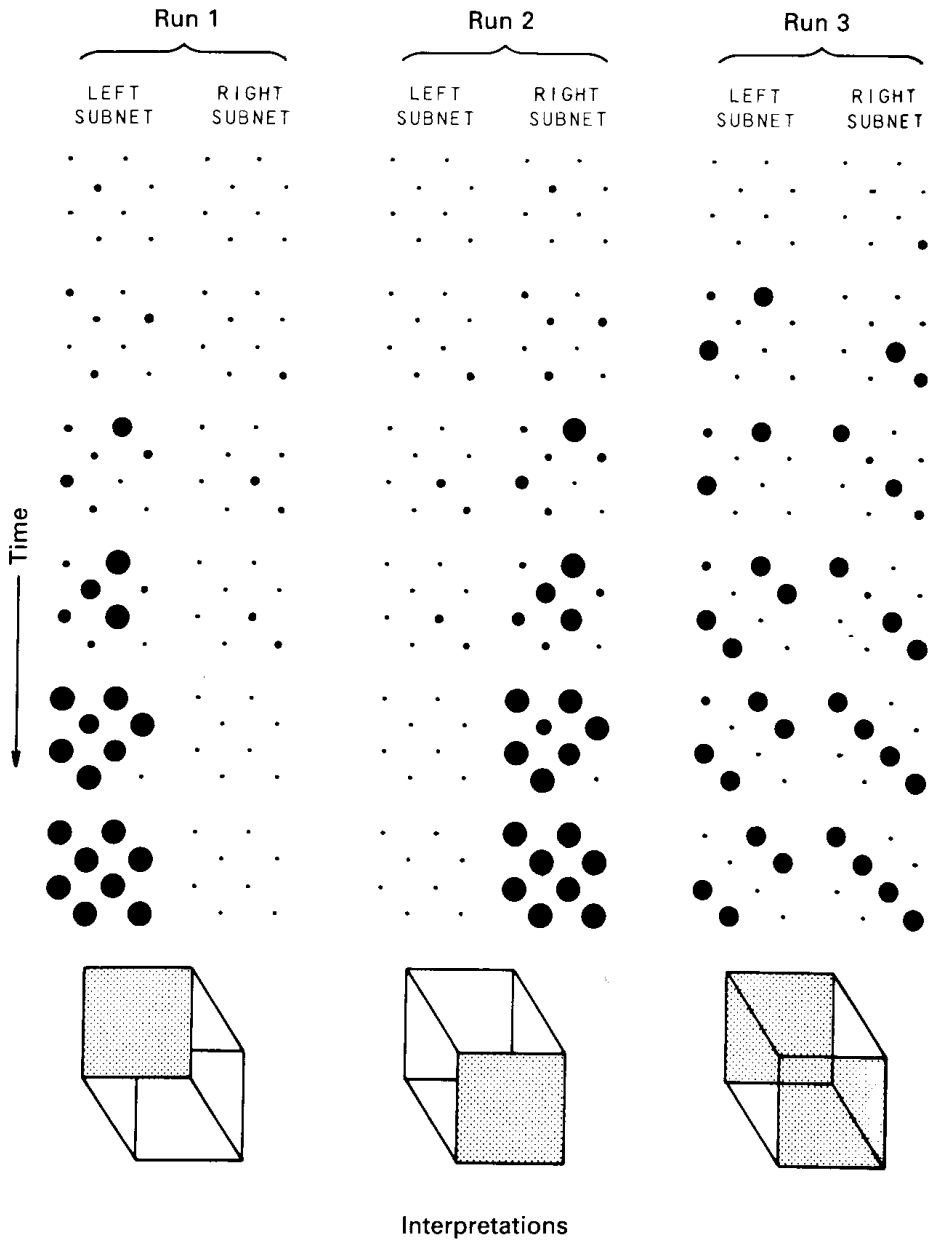
Can you formulate that as a CSP? *FUN FACT: Constraint Propagation always works perfectly.*

Waltz Examples



Constraint satisfaction via a Neural Network




Run 1
Run 2
Run 3
LEFT SUBNET
RIGHT SUBNET
LEFT SUBNET
RIGHT SUBNET
LEFT SUBNET
RIGHT SUBNET
Time
Interpretations

Scheduling

A very big, important use of CSP methods.

- Used in many industries. Makes many multi-million dollar decisions.
- Used extensively for space mission planning.
- Military uses.

People *really care* about improving scheduling algorithms!

Problems with phenomenally huge state spaces. But for which solutions are needed very quickly.

Many kinds of scheduling problems e.g.:

- *Job shop*: Discrete time; weird ordering of operations possible; set of separate jobs.
- *Batch shop*: Discrete or continuous time; restricted operation of ordering; grouping is important.
- *Manufacturing cell*: Discrete, automated version of open job shop.

Job Shop scheduling

At a job-shop you make various products. Each product is a "job" to be done. E.G.

Job_1 = Make a polished-thing-with-a-hole
Job_2 = Paint and drill a hole in a widget

Each job requires several operations. E.G.

Operations for Job_1: Polish, Drill
Operations for Job_2: Paint, Drill

Each operation needs several resources. E.G.

Polishing needs the Polishing machine
Polishing needs Pat (a Polishing expert)
Drilling needs the Drill
Drilling needs Pat (also a Drilling expert)
Or Drilling can be done by Chris

Some operations need to be done in a particular order (e.g. Paint after you've Drilled)

Job Shop Formalized

A Job Shop problem is a pair (J , RES)

J is a set of jobs $J = \{j_1 , j_2 , \ldots j_n\}$

RES is a set of resources $RES = \{R_1 .. R_m\}$

Each job j_l is specified by:

- a set of operations $O^l = \{O^l_1 \; O^l_2 \; \ldots \; O^l_{n(l)} \}$
- and must be carried out between release-date rd_l and due-date dd_l.
- and a partial order of operations: (O^l_i before O^l_j), ($O^l_{i'}$ before $O^l_{j'}$), etc…

Each operation O^l_i has a variable start time st^l_i and a fixed duration du^l_i and requires a set of resources. e.g.: O^l_i requires { R^l_{i1} , R^l_{i2} … }.

Each resource can be accomplished by one of several possible physical resources, e.g. R^l_{i1} might be accomplished by any one of {r^l_{ij1} , r^l_{ij2} , …}. Each of the r^l_{ijk}s are a member of RES.

Job Shop Example

j_1 = *polished-hole-thing* = { O^1_1 , O^1_2 }

j_2 = *painted-hole-widget* = { O^2_1 , O^2_2 }

RES = { Pat,Chris,Drill,Paint,Drill,Polisher }

O^1_1 = *polish-thing: need resources…*

{ R^1_{11} = *Pat* , R^1_{12} = *Polisher* }

O^1_2 = *drill-thing: need resources…*

{ R^1_{21} = (r^1_{211}=*Pat* or r^1_{212}=*Chris*), R^1_{22} = *Drill* }

O^2_1 = *paint-widget: need resources…*

{ R^2_{11} = *Paint* }

O^2_2 = *drill-widget : need resources…*

{ R^2_{21} = (r^2_{211}=*Pat* or r^2_{212}=*Chris*), R^2_{22} = *Drill* }

Precedence constraints : O^2_2 before O^2_1. All operations take one time unit du^l_i = 1 forall *i*,*l*. Both jobs have release-date rd^l = 0 and due-date dd^l = 1.

Job-shop: the Variables and Constraints

Variables

- The operation state times st^l_i
- The resources R^l_{ij} (usually these are obvious from the definition of O^l_i. Only need to be assigned values when there are alternative physical resources available, e.g. *Pat* or *Chris* for operating the *drill*).

Constraints:

- Precedence constraints. (Some O^l_is must be before some other O^l_js).
- Capacity constraints. There must never be a pair of operations with overlapping periods of operation that use the same resources.

Non-challenging question. Can you schedule our Job-shop?

Next Class: Tuesday Oct 18

- Problem set 2 due
- Review for midterm:

  Everything **except** Constraint Satisfaction