Constraint Satisfaction Problems

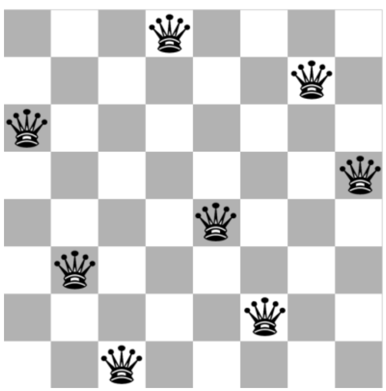
Chapter 6.1 – 6.4

Derived from slides by S. Russell and P. Norvig, A. Moore, and R. Khoury

Constraint Satisfaction Problems (CSPs)

- Standard search problem:
  - **state** is a "black box" – any data structure that supports successor function, heuristic function, and goal test
- CSP:
  - **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

Example: 8-Queens



Example: 8-Queens

- Variables: 64 squares, number of queens

  V = {$S_{1,1}$, $S_{1,2}$, …, $S_{8,8}$, Number_of_queens}
- Values: Queen or no queen

  $S_{i,j} \in D_S$ = {queen, empty}

  Number_of_queens $\in D_N$ = [0, 64]
- Constraints: Attacks, queen count

  {Number_of_queens = 8,

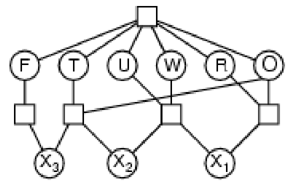
  $S_{i,j}$ = queen $\Rightarrow S_{i,j+n}$ = empty,

  $S_{i,j}$ = queen $\Rightarrow S_{i+n,j}$ = empty,

  $S_{i,j}$ = queen $\Rightarrow S_{i+n,j+n}$ = empty}
- States: All board configurations
  - 2.8×10^{14} complete states
  - 1.8×10^{14} complete states with 8 queens
  - 92 complete and consistent states
  - 12 unique complete and consistent states

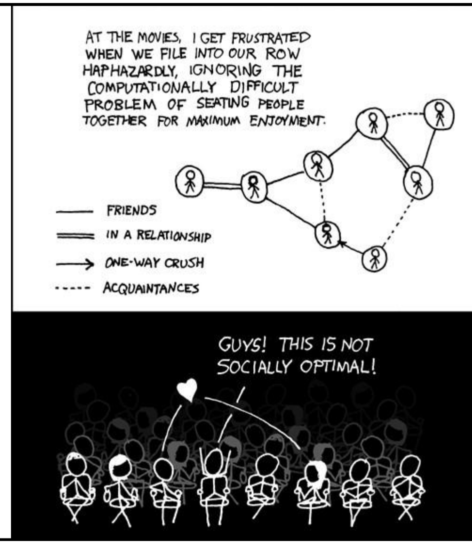
Example: Cryptarithmetic

```
  T W O
+ T W O
-------
F O U R
```



- **Variables**: $F, T, U, W, R, O, X_1, X_2, X_3$
- **Domains**: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- **Constraints**: *Alldiff* (*F, T, U, W, R, O*)
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F, T \neq 0, F \neq 0$

Movie Seating Problem

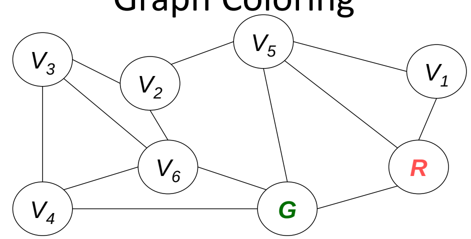


Some Applications of CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetable problems
  - e.g., which class is offered when and where?
- Scheduling problems
- VLSI or PCB layout problems
- Boolean satisfiability
- N-Queens
- Graph coloring
- Games: Minesweeper, Magic Squares, Sudoku, Crosswords
- Line-drawing labeling

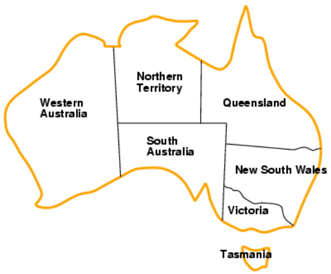
- Notice that many real-world problems involve real-valued variables

A Constraint Satisfaction Problem: Graph Coloring



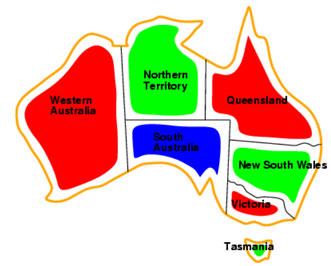
- Inside each circle marked V_1 .. V_6 we must assign: *R*, *G* or *B*
- No two adjacent circles may be assigned the same value
- Notice that two circles have already been given an assignment

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)}

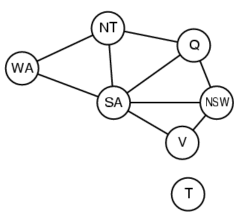
Example: Map-Coloring



Solutions are **complete** (i.e., all variables are assigned values) and **consistent** (i.e., does not violate any constraints) 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, Boolean satisfiability
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

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

Local Search for CSPs

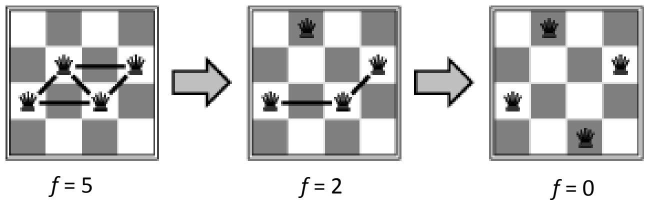
- Hill-climbing, simulated annealing, genetic algorithms typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints, i.e., hill-climb with $f(n)$ = total number of violated constraints

Local Search

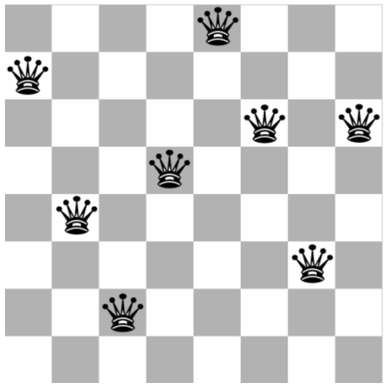
- **Min-Conflicts Algorithm**:
  - Assign to each variable a random value
  - While state not consistent
    - Pick a variable (randomly or with a heuristic) that has constraints violated
    - Find values that minimize the *total* number of violated constraints (over all variables)
    - If there is only one such value
      - Assign that value to the variable
    - If there are several values
      - Assign a random value from that set to the var

Example: 4-Queens

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



Min-Conflicts Algorithm



Min-Conflicts Algorithm

- Advantages
  - Simple and Fast: Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 1,000,000 can be solved on average in about 50 steps!)
- Disadvantages
  - Only searches states that are reachable from the initial state
    - Might not search all state space
  - Does not allow worse moves (but can move to neighbor with *same* cost)
    - Might get stuck in a local optimum
  - Not complete
    - Might not find a solution even if one exists

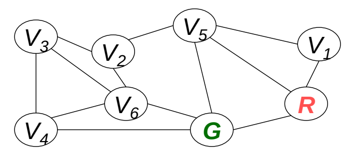
Standard Tree Search Formulation

States are defined by all the values *assigned so far*

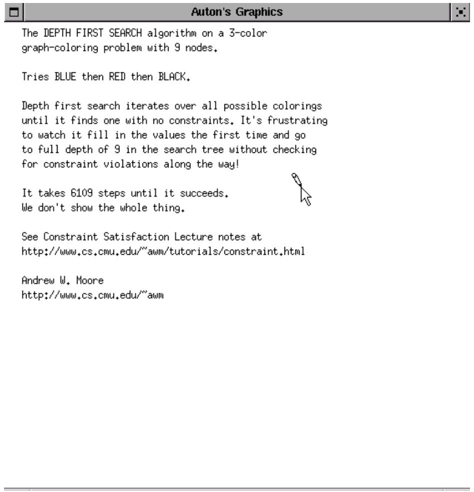
- **Initial state**: the empty assignment { }
- **Successor function**: assign a value to an unassigned variable
- **Goal test**: the current assignment is complete: all variables assigned a value and all constraints satisfied

- Find ***any*** solution, so cost is not important
- Every solution appears at depth *n* with *n* variables
  → use depth-first search

DFS for CSPs



- Variable assignments are **commutative**}, i.e.,
  [WA=R then NT=G] same as [NT=G then WA=R]
- What happens if we do DFS with the order of assignments as *B* tried first, then *G*, then *R*?
- **Generate-and-test strategy**: Generate candidate solution, then test if it satisfies all the constraints
- This makes DFS look very stupid!
- Example: http://www.cs.cmu.edu/~awm/animations/constraint/9d.html



Improved DFS: Backtracking w/ Consistency Checking

- Don't ever try a successor that causes inconsistency with its neighbors, i.e., perform **consistency checking** when node is generated
- Successor function assigns a value to an unassigned variable that does ***not*** conflict with current assignments
  - Fail if no legal assignments (i.e., no successors)
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve *n*-Queens for $n \approx 25$

Backtracking w/ Consistency Checking

Start with empty state

while not all vars in state assigned a value **do**

  Pick a variable (randomly or with heuristic)

  if it has a value that does not violate any constraints

    then Assign that value

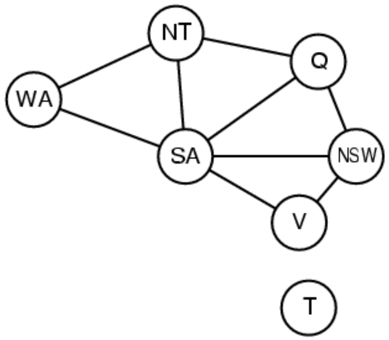
  else

    Go back to previous variable

    Assign it another value

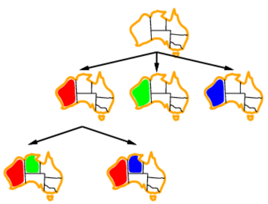
Backtracking Example

Australia Constraint Graph

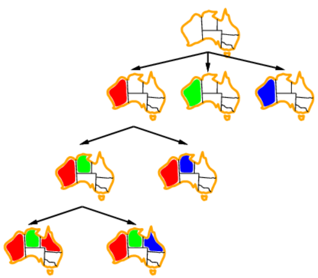


Backtracking Example

Backtracking Example



Backtracking Example



Backtracking Search

- Depth-first search algorithm
  - Goes down one variable at a time
  - In a dead end, back up to last variable whose value can be changed without violating any constraints, and change it
  - If you backed up to the root and tried all values, then there are no solutions
- Algorithm is complete
  - Will find a solution if one exists
  - Will expand the entire (finite) search space if necessary
- Depth-limited search with limit = *n*

Auton's Graphics

The BACKTRACKING algorithm on a 3-color graph-coloring problem with 9 nodes.

Tries BLUE then RED then BLACK.

This prunes parts of the depth first search as soon as it notices a violation. Beats the heck out of DFS, though it still backtracks a little bit.

It takes 15 steps until it succeeds.

See Constraint Satisfaction Lecture notes at http://www.cs.cmu.edu/~awm/tutorials/constraint.html

Andrew W. Moore
http://www.cs.cmu.edu/~awm

Auton's Graphics

The BACKTRACKING algorithm on a 3-color graph-coloring problem with 27 nodes.

Tries BLUE then RED then BLACK.

This prunes parts of the depth first search as soon as it notices a violation. But notice how early decisions mean that no matter what it tries, for a long time nothing will work up in the top left node.

It takes 65448 steps until it succeeds.

See Constraint Satisfaction Lecture notes at http://www.cs.cmu.edu/~awm/tutorials/constraint.html

Andrew W. Moore
http://www.cs.cmu.edu/~awm

Top-left node is hard to label!

Improving Backtracking Efficiency

- General-purpose **heuristics** 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?

Which Variable Next?
Most Constrained Variable

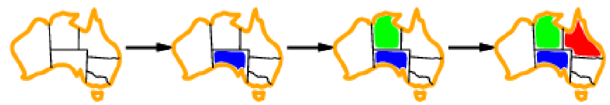
- ***Most constrained*** variable:

  choose the variable with the *fewest* legal values

- Called **minimum remaining values (MRV)** heuristic
- Tries to cut off search asap

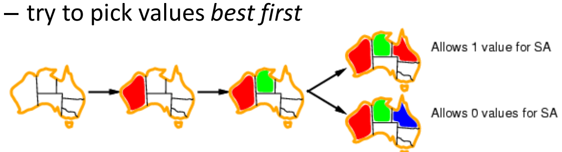
Which Variable Next?
Most Constraining Variable

- Tie-breaker among most constrained variables
- ***Most constraining*** variable:
  - choose the variable with the *most* constraints on remaining variables
- Called **degree heuristic**
- Tries to cut off search asap



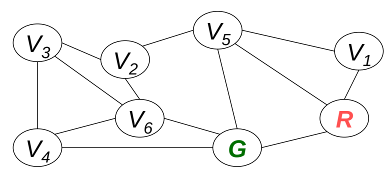
Which Value Next?
Least Constraining Value

- Given a variable, choose the ***least constraining*** value:
  - i.e., the one that rules out the *fewest values* in the remaining variables
  - try to pick values *best first*



- Combining these heuristics makes 1000-Queens feasible

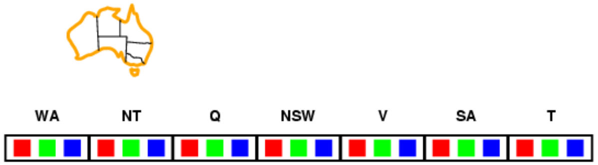
Improvement: **Forward Checking**



- At start, for each variable, record the current **set of *all* possible legal values for it**
- When you assign a value to a variable in the search, *update the set of legal values for* ***all*** *unassigned variables*. *Backtrack immediately if you* ***empty*** *a variable's set of possible values*.
  - What happens if we do DFS with the order of assignments as *B* tried first, then *G*, then *R*?
  - Example: http://www.cs.cmu.edu/~awm/animations/constraint/27f.html

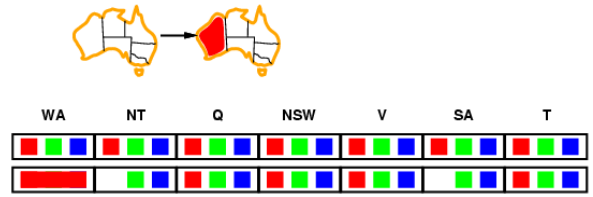
Forward Checking Algorithm

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



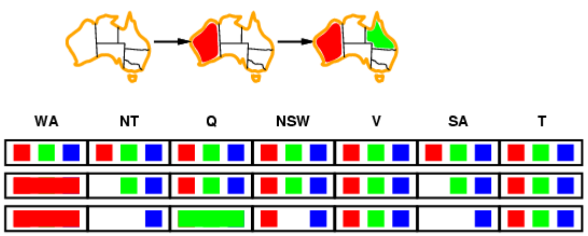
Forward Checking

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



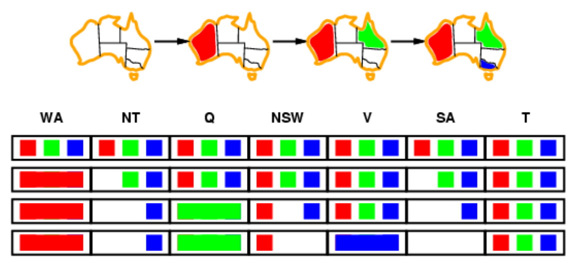
Forward Checking

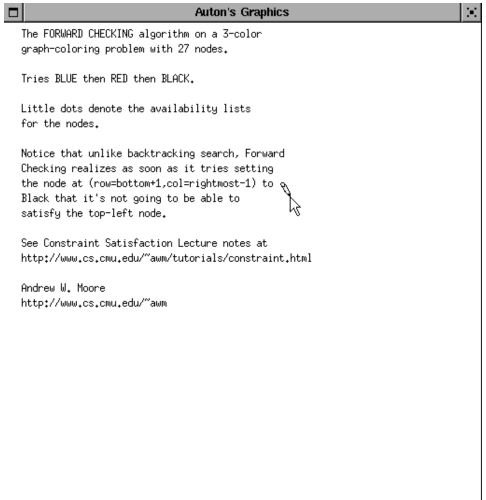
- Idea:
  - Keep track of remaining legal values for all unassigned variables
  - Terminate search when any variable has no legal values



Forward Checking

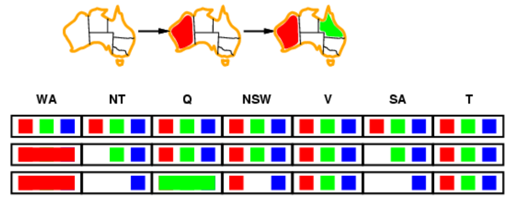
- Idea:
  - Keep track of remaining legal values for all unassigned variables
  - Terminate search when any variable has no legal values





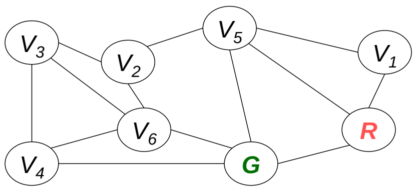
Constraint Propagation

- 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** repeatedly enforces constraints locally with its neighbors

Constraint Propagation



Main idea: When you delete a value from your domain, check all variables connected to you. If any of them change, delete all inconsistent values connected to them, etc.

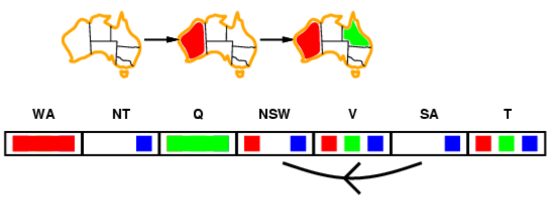
In the above example, nothing changes

Web Example:
http://www.cs.cmu.edu/~awm/animations/constraint/27p.html

Arc Consistency

- Simplest form of propagation makes each **arc consistent**
- *X* → *Y* is consistent iff

  for every value *x* at *X* there is some allowed *y*, i.e., there is at least 1 value of *Y* that is consistent with *x*

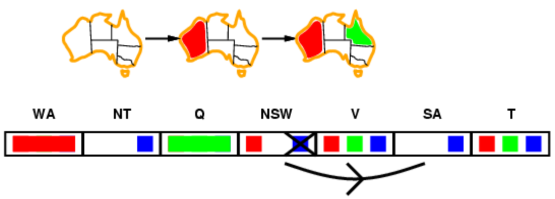


X = SA
Y = NSW

Arc Consistency

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

  for every value x at X there is some allowed y; if not, delete x

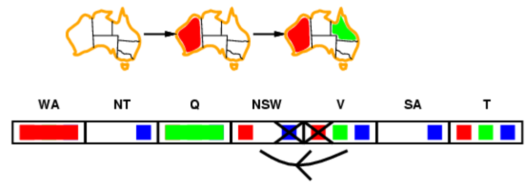


X = NSW
Y = SA

Arc Consistency

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

  for every value x at X there is some allowed y; if not, delete x



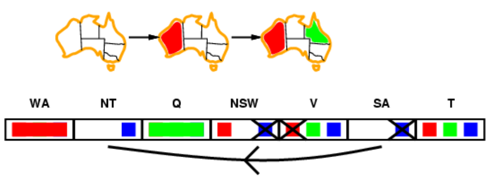
X = V
Y = NSW

- If X loses a value, all neighbors of X need to be rechecked

Arc Consistency

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

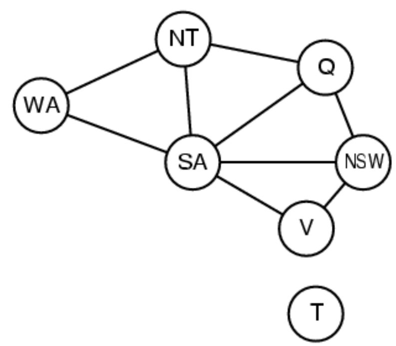
  for every value x at X there is some allowed y; if not, delete x

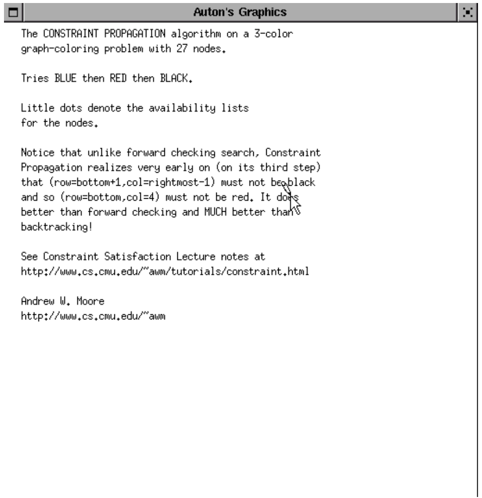


X = SA
Y = NT

- If X loses a value, all 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

Australia Constraint Graph





Bottom-right node must not be red because node to upper-left must not be black

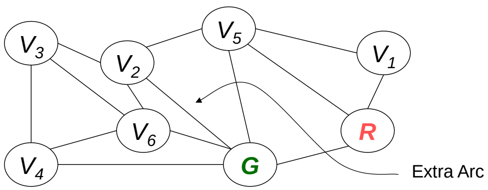
Arc Consistency Algorithm "**AC-3**"

function AC-3(*csp*) // returns false if inconsistency is found and true otherwise
  // input: *csp*, a binary CSP with components (*X*, *D*, *C*)
  // local variables: *queue*, a queue of arcs; initially all arcs in *csp*
  while *queue* not empty **do**
    (X_i, X_j) = Remove-First(*queue*)
    if Revise(*csp*, X_i, X_j) **then** **// make arc consistent**
      if size of D_i = 0 **then return** *false*
      foreach X_k in X_i.Neighbors – { X_j } **do** **// propagate changes to neighbors**
        add (X_k, X_i) to *queue*
  return *true*

function Revise(*csp*, X_i, X_j) // returns *true* iff we revise the domain of X_i
  revised = *false*
  foreach *x* **in** D_i **do**
    if no value *y* in D_j allows (*x*, *y*) to satisfy the constraint betweeen X_i and X_j **then**
      {delete *x* from D_i ; *revised* := *true*}
  return *revised*

Check if $X_i \rightarrow X_j$ consistent

Constraint Propagation



- In this example, constraint propagation solves the problem without search … Not always that lucky!
- Constraint propagation can be done as a preprocessing step (cheap)
- Or it can be performed dynamically during the search. Expensive: when you backtrack, you must undo some of your additional constraints.

Combining Search with CSP

- Idea: Interleave search and CSP inference
- Perform DFS
  - At each node assign a selected value to a selected variable
  - Run CSP to check if any inconsistencies arise as a result of this assignment

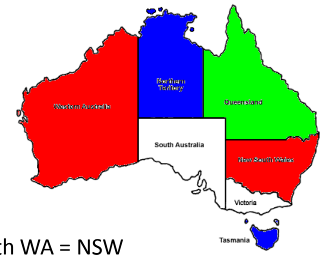
Combining Backtracking Search with CSP

```
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
```

Conflict-Directed Backjumping

- Suppose we color Australia in this order:
  - WA = R
  - NSW = R
  - T = B
  - NT = B
  - Q = G
  - SA = ?



- Deadend at SA
  - No possible solution with WA = NSW
  - Backtracking will try to change T on the way, even though it has nothing to do with the problem, before going to NSW

Slide credit: R. Khoury

Conflict-Directed Backjumping

- Backtracking goes back one level in the search tree at a time
  - Chronological backtracking
- Not rational in cases where the previous step is *not* involved with the conflict
- **Conflict-Directed Backjumping**
  - Go back to a variable involved in the conflict
  - Skip several levels if needed to get there
  - *Non*-chronological backtracking

Slide credit: R. Khoury

Conflict-Directed Backjumping

- Maintain a ***conflict set*** for each variable
  - List of **previously-assigned variables that are related by constraints**

conf(WA) = {}
conf(NSW) = {}
conf(T) = {}
conf(NT) = {WA}
conf(Q) = {NSW,NT}
conf(SA) = {WA,NSW,NT,Q}

- When we hit a deadend, backjump to the most recent variable in the conflict set

Slide credit: R. Khoury

Conflict-Directed Backjumping

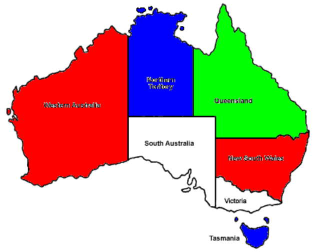
- Learn from a conflict by updating the conflict set of the variable we jumped to
- Example: Conflict at X_j and backjump to X_i
  - conf(X_i) = {X_1, X_2, X_3}
  - conf(X_j) = {X_3, X_4, X_5, X_i}
- conf(X_i) = conf(X_i) $\cup$ conf(X_j) – {X_i}

  = {X_1, X_2, X_3, X_4, X_5}
- X_i *absorbed* the conflict set of X_j

Slide credit: R. Khoury

Conflict-Directed Backjumping

conf(WA) = {}
conf(NSW) = {WA}
conf(NT) = {WA,NSW}
conf(Q) = {WA,NSW,NT}

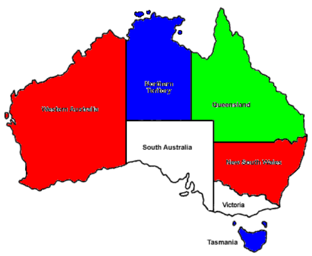


- SA's domain empty ⇒ deadend
- SA backjumps to most recent var in conf(SA): Q
  - Update conf(Q) = {WA, NSW, NT}
  - Meaning: "There is no consistent solution from Q=G onwards, given preceding assignments WA=R, NSW=R and NT=B"

Slide credit: R. Khoury

Conflict-Directed Backjumping

conf(WA) = {}
conf(NSW) = {WA}
conf(NT) = {WA,NSW}
conf(Q) = {WA,NSW,NT}

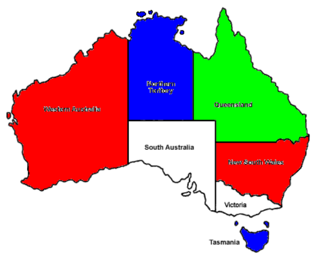


- Q's domain empty ⇒ deadend
- Q backjumps to NT (i.e., most recent var in conf(Q))
  - Update conf(NT) = {WA, NSW}
  - There is no consistent solution from NT=B onwards, given preceding assignments WA=R and NSW=R

Slide credit: R. Khoury

Conflict-Directed Backjumping

conf(WA) = {}
conf(NSW) = {WA}
conf(NT) = {WA,NSW}
conf(Q) = {WA,NSW,NT}

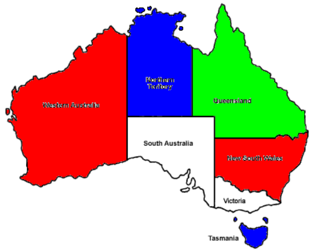


- Try NT=G (which is consistent with WA=R, NSW=R)
  - Retrying Q and SA fails again
  - So, there is no consistent solution from NT=G onwards, given preceding assignments WA=R and NSW=R
- NT's domain now empty ⇒ deadend

Slide credit: R. Khoury

Conflict-Directed Backjumping

conf(WA) = {}
conf(NSW) = {WA}
conf(NT) = {WA,NSW}
conf(Q) = {WA,NSW,NT}



- NT backjumps to NSW
  - Update conf(NSW) = {WA}
  - Skips T, which is irrelevant in this conflict
  - Discovers the relationship between NSW and WA, which is not present in our constraints, so try NSW=G …

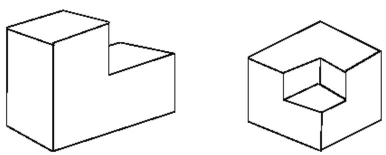
Slide credit: R. Khoury

Constraint Learning

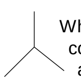
- When a contradiction occurs, remember the minimum set of variables from the conflict set that was responsible for the problem
- Save these "**no-goods**" as *new constraints* so that they are never attempted again somewhere else in search
- For example, {WA=R, NSW=R, NT=B}

The Waltz Algorithm

- One of the earliest examples of a computation posed as a CSP
- The Waltz algorithm is used 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.

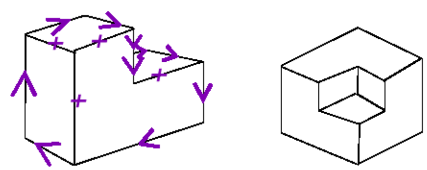
Waltz Algorithm on Simple Scenes

Assume all objects:

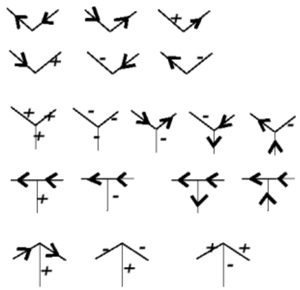
- Have no shadows or cracks
- Three-faced vertices
- "General position": no junctions change with small movements of the eye.

Then each line on image is one of the following 3 types:

- 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

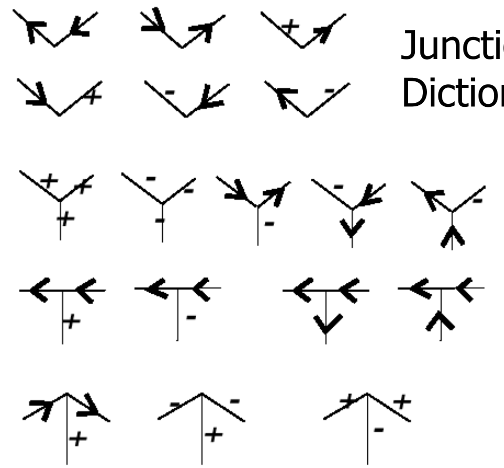


Given a representation of the diagram, label each junction in one of the above ways.

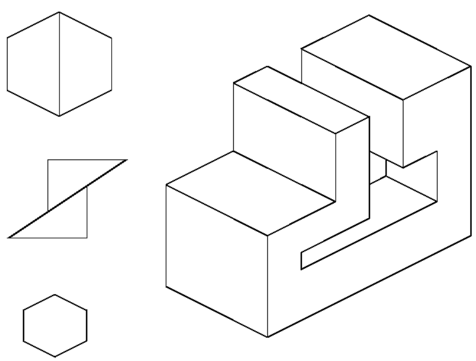
The junctions must be labeled so that lines are labeled consistently at both ends.

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

Junction Dictionary



Waltz Examples



Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node plus simple consistency checking
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice