Vision Computer group heidelberg

IWR Interdisciplinary Center for Scientific Computing

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Artificial Intelligence

Prof. Björn Ommer
HCI & IWR

Outline – Constraint Satisfaction Problems

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

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: Map-Coloring

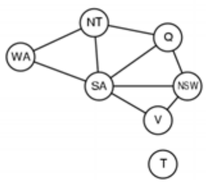
- Variables *WA, NT, Q, NSW, V, SA, T* Tasmania
- Domains D_i = {red,green,blue}
- Constraints: adjacent regions must have different colors
  - e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

Example: Map-Coloring

- Solutions are complete and consistent assignments (satisfying all constraints),
  - 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



- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

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., $StartJob_1 + 5 \leq StartJob_3$
    - linear constraints solvable, nonlinear undecidable
- 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., constraints from large neighborhoods
- Preferences (soft constraints), e.g., red is better than green often representable by a cost for each variable assignment

  ⇨constrained optimization problems (c.f. simplex)

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

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

Idea from informed search (last chapter):

- Relaxed problems
- Optimal solution cost of a relaxed problem is not greater than the optimal solution cost of the real problem
- ⇨ cost of solving relaxed problem is an admissible heuristic for original problem
- Solve relax problem / then try to fix relaxed problem

Standard search formulation (incremental)

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 (not fixable)
- Goal test: the current assignment is complete

This is the same for all CSPs:

- Every solution appears at depth *n* with *n* variables
  → use depth-first search
- Path is irrelevant, so can also use complete-state formulation
- $b = (n - l) \cdot d$ at depth l , hence $n! \cdot d^n$ leaves!!

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
  →simplifies to b = d and number of leaves = d^n
- 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* ≈ 25

Solutions for n queens problem:

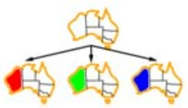
n:	1	2	3	4	5	6	7	8	9	10	...	24	25	26	27
fundamental:	1	0	0	1	2	1	6	12	46	92	...	28,439,272,956,934	275,986,683,743,434	2,789,712,466,510,289	29,363,791,967,678,199
all:	1	0	0	2	10	4	40	92	352	724	...	227,514,171,973,736	2,207,893,435,808,352	22,317,699,616,364,044	234,907,967,154,122,528

Backtracking search

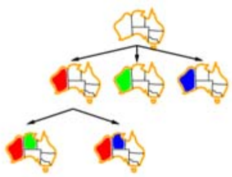
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```


Backtracking example

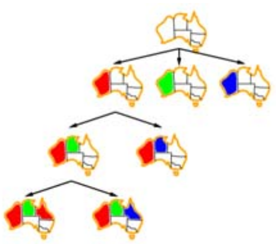
Backtracking example



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?
  - Can we take advantage of problem structure?

Most constrained variable

- **Most constrained variable:**
  choose the variable with the fewest legal values

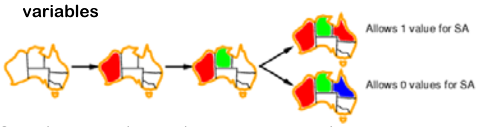
- a.k.a. minimum remaining values (MRV) heuristic

Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
- choose the variable with the most constraints on remaining variables

Least constraining value

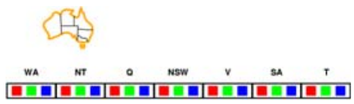
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables



- Combining these heuristics makes 1000 queens feasible

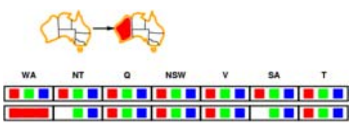
Forward checking

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



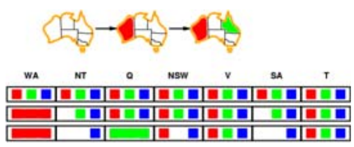
Forward checking

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



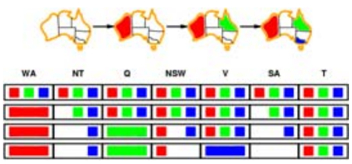
Forward checking

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



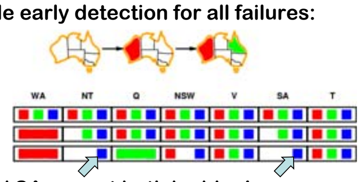
Forward checking

- **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:

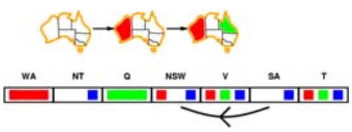


- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

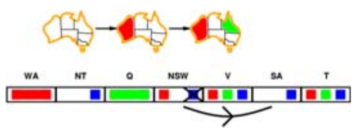
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y



Arc consistency

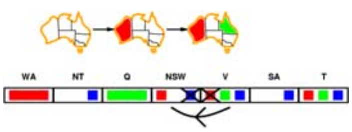
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
- for every value x of X there is some allowed y



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

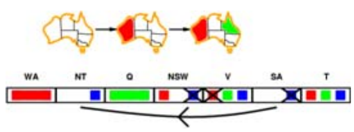


- If X loses a value, neighbors of X need to be rechecked

Arc consistency

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

  for every value x of X there is some allowed y



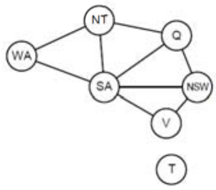
- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

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 REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue

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

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

Problem Structure

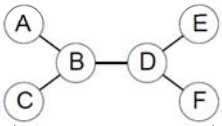


- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph

Problem Structure

- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $n/c \cdot d^c$, linear in n
- E.g., n=80, d=2, c=20
  - 2^{80} = 4 billion years at 10 million nodes/sec brute force vs.
  - $4 \cdot 2^{20}$ = 0.4 seconds at 10 million nodes/sec

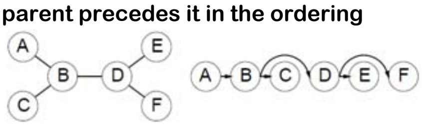
Tree-structured CSPs



- 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

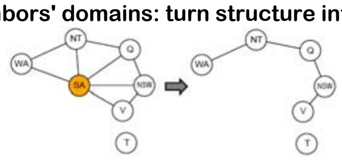
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



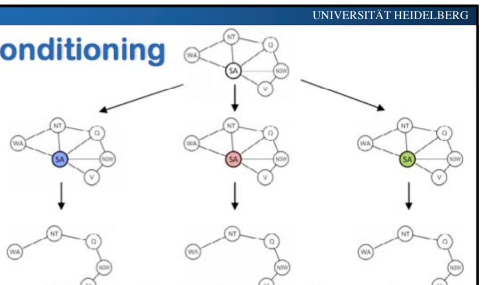
2. For j from n down to 2, apply RemoveInconsistent(Parent(X_j);X_j)
3. For j from 1 to n, assign X_j consistently with Parent(X_j)

Nearly tree-structured CSPs

- Conditioning: instantiate a variable, prune its neighbors' domains: turn structure into tree



Cutset Conditioning



- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Remove the values that conflict with the cutset assignment
- Solve the resulting tree structured CSPs

Cutset Conditioning

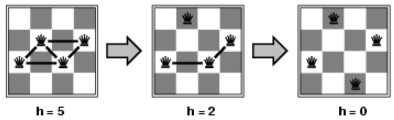
- Cutset size c ⇨ runtime O(d^c (n - c)d^2), very fast for small c

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

Example: 4-Queens

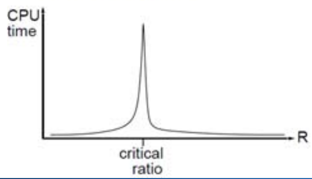
- States: 4 queens in 4 columns (4^4 = 256 states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: *h(n)* = number of attacks



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

Performance of min-conflicts

- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)
- The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio $R = \frac{\text{number of constraints}}{\text{number of variables}}$



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
- 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