Problem Solving: CSP Search

References

Outline

- CSP examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs
Recall the standard search problems

- state is a “black box'', i.e. any data structure that supports:
  - INITIAL-STATE
  - GOAL-TEST?
  - SUCCESSORS
  - STEP-COST
Constraint Satisfaction Problems (CSPs)

- CSP search problem
  - a state is defined by a set of variables $X_k$, each with values from a domain $D_k$
  - a goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Allows useful general-purpose algorithms with more power than standard search algorithms
Example 1: 4-Queens

- **States:** 4 queens in 4 columns (4^4 = 256 states)
- **Operators:** move queen in column
- **Goal test:** no attacks
- **Evaluation:** \( h(n) = \) number of attacks
Example 2: cryptarithmetic

- **Variables:** \{ F, T, U, W, R, O, X1, X2, X3 \}
- **Domains:** \{0,1,2,3,4,5,6,7,8,9\}
- **Constraints:**
  - F, T, U, W, R, O must be all different
  - O + O = R + 10 * X1
  - etc.
Example 3: map coloring

**CSP-oriented description:**

- **Variables:** { WA, NT, Q, NSW, V, SA, T }
- **Domains:** for each variable, $D = \{ \text{red, green, blue} \}$
- **Constraints:** adjacent regions must have different colors
Example 3: map coloring (a solution)

CSP-oriented description:

- Solutions are assignments satisfying all constraints, e.g., {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}
Constraint graphs

- Binary CSPs: each constraint relates at most two variables
- Constraint graph for binary CSPs: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search (e.g., Tasmania is an independent subproblem!)
Variety of CSPs: Variables

- **Variables:**
  - **discrete variables for finite domains:**
    - being \( n \) the number of variables, each with a domain of size \( d \), there are \( O(d^n) \) complete assignments
    - when \( d = 2 \) we have Boolean CSPs (the satisfiability problem for Boolean CSPs is NP-complete)
  - **discrete variables for infinite domains:**
    - (integers, strings, etc.), e.g., job scheduling, variables are start/end days for each job. In this case, a constraint language is needed, e.g., \( \text{StartJob1} + 5 \leq \text{StartJob3} \)
  - **continuous variables**
    - e.g., start/end times for Hubble Telescope observations
Variety of CSPs: Constraints

- **Linear / nonlinear constraints:**
  - linear constraints are solvable in polynomial time by LP methods
  - nonlinear constraints are undecidable

- **Unary / binary / higher-order 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
Variety of CSPs: Soft constraints

- **Soft constraints:**
  - Preferences are constraints that should be satisfied, although a solution may also disregard some of them, (e.g., red is better than green often representable by a cost for each variable assignment)
  - Used in constrained optimization problems
Real world CSP problems

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floor planning

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

- **States:**
  - a state is defined by the values assigned so far

- **Initial state:**
  - the empty assignment, i.e. \{\}

- **Successors function:**
  - assign a value to an unassigned variable that does not conflict with current assignment, or fail if no legal assignments are feasible

- **Goal test:**
  - the current assignment is complete
Search formulation

- Note that, for any CSP:
  - Every (partial) solution appears at depth $n$ with $n$ variables: use depth-first search
  - In principle:
    - the branching factor at depth $k$ is: $b = (n-k) \cdot d$
    - the number of leaves is: $n! \cdot d^n$
  - But ... variable assignments are commutative, e.g.:
    - $[WA=red, NT=green] \equiv [NT=green, WA=red]$
Backtracking search

- Hence, we only need to consider assignments to a single variable at each node
  - the branching factor at any depth is: $b = d$
  - the number of leaves is: $d^n$

- Depth-first search for CSPs with single-variable assignments (and failure check) is called backtracking search

- Backtracking search is the basic uninformed algorithm for CSPs
  - e.g. it can solve the n-queens problem for $n \approx 25$
Backtracking search algorithm

- We need a dictionary of <key, values> pairs, in which key is in fact a variable and values contains the set of values that -so far- have been found compatible with the constraints imposed by the CSP problem.
Backtracking search algorithm

```lisp
defun BACKTRACKING-SEARCH(csp) returns solution
  return RECURSIVE-BT ( csp, {} )

defun RECURSIVE-BT(csp, assigned) returns solution
  if problem.GOAL-TEST?(assigned) then return assigned
  uvar ← csp.SELECT-VAR(VARS(csp), assigned)
  for each value in csp.ORDER-DOMAIN-VALS(uvar, assigned) do
    if CONSTRAINTS(csp).ARE-SATISFIED-BY?(uvar, value) then
      assigned[uvar].add(value)
      result ← RECURSIVE-BT(csp, assigned)
      if result <> *FAILURE* then return result
  return *FAILURE*
```
Backtracking example
Backtracking example
Backtracking example
Improving backtracking efficiency

- Decisions:
  - 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?
Improving backtracking efficiency

- General purpose heuristics - choose among:
  - the most-constrained variables (i.e., vars with the fewest legal values)
  - the most-constraining variables (i.e., vars that affect the highest number of other variables)
  - the least-constraining value (i.e., vars that affect as less as possible the values of other variables)

- With the above heuristics combined, the 1000-queens problem becomes feasible!
Most-constrained variables

- Choose (among) the variable(s) with the fewest legal values
Most-constraining variables

- To break ties among the most constrained variables choose the variable with the most constraints on remaining variables

0/T, 1/-, 2/WA+V, 3/NT+Q+NSW, 5/SA
Least-constraining value

Given a variable, choose its least constraining value, i.e. the one that rules out the fewest values in the remaining variables.
Forward checking

- Idea: Keep track of the remaining legal values for unassigned variables, and terminate the search (with failure) when a variable with no selectable legal values exists.
Forward checking
Forward checking
Forward checking
Forward checking
Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures
- Constraint propagation repeatedly enforces constraints locally
Constraint propagation

- E.g. NT and SA cannot both be blue!
Arc consistency

- The simplest form of propagation makes each arc consistent:

\[ X \rightarrow Y \text{ is consistent iff for every value } x \text{ of } X \]
\[ \text{there is some allowed } y \text{ for } Y \]
Arc consistency

\[ SA \rightarrow NSW \]
Arc consistency

- NSW \not\leftrightarrow SA
Arc consistency

- If X (e.g. NSW) loses a value, neighbors of X need to be rechecked
Arc consistency

- Arc consistency detects failures earlier than forward checking
- It can be run as a preprocessor or after each assignment
Arc consistency algorithm

class Constraint-Satisfaction-Problem
def method ARC-CONSISTENCY() --- modifies the CSP instance
    --- tries to reduce the domains of csp
    q ← csp.ARCS() --- a queue of arcs, initially all the arcs in csp
    while not IS-EMPTY?(q) do
        <X_i, X_j> ← q.get()
        if csp.REMOVE-INCONSISTENT(X_i, X_j) then
            for each X_k in csp.NEIGHBORS(X_i) do q.put ( <X_k, X_i> )
Arc consistency algorithm

class Constraint-Satisfaction-Problem
def method REMOVE-INCONSISTENT(X, Y) returns boolean
    --- returns true iff we modify the domain of X
    removed ← false
    for each x in DOMAIN(X) do
        if [ no y in DOMAIN(Y) allows <x,y> to satisfy “X → Y” ] then
            DOMAIN(X).delete(x)
            removed ← true
    return removed

O(n²d³) can be reduced to O(n²d²) but cannot detect all failures in polynomial time!
Problem structure

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

- Suppose that the problem has $n$ variables and that each subproblem has $c$ variables.
- Worst-case solution cost is $d^c \cdot \frac{n}{c}$, which is linear in $n$.
- E.g., $n=80$, $d=2$, $c=20$
  
  $2^{80} = 4$ billion years at 10 million nodes/sec
  $4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec
Tree-structured CSPs

- If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time
  - Compare to general CSPs, where the 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, and order variables from root to leaves such that every node's parent precedes it in the ordering ]

2. for j in [n .. 2] do
   csp.REMOVE-INCONSISTENT(PARENT(X_j), X_j)

3. for j do [1 .. n] do
   [ assign X_j consistently with PARENT(X_j) ]
Nearly tree-structured CSPs

- **Conditioning**: instantiate a variable, and prune its neighbors' domains

- **Cutset conditioning**: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

- **Cutset size c**: runtime $O(d^c \cdot (n-c)d^2)$, very fast for small $c$
Iterative algorithms for CSPs

- Hill-climbing and 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., hillclimb with \( h(n) = \) total number of violated constraints
Example: 4-Queens

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

- Also works in hard problems, e.g.:
  - schedule observations for the Hubble Space Telescope (good to repair schedules with minimum number of changes)
  - airline schedule when weather renders it unfeasible
Summary

- CSPs are a special kind of problem:
  - states are defined by values of a fixed set of variables
  - the goal test is defined by constraints on variable values
Summary (cont)

- **Speed up:**
  - 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
Summary (cont)

- CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice