Artificial Intelligence

Constraint Satisfaction Problem

Ch. 6
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

• Allows useful **general-purpose** algorithms with more power than standard search algorithms
Example: Map-Coloring

- **Variables** $WA, NT, Q, NSW, V, SA, T$
- **Domains** $D_i = \{\text{red, green, blue}\}$
- **Constraints**: adjacent regions must have different colors
  e.g., $WA \neq NT$, 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, 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
Example: 8-queens problem

- Variables: Queens
- Domains: positions
- Constraints: no queens attack each other
Varieties of CSPs

• Discrete variables
  – finite domains:
    » $n$ variables, domain size $d \rightarrow O(d^n)$ 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., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$

• 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
Example: Cryptarithmetic

• **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* $(F,T,U,W,R,O)$
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F$, $T \neq 0$, $F \neq 0$
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
• 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
• This is the same for all CSPs
  1. Every solution appears at depth \(n\) with \(n\) variables
    → use depth-first search
  2. Path is irrelevant, so can also use complete-state formulation
  3. \(b = (n - \ell)d\) at depth \(\ell\) hence \(n! \cdot d^n\) leaves
Backtracking search

- Variable assignments are commutative, i.e.,
  \[ \text{WA} = \text{red then NT} = \text{green} \] same as \[ \text{NT} = \text{green then WA} = \text{red} \]

- 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 example
Backtracking example
Backtracking example
Backtracking example
Improving backtracking efficiency

• **General-purpose** methods can give huge gains in CSP speed:
  – Which variable should be assigned next?
  – In what order should its values be tried?
  – Can we detect inevitable failure early?
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

<table>
<thead>
<tr>
<th></th>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Forward checking

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

- In many optimization problems, path is irrelevant; the goal state itself is the solution
- Then state space = set of “complete” configurations; find optimal configuration, e.g., TSP
  or, find configuration satisfying constraints, e.g., timetable
- In such cases, can use iterative improvement algorithms, i.e., keep a single “current” state, try to improve it
- Constant space, suitable for online as well as offline search
Example: Traveling Salesman Problem

• Start with any complete tour, perform pairwise exchanges

• Variants of this approach get within 1% of optimal very quickly with thousands of cities
Example: 4-Queens

• Put \( n \) queens on an \( n \times n \) board with no two queens on the same row, column, or diagonal

• Move a queen to reduce number of conflicts

• Almost always solves \( n \)-queens problems almost instantaneously for very large \( n \), e.g., \( n=1 \) million
**Hill-climbing search**

- "Like climbing Everest in thick fog with amnesia"

```plaintext
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
current ← neighbor
end
```
Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima
- Random-restart hill climbing overcomes local maxima -- trivially complete
- Random sideways moves escape from shoulders loop on at maxima
Hill-climbing search: 8-queens problem

- \( h = \) number of pairs of queens that are attacking each other, either directly or indirectly
- \( h = 17 \) for the above state
Hill-climbing search: 8-queens problem

- A local minimum with $h = 1$
Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
             schedule, a mapping from time to "temperature"
    local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps
    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] − VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability \( e^{\Delta E / T} \)
```
Properties of simulated annealing search

• One can prove: If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

• Widely used in VLSI layout, airline scheduling, etc
Local beam search

- Idea:
  - Keep track of $k$ states rather than just one
  - Start with $k$ randomly generated states
  - At each iteration, all the successors of all $k$ states are generated
  - If any one is a goal state, stop; else select the $k$ best successors from the complete list and repeat.

- Not the same as $k$ searches run in parallel!
  Searches that find good states recruit other searches to join them
- Problem: quite often, all $k$ states end up on same local hill
- Idea: choose $k$ successors randomly, biased towards good ones
- Observe the close analogy to natural selection!
Genetic algorithms

• = stochastic local beam search + generate successors from pairs of states

• Idea:
  – Start with $k$ randomly generated states (population)
  – A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
  – Evaluation function (fitness function). Higher values for better states.
  – Produce the next generation of states by selection, crossover, and mutation
Genetic algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$ etc
Genetic algorithms

Crossover helps iff substrings are meaningful components