Constraint Satisfaction Problems

Chapter 6, Sections 1–5
Outline

◇ CSP examples
◇ Backtracking search for CSPs
◇ Problem structure and problem decomposition
◇ Local search for CSPs
Constraint satisfaction problems (CSPs)

Standard search problem:
- the state is a “black box”—any old data structure that supports goal test, eval, successor

CSP is a more specific search problem:
- the state is defined by variables $X_i$ with values from domain $D_i$
- the 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: \textit{WA, NT, Q, NSW, V, SA, T}

Domains: \(D_i = \{\text{red, green, blue}\}\)

Constraints: adjacent regions must have different colors

\(\text{e.g., } \text{WA} \neq \text{NT}, \text{WA} \neq \text{SA}, \text{NT} \neq \text{SA}, \text{NT} \neq Q, \text{SA} \neq Q, \ldots\)
Solutions are assignments satisfying all constraints, e.g.,
\[
\{ \text{WA = red, NT = green, Q = red, NSW = green, } \\
V = \text{red, SA = blue, T = green} \}
\]
Binary CSP: each constraint relates at most two variables

Constraint graph: 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!
Varieties of CSPs

Discrete variables

◊ finite domains; 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., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$
  – linear constraints solvable, nonlinear undecidable

Continuous variables

◊ e.g., start/end times for Hubble Telescope observations
◊ linear constraints solvable in polynomial time by LP methods
Varieties of constraints

**Unary** constraints involve a single variable,  
e.g., $SA \neq \text{green}$

**Binary** constraints involve pairs of variables,  
e.g., $SA \neq WA$

**Higher-order** constraints involve 3 or more variables,  
e.g., cryptarithmetic puzzles

**Preferences** (soft constraints), e.g., *red* is better than *green*  
often representable by a cost for each variable assignment  
→ constrained optimization problems
Example: Cryptarithmetic puzzle

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

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.
Real-world CSPs

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

Floorplanning
Standard search formulation (incremental)

Let’s start with the straightforward, dumb 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.
  \[\Rightarrow\] fail if there are 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
  \[\Rightarrow\] use depth-first search
3) The path is irrelevant, so we can also use a complete-state formulation
4) \(b = (n - \ell)d\) at depth \(\ell\), where \(d\) is the domain size
  \[\Rightarrow\] hence there are \(n!\ d^n\) leaves!!!! 😁
Backtracking search

Variable assignments are commutative, i.e.,

\[
\begin{align*}
\text{[first } & WA = \text{red then } NT = \text{green}] \quad \text{is the same as} \\
\text{[first } & NT = \text{green then } WA = \text{red}] 
\end{align*}
\]

We only need to consider assignments to a single variable at each node

\[ b = d, \] so there are \( d^n \) leaves  \( \text{ (instead of } n!d^n) \)

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 \( n \approx 25 \)
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:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?
Minimum remaining values (MRV):
- choose the variable with the fewest legal values
Degree heuristic

If there are several MRV variables, we can use the **degree heuristic**:

– choose the variable with the most constraints on remaining variables
Least constraining value

When we have selected a variable using MRV and degree heuristic, we choose the least constraining value:
– the one that rules out the fewest values in the remaining variables

Combining these heuristics makes $n$-queens feasible for $n \approx 1000$
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 \( y \)

[Diagram of Australia with states WA, NT, Q, NSW, V, SA, T, showing consistency propagation]
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
Problem structure

Tasmania and mainland are independent subproblems — identifiable as connected components of the constraint graph.

Suppose that each subproblem has $c$ variables out of $n$ total.

The worst-case solution cost is $\frac{n}{c} \cdot d^c$, which is linear in $n$.

E.g., $n = 80$, $d = 2$, $c = 20$:

$2^{80} = 4$ billion years at 10 million nodes/sec

if we divide it into 4 equal-size subproblems:

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec
Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- we allow states with unsatisfied constraints
- the operators \textbf{reassign} variable values

The \textbf{min-conflicts algorithm}:
Variable selection:
- randomly select any conflicted variable
Value selection by the \textbf{min-conflicts} heuristic:
- choose the 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 \( (4^4 = 256 \text{ states}) \)

**Operators:** move queen in column

**Goal test:** no attacks

**Evaluation:** \( h(n) = \text{number of attacks} \)
Performance of min-conflicts

Given a random initial state, we 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 are defined by values of a fixed set of variables
- goal test is 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