Constraint satisfaction problems (CSPs)

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

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

Outline

- CSP examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs

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., $WA \neq NT$ (if the language allows this), or $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), \ldots\}$
Example: Map-Coloring contd.

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

Constraint graph

Binary CSP: each constraint relates at most two variables
Constraint graph: nodes are variables, arcs show constraints

Varieites 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 poly time by LP methods

Varieites 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 column constraints

Preferences (soft constraints), e.g., \text{red} is better than \text{green} often representable by a cost for each variable assignment
\(\rightarrow\) constrained optimization problems
Example: Cryptarithmetic

\[
\begin{array}{c}
T \ W \ O \\
+ \ T \ W \ O \\
\hline
F \ U \ R \\
\end{array}
\]

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
- \( \text{alldiff}(F, T, U, W, R, O) \)
- \( O + O = R + 10 \cdot X_1 \)

Real-world CSPs

Assignment problems
- e.g., who teaches what class, who flies which flight

Timetabling problems
- e.g., which class is offered when and where, which flight is scheduled when and where

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

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

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 no legal assignments (not fixable!)
- **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 \((d \text{ values each})\)
   - \( \Rightarrow \) use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) \( b = (n - \ell)d \) at depth \( \ell \), hence \( n!d^n \) leaves!!!!

Backtracking search

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

Order of the variable assignments is not important, pick an arbitrary order

Consider assignments to a different variable at each level (according to the order)
- \( \Rightarrow b = d \) 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 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
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?

Choosing a variable: Minimum remaining values

Minimum remaining values (MRV):
choose the variable with the fewest legal values

Choosing a variable: Degree heuristic

Tie-breaker among MRV variables
Degree heuristic:
choose the variable with the most constraints on remaining variables (highest degree)
Choosing a value: Least constraining value

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

- Allows 1 value for SA
- Allows 0 values for SA

Combining these heuristics (most-constraining variables, least-constraining values) makes 1000 queens feasible

Forward checking (1-step look ahead)

- Keep track of remaining legal values for unassigned variables
  - Help MRV
  - Terminate search when any variable has no legal values
**Forward checking (1-step look ahead)**

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

**Arc consistency (multi-step look ahead)**

Simplest form of propagation makes each arc consistent

\[ X \rightarrow Y \] is consistent iff

for every value \( x \) of \( X \) there is some allowed \( y \)

\[ NT \] and \( SA \) cannot both be blue!

Constraint propagation repeatedly enforces constraints locally
Arc consistency (multi-step look ahead)

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

function \( \text{AC-3}(csp) \) returns the CSP, possibly with reduced domains

inputs: \( csp \), a binary CSP with variables \( \{X_1, X_2, \ldots, X_n\} \)

local variables: \( queue \), a queue of arcs, initially all the arcs in \( csp \)

while \( queue \) is not empty do

(\( X_i, X_j \)) ← Remove-First(\( queue \))

if Remove-Inconsistent-Values(\( X_i, X_j \)) then

for each \( X_k \) in Neighbors[\( X_i \]) do

add (\( X_k, X_i \)) to \( queue \)

function Remove-Inconsistent-Values(\( X_i, X_j \)) returns true iff succeeds

\( removed ← false \)

for each \( x \) in Domain[\( X_i \]) do

if no value \( y \) in Domain[\( X_j \)] allows (\( x, y \)) to satisfy the constraint \( X_i \leftrightarrow X_j \) then

delete \( x \) from Domain[\( X_i \)]; \( removed ← true \)

return \( removed \)

\( O(n^2d^3): n^2 \) arcs, \( d \) enqueue's, \( d^2 \) pairs of values to check [skip p.147-9]

Iterative algorithms 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., 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: \(h(n) = \text{number of attacks}\)

\[
\begin{align*}
\text{h = 5} & \quad \rightarrow & \quad \text{h = 2} & \quad \rightarrow & \quad \text{h = 0}
\end{align*}
\]

MIN-CONFLICTS Algorithm

\[
\begin{align*}
\text{function Min-Conflicts}(csp, \text{max-steps}) & \quad \text{returns a solution or failure} \\
\text{inputs: csp, a constraint satisfaction problem} & \quad \text{max-steps, the number of steps allowed before giving up} \\
\text{local variables: current, a complete assignment} & \quad \text{var, a variable} \\
& \quad \text{value, a value for a variable} \\
\text{current} & \leftarrow \text{an initial complete assignment for csp} \\
\text{for} \ i = 1 \ \text{to} \ \text{max-steps} \ \text{do} \\
& \quad \text{var} \leftarrow \text{a randomly chosen, conflicted variable from Variables[csp]} \\
& \quad \text{value} \leftarrow \text{the value} \ v \ \text{for} \ \text{var} \ \text{that minimizes Conflict}(\text{var}, \text{v}, \text{current}, \text{csp}) \\
& \quad \text{set} \ \text{var} = \text{value} \ \text{in current} \\
& \quad \text{if} \ \text{current} \ \text{is a solution for csp} \ \text{then return current} \\
\text{end} \\
\text{return failure}
\end{align*}
\]

Example: 8-Queens Min conflicts

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 \(R\):

\[
R = \frac{\text{number of constraints}}{\text{number of variables}}
\]

CPU time

critical ratio

R
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