Constraint Satisfaction Problems

Sections 3.7 and 4.4, Chapter 5 of AIMA2e

Variables WA, NT, Q, NSW, V, SA, T
Domains Di = {red, green, blue}
Constraints: adjacent regions must have different colors
e.g., WA ≠ NT (if the language allows this), or
(WA, NT) ∈ \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}

Outline

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

Example: Map-Coloring contd.

Solution: are assignments satisfying all constraints, e.g.,
\{WA—red, NT—green, Q—red, NSW—green, V—red, SA—blue, T—green\}

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 with values from domain Di
  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

Constraint graph

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., \(\text{StartJob}_1 + 5 \leq \text{StartJob}_2\)
- 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

### 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**

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
  - \(\Rightarrow\) use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) \(b - (n - \ell)d\) at depth \(\ell\), hence \(\ell\)\(d^n\) leaves!!!

### Example: Cryptarithmetic

![Example Diagram]

**Variables**: \(F, T, U, W, R, O\)

**Domain**: \(\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\)

**Constraints**
- All differ \(F, T, U, W, R, O\)
- \(O + O = R + 10 \cdot X_i\), etc.

### Backtracking search

Variable assignments are commutative, i.e.,

\([WA - \text{red} \Rightarrow NT - \text{green}]\) same as \([NT - \text{green} \Rightarrow WA - \text{red}]\)

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 \(n = 25\)
Backtracking search

function BACKTRACKING-SEARCH(cp) returns solution/failure
  return RECURSIVE-BACKTRACKING([], cp)
function RECURSIVE-BACKTRACKING(assigned, cp) returns solution/failure
  if assigned is complete then return assigned
  var ← SELECT-Unassigned-Variable(Variables[cp], assigned, cp)
  for each value in ORDER-Domain-Values(var, assigned, cp) do
    if value is consistent with assigned according to Constraints[cp] then
      result ← RECURSIVE-BACKTRACKING(var ← value|assigned, cp)
      if result ≠ failure then return result
  end
  return failure

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?
**Most constrained variable**

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

**Forward checking**

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

---

**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 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

---

**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 $y$

---

**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 $y$

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 AC(CSP) returns CSP, possibly with reduced domains
local variables: queue, a queue of arcs, initially all arcs in CSP
loop while queue is not empty do
\( (X_i, X_j) \leftarrow \text{remove-front(queue)} \)
if Remove-Inconsistent(\( X_i, X_j \)) then
for each \( X_k \) in Neighbors[\( X_j \)] do
add (\( X_k, X_j \)) to queue

function Remove-Inconsistent(\( X_i, X_j \)) returns true if we remove a value
loop for each \( x \) in Domain[\( X_j \)] do
if \( (a[x]) \) satisfies the constraint for some value \( y \) in Domain[\( X_i \)]
then delete \( x \) from Domain[\( X_j \)]; \( \text{removed} \leftarrow \text{true} \)
return \( \text{removed} \)

\( O(n^2d^3) \), can be reduced to \( O(n^2d^2) \)
but cannot detect all failures in poly time!

Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in \( O(n^2d^3) \) 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 Remove-Inconsistent(\( \text{Parent}(X_j), X_j \))
3. For \( j \) from 1 to \( n \), assign \( X_j \) consistently with \( \text{Parent}(X_j) \)

Problem structure

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

Problem structure contd.

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., \( n = 30 \), \( d = 2 \), \( c = 20 \)
\( 2^{30} = 4 \text{ billion years at 10 million nodes/sec} \)
\( 4 \cdot 2^{30} = 0.4 \text{ seconds at 10 million nodes/sec} \)

Nearly tree-structured CSPs

Conditioning: instantiate a variable, 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 \( c \Rightarrow \text{runtime } O(d^c \cdot (n-c)d^2) \), very fast for small \( c \)
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

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

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice

---

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

![4-Queens example](image)

h = 5  h = 2  h = 0

---

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

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

![Performance graph](image)