Constraint satisfaction problems (CSPs)

Standard search problem:

- **state** is a “black box”—any "good 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

CSP definition

A **Constraint Satisfaction Problem** consists of three components: $X$, $D$ and $C$:

- $X$ is a set of variables, $\{X_1, \ldots, X_n\}$,
- $D$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable,
- $C$ is a set of constraints that specify allowable combinations of values.

Each constraint $C_i$ consists of a pair $<\text{scope}, \text{rel}>$.

A **solution** to a CSP is a consistent, complete assignment.
Example: 4-Queens as a CSP

Assume one queen in each column. Which row does each one go in?

Variables $Q_1, Q_2, Q_3, Q_4$

Domains $D_i = \{1, 2, 3, 4\}$

Constraints
- $Q_i \neq Q_j$ (cannot be in same row)
- $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

Translate each constraint into set of allowable values for its variables

E.g., values for $(Q_1, Q_2)$ are $(1,3) (1,4) (2,4) (3,1) (4,1) (4,2)$

Example: Map-Coloring

Variables $WA, NT, Q, NSW, V, SA, T$

Domains $D_i = \{\text{red}, \text{green}, \text{blue}\}$

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

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 poly time by LP methods

**Varieties of constraints**

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

- **Binary constraints** involve pairs of variables,
  - e.g., $\text{S} \neq \text{W}$

- **Higher-order constraints** involve 3 or more variables,
  - e.g., cryptarithmetic column constraints,
    - Sometimes called (misleadingly) **global** constraints

- **Preferences** (soft constraints), e.g., red is better than green
  - Often representable by a cost for each variable assignment
    → Constrained optimization problems

**Example: Cryptarithmetic**

```
T W O
+ T W O
---
F O U R
```

- **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**
  - $\text{alldiff}(F, T, U, W, R, O)$
  - $O + O = R + 10 \cdot X_3$, 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**
- **Floor-planning**

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.
  ⇒ 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
  ⇒ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - ℓ)d$ at depth $ℓ$, hence $n!d^n$ leaves!!!! 😊

Backtracking search

Variable assignments are commutative, i.e.,

$[WA = \text{red} \text{ then } NT = \text{green}]$ same as $[NT = \text{green} \text{ then } WA = \text{red}]$

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

```plaintext
function Backtracking-Search(esp) returns solution/failure
  return Backtrack({}, esp)

function Backtrack(assignment, esp) returns solution/failure
  if assignment is complete then return assignment
  var ← Select-Unassigned-Variable(Variables[esp], assignment, esp)
  for each value in Order-Domain-Values(var, assignment, esp) do
    if value is consistent with assignment given Constraints[esp] then
      add {var = value} to assignment
      result ← Backtrack(assignment, esp)
      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?
Minimum remaining values

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

Degree heuristic

Tie-breaker among MRV variables
Degree heuristic: 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

Forward checking

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

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

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
**Node consistency**

Simplest form of propagation: makes each node node-consistent

Node \( X \) is node-consistent iff
  for every value \( x \) of \( X \) all the unary constraints of \( X \) are satisfied

Needs to be run only once.

---

**Arc consistency**

This form of propagation makes each arc consistent

\( X \to Y \) 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

This 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.

Global constraints

Involve an arbitrary number of variables, but not necessarily all.

- \text{alldiff}
- \text{atmost}, e.g. \text{atmost}(10, X_1, X_2, X_3, X_4)
- \text{diff2}, e.g. \text{diff2}([x_1, y_1, dx_1, dy_1], [x_2, y_2, dx_2, dy_2], \ldots)
- \text{cumulative} (scheduling),
- bounds propagation and bounds consistency

Instead of \( \{v_1, v_2, \ldots, v_n\} \) we deal with \( [v_1, v_n] \).

Arc consistency algorithm

\begin{function}
function \text{AC-3}(\text{csp}) \text{ returns } \text{the CSP, possibly with reduced domains}
inputs: \text{csp}, \text{a binary CSP with variables } \{X_1, X_2, \ldots, X_n\}
local variables: \text{queue}, \text{a queue of arcs, initially all the arcs in } \text{csp}
while \text{queue} is not empty do
\begin{enumerate}
\item \( (X_i, X_j) \leftarrow \text{Remove-First}(\text{queue}) \)
\item if \text{Remove-Inconsistent-Values}(X_i, X_j) then
\begin{enumerate}
\item for each \( X_k \) in \text{Neighbors}[X_i] do
\begin{enumerate}
\item add \((X_k, X_i)\) to \text{queue}\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{function}

\begin{function}
function \text{Remove-Inconsistent-Values}(X_i, X_j) \text{ returns } \text{true if succeeds}
removed \leftarrow \text{false}
for each \( x \) in \text{Domain}[X_i] do
\begin{enumerate}
\item if no value \( y \) in \text{Domain}[X_j] allows \((x, y)\) to satisfy the constraint \( X_i \leftrightarrow X_j \) then delete \( x \) from \text{Domain}[X_i]; \text{removed} \leftarrow \text{true}
\end{enumerate}
return \text{removed}
\end{function}

\( O(n^2d^3) \), can be reduced to \( O(n^2d^2) \) (but detecting all is NP-hard)

Backtracking search with inference

\begin{function}
function \text{Backtracking-Search}(\text{csp}) \text{ returns } \text{solution/failure}
return \text{Backtrack}(\{\}, \text{csp})
\end{function}

\begin{function}
function \text{Backtrack}(\text{assignment, csp}) \text{ returns } \text{solution/failure}
if \text{assignment} is complete then return \text{assignment}
var \leftarrow \text{Select-Unassigned-Variable}(\text{Variables}[\text{csp}], \text{assignment}, \text{csp})
for each \text{value} in \text{Order-Domain-Values}(\text{var}, \text{assignment}, \text{csp}) do
\begin{enumerate}
\item if \text{value} is consistent with \text{assignment} given \text{Constraints}[\text{csp}] then
\begin{enumerate}
\item add \{var = \text{value}\} to \text{assignment}
\item \text{inferences} \leftarrow \text{Inference}(\text{csp, var, value})
\item add \text{inferences} to \text{assignment}
\item result \leftarrow \text{Backtrack}(\text{assignment, csp})
\item if result \neq \text{failure} then return result
\item remove \{var = \text{value}\} from \text{assignment}
\end{enumerate}
\end{enumerate}
return \text{failure}
\end{function}
Constraints programming has finally reached the masses, thousands of newspaper readers are solving their daily constraint problem (Helmut Simonis, Imperial College).

Variables: $v[i,j] :: \{1..9\}$

Constraints:
// Rows
$v[1,1] \neq v[1,2],...$
// Columns
$v[1,1] \neq v[2,1],...$
// Squares
$v[1,1] \neq v[2,2],...$

First row, simple consistency check:
2
6
\{1, 8..9\}
3
\{4..5, 7..9\}
\{5, 7, 9\}
\{1, 5, 8..9\}
\{5, 8..9\}
\{5, 8..9\}

Note rows 3, 7, 8, 9!
First row, more advanced consistency check:

\[
\begin{array}{ccc}
2 & 6 & 3 \\
5 & 1 & 4 \\
3 & 4 & 7 \\
\end{array}
\]

\{1, 8..9\}
3
4
7
\{1, 5, 8..9\}
\{5, 8..9\}
\{5, 8..9\}
all distinct

In MiniZinc:

```mzn
include "globals.mzn";
array [1..9,1..9] of var 1..9: v;
predicate row_diff(int: r) = all_different ([v[r,c] | c in 1..9]);
predicate col_diff(int: c) = all_different ([v[r,c] | r in 1..9]);
predicate subgrid_diff(int: r, int: c) = all_different ([v[r+i,c+j] | i,j in 0..2]);
constraint forall (r in 1..9) (row_diff(r));
constraint forall (c in 1..9) (col_diff(c));
constraint forall (r,c in {1,4,7}) (subgrid_diff(r,c));
solve satisfy;
output ["v = ", show(v), 
"
"];
```

```
2 6 3 7 1 4 5 2 1
5 1 4 2 6 8 3 4 2
3 4 7 9 1 5 8 5 3
```

Problem structure

Victoria WA
NT
Q
SA
NSW
V
T

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 \( n/c \cdot d \), linear in \( n \)
E.g., \( n = 80, d = 2, c = 20 \)
\[
2^{80} = 4 \text{ billion years at } 10 \text{ million nodes/sec}
4 \cdot 2^{20} = 0.4 \text{ seconds at } 10 \text{ million nodes/sec}
\]
Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ 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 RemoveInconsistent($\text{Parent}(X_j), X_j$).

3. For $j$ from 1 to $n$, assign $X_j$ consistently with $\text{Parent}(X_j)$.

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 $c \Rightarrow$ runtime $O(d^c \cdot (n - c) d^2)$, very fast for small $c$.

Local Search, or 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 $\text{reassign}$ variable values

Variable selection: randomly select any conflicted variable.

Value selection by $\text{min-conflicts}$ 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., $n = 10,000,000$)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio $R = \frac{\text{number of constraints}}{\text{number of variables}}$

CPU time

---

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

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

But: in the worst case search will be exponentially complex anyway!