Constraint Satisfaction Problems

Chapter 5
Outline

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

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

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., $StartJob_1 + 5 \leq StartJob_3$
  - 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., \( SA \neq \text{green} \)

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

**Higher-order** constraints involve 3 or more variables

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

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

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

$O + O = R + 10 \cdot X_1$, 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

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables
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.
  \[ \Rightarrow \text{ fail if no legal assignments (not fixable!)} \]

♦ **Goal test:** the current assignment is complete

1) This is the same for all CSPs! 😊
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) Can we use depth-first search?
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
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) $b =$ ?
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) \( b = (n - \ell)d \) at depth \( \ell \), hence \( n!d^m \) leaves!!!! 😞
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) \( b = (n - \ell)d \) at depth \( \ell \), hence \( n!d^n \) leaves!!!! 😞

4) Path is irrelevant, so can also use complete-state formulation
Backtracking search

Variable assignments are **commutative**, i.e.,

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

Only need to consider assignments to a single variable at each node

\[ \Rightarrow \ b = d \text{ and there are } d^n \text{ leaves} \]

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 \( n \approx 25 \)
function Backtracking-Search(\textit{csp}) \textbf{returns} solution/failure
    return Recursive-Backtracking({}, \textit{csp})

function Recursive-Backtracking(assignment, \textit{csp}) \textbf{returns} soln/failure
    if assignment is complete then return assignment
    \textit{var} ← Select-Unassigned-Variable(Variables[\textit{csp}], assignment, \textit{csp})
    for each \textit{value} in Order-Domain-Values(\textit{var}, assignment, \textit{csp}) do
        if \textit{value} is consistent with assignment given Constraints[\textit{csp}] then
            add \{\textit{var} = \textit{value}\} to assignment
            \textit{result} ← Recursive-Backtracking(assignment, \textit{csp})
            if result \neq failure then return result
            remove \{\textit{var} = \textit{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

Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables
Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables

Seems simple (and is), but is still best method for k-colouring.
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

![Diagram of Australia with states WA, NT, Q, NSW, V, SA, T colored differently]
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:

\[ \text{WA} \quad \text{NT} \quad \text{Q} \quad \text{NSW} \quad \text{V} \quad \text{SA} \quad \text{T} \]

\[ \begin{array}{cccccccc}
\text{WA} & \text{NT} & \text{Q} & \text{NSW} & \text{V} & \text{SA} & \text{T} \\
\text{WA} & \text{NT} & \text{Q} & \text{NSW} & \text{V} & \text{SA} & \text{T} \\
\text{WA} & \text{NT} & \text{Q} & \text{NSW} & \text{V} & \text{SA} & \text{T} \\
\end{array} \]

\text{NT} and \text{SA} cannot both be blue!

Constraint propagation repeatedly enforces constraints locally
Arc consistency

Simplest form of propagation makes each arc consistent

\[ X \rightarrow Y \text{ is consistent iff} \]

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
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-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
  \((X_i, X_j) \leftarrow \text{Remove-First}(queue)\)
  if \text{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 \text{Remove-Inconsistent-Values}(X_i, X_j) returns true iff succeeds
removed \leftarrow 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 \leftarrow true
return removed

Complexity?
Arc consistency algorithm

**function 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**

\[(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})\]

**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

\[\text{removed} \leftarrow \text{false}\]

**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]; \text{removed} \leftarrow \text{true}

**return** removed

\[O(n^2d^3), \text{ can be reduced to } O(n^2d^2) \text{ (but detecting all is NP-hard)}\]
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 \( \frac{n}{c} \cdot d^c \), \textbf{linear} in \( n \).

E.g., \( n = 80, \ d = 2, \ c = 20 \)

\[
2^{80} = 4 \text{ billion years at 10 million nodes/sec}
\]

\[
4 \cdot 2^{20} = 0.4 \text{ seconds at 10 million nodes/sec}
\]
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