Constraint Satisfaction Problems

CHAPTER 3, SECTION 7 AND CHAPTER 4, SECTION 4.4

Outline

◇ CSP examples
◇ General search applied to CSPs
◇ Backtracking
◇ Forward checking
◇ Heuristics for CSPs
**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 $V_i$ 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: 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 $(1, 3)$ $(1, 4)$ $(2, 4)$ $(3, 1)$ $(4, 1)$ $(4, 2)$
Constraint graph

*Binary CSP:* each constraint relates at most two variables

*Constraint graph:* nodes are variables, arcs show constraints

---

Example: Cryptarithmetic

**Variables**

\[ D \ E \ M \ N \ O \ R \ S \ Y \]

**Domains**

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

**Constraints**

\[ M \neq 0, \ S \neq 0 \ (unary \ constraints) \]

\[ Y = D + E \text{ or } Y = D + E - 10, \text{ etc.} \]

\[ D \neq E, \ D \neq M, \ D \neq N, \text{ etc.} \]
Example: Map coloring

Color a map so that no adjacent countries have the same color.

Variables
- Countries $C_i$

Domains
- $\{\text{Red, Blue, Green}\}$

Constraints
- $C_1 \neq C_2$, $C_1 \neq C_5$, etc.

Constraint graph:

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
Applying standard search

Let's start with the straightforward, dumb approach, then fix it.
States are defined by the values assigned so far.
Initial state: all variables unassigned.
Operators: assign a value to an unassigned variable.
Goal test: all variables assigned, no constraints violated.

Notice that this is the same for all CSPs!

Implementation

CSP state keeps track of which variables have values so far.
Each variable has a domain and a current value.

datatype CSP-STATE
    components: UNASSIGNED, a list of variables not yet assigned
                ASSIGNED, a list of variables that have values

datatype CSP-VAR
    components: NAME, for I/O purposes
                DOMAIN, a list of possible values
                VALUE, current value (if any)

Constraints can be represented
    explicitly as sets of allowable values, or
    implicitly by a function that tests for satisfaction of the constraint.
Standard search applied to map-coloring

Complexity of the dumb approach

Max. depth of space $m = \text{??}$

Depth of solution state $d = \text{??}$

Search algorithm to use $\text{??}$

Branching factor $b = \text{??}$

This can be improved dramatically by noting the following:

1) Order of assignment is irrelevant, hence many paths are equivalent
2) Adding assignments cannot correct a violated constraint
Complexity of the dumb approach

Max. depth of space $m = \infty$ $n$ (number of variables)

Depth of solution state $d = \infty$ $n$ (all vars assigned)

Search algorithm to use?? depth-first

Branching factor $b = \infty \sum |D_i|$ (at top of tree)

This can be improved dramatically by noting the following:

1) Order of assignment is irrelevant so many paths are equivalent
2) Adding assignments cannot correct a violated constraint

Backtracking search

Use depth-first search, but

1) fix the order of assignment, $\Rightarrow b = |D_i|$
   (can be done in the SUCCESSORS function)
2) check for constraint violations

The constraint violation check can be implemented in two ways:

1) modify SUCCESSORS to assign only values that are allowed, given the values already assigned
2) check constraints are satisfied before expanding a state

Backtracking search is the basic uninformed algorithm for CSPs

Can solve $n$-queens for $n = 15$
**Forward checking**

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

**Simplified map-coloring example:**

<table>
<thead>
<tr>
<th>RED</th>
<th>BLUE</th>
<th>GREEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>C_2</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>C_3</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>C_4</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C_5</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Can solve \(n\)-queens up to \(n \approx 30\)

---

**Heuristics for CSPs**

More intelligent decisions on
which value to choose for each variable
which variable to assign next

*Given \(C_1 = \text{Red}, C_2 = \text{Green}\)*, choose \(C_3 = ??\)

*Given \(C_1 = \text{Red}, C_2 = \text{Green}\)*, what next??

Can solve \(n\)-queens for \(n \approx 1000\)
**Heuristics for CSPs**

More intelligent decisions on
which value to choose for each variable
which variable to assign next

Given $C_1 = \text{Red}$, $C_2 = \text{Green}$, choose $C_3 = ?$?

$C_3 = \text{Green: least-constraining-value}$

Given $C_1 = \text{Red}$, $C_2 = \text{Green}$, what next??

$C_3$: most-constrained-variable

Can solve $n$-queens for $n \approx 1000$

---

**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

$\text{min-conflicts}$ heuristic:
choose value that violates the fewest constraints
i.e., hillclimb with $h(n) = \text{total number of violated constraints}$
**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

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nD^2)$ 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 complexity of reasoning.

Algorithm for tree-structured CSPs

Basic step is called filtering.

$\text{FILTER}(V_i, V_j)$
- removes values of $V_i$ that are inconsistent with ALL values of $V_j$

Filtering example:

$V_i$ $V_j$

allowed pairs: $<1,1>$ $<3,2>$ $<3,3>$

remove 2 from domain of $V_j$
Algorithm contd.

1) Order nodes breadth-first starting from any leaf:

2) For \( j = n \) to 1, apply \( \text{Filter}(V_i, V_j) \) where \( V_i \) is a parent of \( V_j \)

3) For \( j = 1 \) to \( n \), pick legal value for \( V_j \) given parent value

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
- 1) fixed variable order
- 2) only legal successors

Forward checking prevents assignments that guarantee later failure

Variable ordering and value selection heuristics help significantly

Iterative min-conflicts is usually effective in practice

Tree-structured CSPs can always be solved very efficiently