CHAPTER 3, SECTION 7 AND CHAPTER 4, SECTION 4.4

Constraint Satisfaction Problems
Outline

◊ Heuristics for CSPs
◊ Forward checking
◊ Backtracking
◊ General search applied to CSPs
◊ CSP examples
than standard search algorithms.

Allows useful general-purpose algorithms with more power.

Simple example of a formal representation language:

\[
\text{goal test is a set of constraints specifying} \]

\[
\text{state is defined by variables} Y \text{ with values from domain } D
\]

\[
\text{CSP: that supports goal test, eval, successor}
\]

\[
\text{state is a “black box” — any old data structure}
\]

Standard search problems:

\[
\text{Constraint satisfaction problems (CSPs)}
\]
Translate each constraint into set of allowable values for its variables.

\[
\begin{align*}
\mathcal{D} & = \{1, 2, 3, 4\} \\
\mathcal{D} & = \{1, 2, 3, 4\}
\end{align*}
\]

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

**Example: 4-Queens as a CSP**
Constraint graph: nodes are variables, arcs show constraints

Binary CSP: each constraint relates at most two variables
Example: Cryptarithmic

\[
\begin{align*}
\text{MONEY} + \text{MORE} & \rightarrow \text{SENDS} \\
D & \neq E \neq M \neq E \neq N \neq 0, \text{ etc.} \\
D & \neq E \neq M \neq 10, \text{ etc.} \\
\text{Constraints} & \text{ (unary constraints)} \\
0 & \neq S \neq W \\
D & \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\text{Domains} & \text{ Variables} \\
D \in \{M, N, O, R, S, Y\} \\
\end{align*}
\]
Example: Map coloring

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

Variables: Countries $C_i$

Domains: \{Red, Blue, Green\}

Constraints: $C_i \neq C_j$, $C_i \neq C_k$, etc.

Constraint graph:
Notice that many real-world problems involve real-valued variables.

Floorplanning

Factory scheduling

Transportation scheduling

Spreadsheet

Hardware configuration

e.g., which class is offered when and where?

Timetabling problems

e.g., who teaches what class

Assignment problems

Real-world CSPs
Notice that this is the same for all CSPs.

**Goal:** All variables assigned, no constraints violated

**Operators:** Assign a value to an unassigned variable

**Initial state:** All variables unassigned

States are defined by the values assigned so far

Let's start with the straightforward, dumb approach, then fix it.
Implicitly by a function that tests for satisfaction of the constraint

explicitly as sets of allowable values, or

Constraints can be represented

```
VALUE, current value (if any)
DOMAIN, a list of possible values
NAME, for I/O purposes
```

```
components: CSP-VAR
```

datatype CSP-VAR

```
ASSIGNED, a list of variables that have values
UNASSIGNED, a list of variables not yet assigned
```

datatype CSP-STATE

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

Implementation
Adding assignments cannot correct a violated constraint.

1) Order of assignment is irrelevant, hence many paths are equivalent.

2) This can be improved dramatically by noting the following:

\[ \text{Branching factor} = q \]

\[ \text{Search algorithm to use?} \]

\[ \text{Depth of solution state} = p \]

\[ \text{Max. depth of space} = m \]
Adding assignments cannot correct a violated constraint
Order of assignment is irrelevant so many paths are equivalent

This can be improved dramatically by noticing the following:

Benching factor $q = \frac{|D'|}{|D|}$ (at top of tree)

Search algorithm to use? depth-first

Depth of solution state $p$ (all vars assigned)

Max. depth of space $m$ (number of variables)

Complexity of the dumb approach
Backtracking search

Use depth-first search, but
1) fix the order of assignment, \( \Rightarrow b = |D_i| \)
   (can be done in the \texttt{SUCCESSORS} function)
2) check for constraint violations

The constraint violation check can be implemented in two ways:
1) modify \texttt{SUCCESSORS} to assign only values that
   are allowed, given the values already assigned
   or 2) check constraints are satisfied before expanding a state

Backtracking search is the basic uninformed algorithm for CSPs

Can solve \( n \)-queens for \( n \approx 15 \)
Can solve $n$-queens up to $n \approx 30$

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

Simplified map-coloring example:

Terminate search when any variable has no legal values

Idea: Keep track of remaining legal values for unassigned variables

**Forward checking**
Can solve n-queens for n \approx 1000

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

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

Which variable to assign next

Which value to choose for each variable

More intelligent decisions on

Heuristics for CSPs
Can solve n-queens for n \approx 1000

C₂:\ most-constraining-variable
Given C₁ = Red, C₂ = Green, what next?

C₃ = Green: least-constraining-variable
Given C₁ = Red, C₂ = Green, choose C₃ = ?

Which variable to assign next
Which value to choose for each variable

More intelligent decisions on

Heuristics for CSPs
\( i.e., h_{\text{hillclimb}}(n) = \text{total number of violated constraints} \)

\[
\text{Choose value that violates the fewest constraints.}
\]

\text{min-conflicts heuristic:}

Variable selection: randomly select any conflicted variable

Operators reassign variable values

allow states with unsatisfied constraints

To apply to CSPs:

\text{Hill-climbing, simulated annealing typically work with}

Iterative algorithms for CSPs
Evaluation: \( h(n) = \) number of attacks

Goal test: no attacks

Operators: move queen in column

States: 4 queens in 4 columns (4^4 = 256 states)

Example: 4 Queens
The same appears to be true for any randomly-generated CSP
for arbitrary n with high probability (e.g., n = 10^4,000,000)
given random initial state, can solve n-queens in almost constant time

Performance of min-conflicts
complexity of reasoning.

An important example of the relation between syntactic restrictions and

This property also applies to logical and probabilistic reasoning:

\[ (\exists u | D | O (u | D | u) O (D | u) O ) \]

Compare to general CSPs, where worst-case time is \( O(\frac{3^n}{2}) \) time

There exists: if the constraint graph has no loops, the CSP can be solved

**Tree-structured CSPs**
Filtering example:

\[ \text{filter}(\mathcal{V}, x) \text{ removes values of } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \text{ that are inconsistent with } \mathcal{V} \]
Algorithm contd.
Tree-structured CSPs can always be solved very efficiently

Iterative min-conflicts is usually ineffective in practice

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

2) only legal successors

1) fixed variable order

Backtrack into depth-first search with

goal test defined by constraints on variable values

states defined by values of a fixed set of variables

CSPs are a special kind of problem:

Summary