Chapter 3, Section 7 and Chapter 4, Section 4.4

Constraint Satisfaction Problems
Outline

- CSP examples
- General search applied to CSPs
- Heuristics for CSPs
- Forward checking
- Backtracking
- Examples
A standard search problem:

- **State** is any old data structure that supports goal test, eval, and successor.

A **CSP** is a black box—any old data structure:

**CSPs**

A **constraint satisfaction problem (CSP)** is defined by variables \( V \), values \( D \) for these variables, and a goal test that is a set of constraints specifying allowable combinations of values for subsets of variables.
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)

$$Q_1 = 1 \quad Q_2 = 3$$

Translate each constraint into set of allowable values for its variables

E.g., values for $(Q_1, Q_2)$ are $(1, 3) \quad (1, 4) \quad (2, 4) \quad (3, 1) \quad (4, 1) \quad (4, 2)$
**Constraint graph**

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

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

\[
\begin{array}{c}
\text{SEND}\ + \ MORE \\
\hline
\text{MONEY}
\end{array}
\]

Constraints

\[
\begin{align*}
D & \neq E, \ D \neq M, \ D \neq N, \ \text{etc.} \\
D \neq E, \ D \neq M, \ D \neq N, \ \text{etc.}
\end{align*}
\]

\[
\begin{align*}
Y &= D + E \ \text{or} \ Y &= D + E \ \text{mod} \ 10, \ \text{etc.} \\
Y &= D + E \ \text{or} \ Y &= D + E \ \text{mod} \ 10, \ \text{etc.}
\end{align*}
\]

Domains

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

Variables

\[
D \in \{M, N, o, r, s\} N \neq 0, \ S \neq 0
\]

Domains

\[
D \in \{M, N, o, r, s\} N \neq 0, \ S \neq 0
\]
Example: Map coloring

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

Variables
  Countries $C_i$

Domains
  \{Red, Blue, Green\}

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

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

Floor planning

Factroy 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
Applying standard search

Notice that this is the same for all CSPs!

Goal test: 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
Implementation

CSP state keeps track of which variables have values so far.

Each variable has a domain and a current value.

Constraints can be represented explicitly as sets of allowable values, or implicitly by a function that tests for satisfaction of the constraint.

```
typedef CSP-STATE
    Unassigned, a list of variables not yet assigned
    Assigned, a list of variables that have values

typedef CSP-VAR
    Name, for I/O purposes
    Domain, a list of possible values
    Value, current value (if any)

components
    CSP-STATE
    CSP-VAR
```

AIMA Slides © Stuart Russell and Peter Norvig, 1998
Chapter 3, Section 4 and Chapter 4, Section 4.1
Standard search applied to map-coloring

\[
\text{UNASSIGNED } C1 \quad C2 \quad C3 \\
\text{ASSIGNED}
\]

\[
\begin{align*}
\text{UNASSIGNED } & C2 \quad C3 \\
\text{ASSIGNED } & C1 = \text{RED}
\end{align*}
\]

\[
\begin{align*}
\text{UNASSIGNED } & C1 \quad C3 \\
\text{ASSIGNED } & C2 = \text{BLUE}
\end{align*}
\]

\[
\begin{align*}
\text{UNASSIGNED } & C1 \quad C2 \\
\text{ASSIGNED } & C3 = \text{GREEN}
\end{align*}
\]
Complexity of the dumb approach

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.

\[ \text{Branching factor} \]
\[ \text{Search algorithm to use} \]
\[ \text{Depth of solution state} \]
\[ \text{Max. depth of space} \]
Complexity of the dumb approach

Max. depth of space $m$

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

Search algorithm to use? depth-first

Benching factor $q = \Delta_i \
\text{at top of tree}$

Order of assignment is irrelevant so many paths are equivalent

1 Adding assignments cannot correct a violated constraint

2 Order of assignment is irrelevant so many paths are equivalent

This can be improved dramatically by noting the following:

- $\Delta_i = \Theta(d)$ (at top of tree)
- $n \Delta_i = p$ (all vars assigned)
- $n \Delta_i = m$ (number of variables)
Backtracking search is the basic uninformed algorithm for CSPs. 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.

The constraint violation check can be done in the Successors function.

Use depth-first search, but can solve $n$-queens for $n \approx 15$. Use depth-first search, but
Forward checking

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

Simplified map-coloring example:

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

Can solve $n$-queens up to $n \approx 30$
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table contains an equal sign (\( = \)) and an inequality sign (\( < \)), indicating comparisons between values.
Heuristics for CSPs

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

Given $C_1 = Red, C_2 = Green$, choose $C_3 = ??$

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

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

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

$C_5: \text{most-constrained-variable}$

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. Hill-climbing simulates annealing typically work with complete states, i.e., all variables assigned.

Operators reassign variable values allow states with unsatisfied constraints.

Variable selection: randomly select any conflicted variable. Heuristic: min-conflicts heuristic: choose value that violates the fewest constraints, i.e., hill-climb 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

\[
\begin{align*}
\text{h = 5} & \quad \rightarrow \quad \text{h = 2} & \quad \rightarrow \quad \text{h = 0}
\end{align*}
\]
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

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
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 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:

\[ V_i \quad V_j \]

allowed pairs: 
\[ <1,1>, <3,2>, <3,3> \]

\[ \rightarrow \]
remove 2 from domain of \( V_i \)
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
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
- states defined by values of a fixed set of variables
- constraints defined by values of a fixed set of variables

Backtracking = depth-first search with
- only legal successors
- fixed variable order
- iterative min-conflicts is usually effective in practice
- forward checking prevents assignments that guarantee later failure
- variable ordering and value selection heuristics help significantly

Tree-structured CSPs can always be solved very efficiently

Summary