Constraint Satisfaction Problems II

AIMA: Chapter 6
Review: Constraint Satisfaction Problems

A CSP consists of:

- **Finite set of variables** $X_1, X_2, \ldots, X_n$
- **Nonempty domain of possible values** for each variable $D_1, D_2, \ldots, D_n$ where $D_i = \{v_1, \ldots, v_k\}$
- **Finite set of constraints** $C_1, C_2, \ldots, C_m$
  - Each constraint $C_i$ limits the values that variables can take.
  - A state is defined as an assignment of values to some or all variables.
- A **solution** to a CSP is a **complete, consistent** assignment, where
  - A consistent assignment does not violate the constraints.
  - An assignment is complete when every variable is assigned a value.
Review: CSP Representations

- **Constraint graph:**
  - *nodes* are variables
  - *edges* are (binary) constraints

- **Standard representation pattern:**
  - variables with values

- **Constraint graph** simplifies search.
  - e.g. Tasmania is an independent subproblem.

- **This problem: A binary CSP:**
  - each constraint relates two variables
Idea 1: CSP as a search problem

- A CSP can easily be expressed as a search problem
  - Initial State: the empty assignment {}.
  - Successor function: Assign value to any unassigned variable provided that there is not a constraint conflict.
  - Goal test: the current assignment is complete.
  - Path cost: a constant cost for every step.

- Solution is always found at depth $n$, for $n$ variables
  - Hence Depth First Search can be used
Idea 2: Improving backtracking efficiency

- **General-purpose** methods & **general-purpose** heuristics can give huge gains in speed, *on average*

- **Heuristics:**
  - Q: Which variable should be assigned next?
    1. Most constrained variable
    2. (if ties:) Most constraining variable
  - Q: In what order should that variable’s values be tried?
    3. Least constraining *value*
  - Q: Can we detect inevitable failure early?
    4. **Forward checking**
Heuristic 1: Most constrained variable

- Choose a variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic
Heuristic 2: Most constraining variable

- Tie-breaker among most constrained variables
- Choose the variable with the most constraints on remaining variables

(These two heuristics each lead to immediate solution of our example problem)
Heuristic 3: Least constraining value

- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

Note: demonstrated here independent of the other heuristics
Heuristic 4: Forward checking

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

(A first step towards Arc Consistency & AC-3)
Forward checking

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

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

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

Terminate! No possible value for SA
Example: 4-Queens Problem

(From Bonnie Dorr, U of Md, CMSC 421)
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X1 \{1,2,3,4\}

X2 \{3,4\}

X3 \{\} \{\} \{\} \{\}

X4 \{2\} \{\} \{\} \{\}

Backtrack!!!

Forward check and then BACKTRACK!
Example: 4-Queens Problem

Picking up a little later after two steps of backtracking....

Initial board:

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Assign value to unassigned variable

Diagram:

- X1: \{2, 3, 4\}
- X2: \{1, 2, 3, 4\}
- X3: \{1, 2, 3, 4\}
- X4: \{1, 2, 3, 4\}

Assignment process:

- Assign 2 to X1
- Assign 3 to X2
- Assign 4 to X3
- Assign 1 to X4

Final board:
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
**Example: 4-Queens Problem**

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Towards 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** goes beyond forward checking & repeatedly enforces constraints locally.
Arc Consistency, Constraint Propagation & AC-3
Idea 3 (big idea): *Inference* in CSPs

- CSP solvers combine search *and inference*
  - Search
    — assigning a value to a variable
  - **Constraint propagation (inference)**
    — Eliminates possible values for a variable
      if the value would violate local consistency
  - *Can do inference first, or intertwine it with search*
    — You’ll investigate this in the Sudoku homework

- Local consistency
  - **Node consistency**: satisfies unary constraints
    — This is trivial!
  - **Arc consistency**: satisfies binary constraints
    — \( X_i \) is arc-consistent w.r.t. \( X_j \) if for every value \( v \) in \( D_i \), there is some value \( w \) in \( D_j \) that satisfies the binary constraint on the arc between \( X_i \) and \( X_j \)
Review: CSP Representations

- **Constraint graph:**
  - *nodes* are variables
  - *edges* are constraints
Edges to Arcs: From Constraint Graph to DAG

- Given a pair of nodes $X_i$ and $X_j$ connected by a constraint *edge*, we represent this not by a single undirected edge, but a *pair of directed arcs*.
  - For a connected pair of nodes $X_i$ and $X_j$, there are *two* arcs that connect them: $(i,j)$ and $(j,i)$.
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 $y$
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, recheck neighbors of $X$
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, we need to recheck neighbors of $X$
- Detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
Arc Consistency

An arc \((i,j)\) is arc consistent if and only if every value \(v\) on \(X_i\) is consistent with some label on \(X_j\).

To make an arc \((i,j)\) arc consistent, for each value \(v\) on \(X_i\),
    if there is no label on \(X_j\) consistent with \(v\)
    then remove \(v\) from \(X_i\)

- Given \(d\) values, checking arc \((i,j)\) takes \(O(d^2)\) time worst case
Replacing Search: Constraint Propagation Invented…

Dave Waltz’s insight:

- By *iterating* over the graph, the arc-consistency *constraints* can be *propagated* along arcs of the graph.

- **Search**: Use constraints to *add* labels to find *one solution*

- **Constraint Propagation**: Use constraints to *eliminate* labels to simultaneously find *all solutions*
The Waltz/Mackworth Constraint Propagation Algorithm

1. Assign every node in the constraint graph a set of all possible values
2. Repeat until there is no change in the set of values associated with any node:
   3. For each node $i$:
      4. For each neighboring node $j$ in the picture:
         5. Remove any value from $i$ which is not arc consistent with $j$. 
Inefficiencies: Towards AC-3

1. At each iteration, we only need to examine those $X_i$ where at least one neighbor of $X_i$ has lost a value in the previous iteration.

2. If $X_i$ loses a value only because of arc inconsistencies with $X_j$, we don’t need to check $X_j$ on the next iteration.

3. Removing a value on $X_i$ can only make $X_j$ arc-inconsistent with respect to $X_i$ itself. Thus, we only need to check that $(j,i)$ is still arc-consistent.

These insights lead a much better algorithm...
AC-3

function AC-3(csp) return the CSP, possibly with reduced domains
inputs: csp, a binary csp with variables \{X_1, X_2, ..., X_n\}
local variables: queue, a queue of arcs initially the arcs in csp
while queue is not empty do
  (X_i, X_j) ← REMOVE-FIRST(queue)
  if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
    for each X_k in NEIGHBORS[X_i] – \{X_j\} do
      add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) return true iff we remove a value
removed ← false
for each x in DOMAIN[X_i] do
  if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraints between X_i and X_j
  then delete x from DOMAIN[X_i]; removed ← true
return removed
AC-3: Worst Case Complexity Analysis

- All nodes can be connected to every other node,
  - so each of $n$ nodes must be compared against $n-1$ other nodes,
  - so total # of arcs is $2\cdot n\cdot (n-1)$, i.e. $O(n^2)$
- If there are $d$ values, checking arc $(i,j)$ takes $O(d^2)$ time
- Each arc $(i,j)$ can only be inserted into the queue $d$ times
- Worst case complexity: $O(n^2d^3)$

(For planar constraint graphs, the number of arcs can only be linear in $N$ and the time complexity is only $O(nd^3)$)
Local search 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 reassign variable values

- Variable selection: randomly select any conflicted variable

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

- **States**: 4 queens in 4 columns ($4^4 = 256$ states)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: $h(n) = \text{number of attacks}$

Given random initial state, local min-conflicts can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 10,000,000$)
Beyond binary constraints:
Path consistency

- Generalizes arc-consistency from individual binary constraints to multiple constraints
- A pair of variables $X_i, X_j$ is path-consistent w.r.t. $X_m$ if for every assignment $X_i=a, X_j=b$ consistent with the constraints on $X_i, X_j$ there is an assignment to $X_m$ that satisfied the constraints on $X_i, X_m$ and $X_j, X_m$

- **Global constraints**
  - Can apply to any number of variables
  - E.g., in Sudoku, all numbers in a row must be different
  - E.g., in cryptarithmetic, each letter must be a different digit
  - Example algorithm:
    - If any variable has a single possible value, delete that variable from the domains of all other constrained variables
    - If no values are left for any variable, you found a contradiction
Simple CSPs can be solved quickly

1. Completely independent subproblems
   • e.g. Australia & Tasmania
   • Easiest

2. Constraint graph is a tree
   • Any two variables are connected by only a single path
   • Permits solution in time linear in number of variables
   • Do a topological sort and just march down the list

A ➔ B ➔ C ➔ D ➔ E ➔ F
Simplifying hard CSPs: Cycle Cutsets

- Constraint graph can be decomposed into a tree
  - Collapse or remove nodes
  - Cycle cutset $S$ of a graph $G$: any subset of vertices of $G$ that, if removed, leaves $G$ a tree

- Cycle cutset algorithm
  - Choose some cutset $S$
  - For each possible assignment to the variables in $S$ that satisfies all constraints on $S$
    - Remove any values for the domains of the remaining variables that are not consistent with $S$
    - If the remaining CSP has a solution, then you have are done
  - For graph size $n$, domain size $d$
    - Time complexity for cycle cutset of size $c$:
      $$O(d^c \cdot d^2(n-c)) = O(d^{c+2}(n-c))$$