Constraint Satisfaction Problems

Chapter 6.1 – 6.4

Derived from slides by S. Russell and P. Norvig, A. Moore, and R. Khoury

Constraint Satisfaction Problems (CSPs)

- Standard search problem:
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test
- CSP:
  - state is defined by variables $X$, with values from domain $D_i$
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
  - Use variable-based model
    - Solution is not a path but an assignment of values for a set of variables that satisfy all constraints

Example: 8-Queens

- Variables: 64 squares, number of queens
  $V = \{S_{1,1}, S_{1,2}, \ldots, S_{8,8}, \text{Number_of_queens} \}$
- Values: Queen or No-Queen
  $S_{i,j}$ is an element of $D_i = \{\text{queen, empty} \}$
  Number_of_queens is an element of $D_n = [0, 64]$
- Constraints: Attacks, queen count
  \n  \begin{align*}
  \text{(Number_of_queens} = 8, \\
  \text{S}_{i,j} = \text{queen } \rightarrow S_{i+k,j} &= \text{empty} \ , \\
  \text{S}_{i,j} = \text{queen } \rightarrow S_{i,j+n} &= \text{empty} \ , \\
  \text{S}_{i,j} = \text{queen } \rightarrow S_{i+n,j+n} &= \text{empty} \}
  \end{align*}
- States: All board configurations
  \begin{itemize}
  \item $\approx 2.8 \times 10^{14}$ complete states
  \item $\approx 1.8 \times 10^{14}$ complete states with 8 queens
  \item $\approx 92$ complete and consistent states
  \item $\approx 12$ unique complete and consistent states
Example: Cryptarithm

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

- 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: \( \text{Alldiff}(F, T, U, W, R, O) \)
  - \( O + O = R + 10 \cdot X_1 \)
  - \( X_1 + W + W = U + 10 \cdot X_2 \)
  - \( X_2 + T + T = O + 10 \cdot X_3 \)
  - \( X_3 = F, T \neq 0, F \neq 0 \)

Some Applications of CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetable problems
  - e.g., which class is offered when and where?
- Scheduling problems
- VLSI or PCB layout problems
- Boolean satisfiability
- N-Queens
- Graph coloring
- Games: Minesweeper, Magic Squares, Sudoku, Crosswords
- Line-drawing labeling
- Note: many real-world problems involve real-valued variables

Movie Seating Problem

A Constraint Satisfaction Problem: Graph Coloring

- Each circle marked \( V_1 \ldots V_6 \) we must assign \( R, G \) or \( B \)
- No two adjacent circles may be assigned the same value
- Note: two variables have already been assigned a color
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, or (WA,NT) in \( \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\} \)

Note: In general, 4 colors are necessary

Example: Map-Coloring

Solutions are **complete** (i.e., all variables are assigned values) and **consistent** (i.e., does not violate any constraints) assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Constraint Graph

- **Binary CSP:** each constraint relates **two** variables

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

Varieties of CSPs

- **Discrete variables**
  - finite domains:
    - \( n \) variables, domain size \( d \) \( \rightarrow O(d^n) \) complete assignments
    - e.g., Boolean CSPs, Boolean satisfiability
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., \( \text{StartJob}_1 + 5 \leq \text{StartJob}_3 \)

- **Continuous variables**
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming
Kinds of Constraints

- **Unary** constraints involve a single variable
  - e.g., SA ≠ green
- **Binary** constraints involve pairs of variables
  - e.g., SA ≠ WA
- **Higher-order** constraints involve 3 or more variables
  - e.g., cryptarithmetic column constraints

Local Search for CSPs

- Hill-climbing, simulated annealing, genetic algorithms typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with some unsatisfied constraints
  - operators assign a value to a variable
- Variable selection: randomly select any conflicted variable
- Value selection by **min-conflicts heuristic**: choose value that violates the fewest constraints, i.e., hill-climb by minimizing \( f(n) = \text{total number of violated constraints} \)

Local Search

**Min-Conflicts Algorithm:**

1. Assign to each variable a random value, defining the initial state
2. while state not consistent do
   2.1 Pick a variable, \( var \), that has constraint(s) violated
   2.2 Find value, \( v \), for \( var \) that minimizes the total number of violated constraints (over all variables)
   2.3 \( var = v \)

Example: 4-Queens

- States: 4 queens in 4 columns (\( 4^4 = 256 \) states)
- Actions: move queen to new row in its column
- Goal test: no attacks
- Evaluation function: \( f(n) = \text{total number of attacks} \)

\[ f = 5 \quad \rightarrow \quad f = 2 \quad \rightarrow \quad f = 0 \]
Min-Conflicts Algorithm

- Advantages
  - Simple and Fast: Given random initial state, can solve $n$-Queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 1,000,000$ can be solved on average in about 50 steps!)
- Disadvantages
  - Only searches states that are reachable from the initial state
  - Might not search entire state space
  - Does not allow worse moves (but can move to neighbor with same cost)
  - Might get stuck in a local optimum
- Not complete
  - Might not find a solution even if one exists

Standard Tree Search Formulation

States are defined by all the values assigned so far

- **Initial state:** the empty assignment {}  
- **Successor function:** assign a value to an unassigned variable  
- **Goal test:** the current assignment is complete: all variables assigned a value and all constraints satisfied
- Find any solution, so cost is not important
- Every solution appears at depth $n$ with $n$ variables
  - use depth-first search

DFS for CSPs

- Variable assignments are **commutative**, i.e.,
  - $[ \text{WA}=\text{R} \text{ then } \text{NT}=\text{G} ]$ same as $[ \text{NT}=\text{G} \text{ then } \text{WA}=\text{R} ]$
- What happens if we do DFS with the order of assignments as $B$ tried first, then $G$, then $R$?
- **Generate-and-test strategy:** Generate candidate solution, then test if it satisfies all the constraints
- This makes DFS look very stupid!
- Example: 
  - [link](http://www.cs.cmu.edu/~awm/animations/constraint/9d.html)
Improved DFS: Backtracking w/ Consistency Checking

- Don’t generate a successor that creates an inconsistency with any existing assignment, i.e., perform consistency checking when node is generated
- Successor function assigns a value to an unassigned variable that does not conflict with all current assignments
  - Fail if no legal assignments (i.e., no successors)
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve $n$-Queens for $n = 25$

Backtracking w/ Consistency Checking

Start with empty state

while not all vars in state assigned a value do
  Pick a variable (randomly or with heuristic)
  if it has a value that does not violate any constraints
    then Assign that value
  else
    Go back to previous variable and assign it another value

Backtracking Example
Australia Constraint Graph

Backtracking Example
Backtracking Search

• Depth-first search algorithm
  – Goes down one variable at a time
  – At a deadend, backs up to last variable whose value can be changed without violating any constraints, and changes it
  – If you backed up to the root and tried all values, then there are no solutions

• Algorithm is complete
  – Will find a solution if one exists
  – Will expand the entire (finite) search space if necessary

• Depth-limited search with depth limit = \( n \)

Improving Backtracking Efficiency

• Heuristics can give huge gains in speed
  – Which \textit{variable} should be assigned next?
  – In what order should its \textit{values} be tried?
  – Can we detect inevitable failure early?
Which Variable Next?
Most-Constrained Variable

- **Most-constrained variable:**
  - choose the variable with the fewest legal values

- Called the **minimum remaining values (MRV) heuristic**
- Tries to cut off search ASAP

Which Variable Next?
Most-Constraining Variable

- Tie-breaker among most-constrained variables
- **Most-constraining variable:**
  - choose the variable with the most constraints on the remaining variables
- Called the **degree heuristic**
- Tries to cut off search ASAP

Which Value Next?
Least-Constraining Value

- Given a variable, choose the **least-constraining value:**
  - i.e., the one that rules out the fewest values in the remaining variables
  - try to pick values best first

- Combining these heuristics makes 1000-Queens feasible

Improvement: **Forward Checking**

- At start, for each variable, record the current set of all possible legal values for it
- When you assign a value to a variable in the search, update the set of legal values for all unassigned variables. **Backtrack immediately if you empty a variable’s set of possible values.**
  - What happens if we do DFS with the order of assignments as B tried first, then G, then R?
  - Example:
    http://www.cs.cmu.edu/~awm/animations/constraint/27f.html
Forward Checking Algorithm

• Idea:
  – Keep track of remaining legal values for all variables
  – Deadend when any variable has no legal values

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 ≠ NT, or (WA,NT) in \{(red,green), (red,blue),
(green,red), (green,blue), (blue,red), (blue,green)\}

Note: in general, 4 colors are necessary

Forward Checking

• Idea:
  – Keep track of remaining legal values for all unassigned variables
  – Deadend when any variable has no legal values

Note: WA is not the most constraining var

Forward Checking

• Idea:
  – Keep track of remaining legal values for all unassigned variables
  – Deadend when any variable has no legal values

Note: Q is not most constrained var
Forward Checking

• **Idea:**
  – Keep track of **remaining legal values** for all unassigned variables
  – Deadend when any variable has **no** legal values

```
<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Note: V is **not** most constrained var

Constraint Propagation

• Forward checking propagates information from assigned to unassigned variables, but doesn’t provide early detection for **all** failures:

```
<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

• NT and SA cannot **both** be blue!
• **Constraint propagation** repeatedly enforces constraints **locally** for all variables

Main idea: When you delete a value from a variable’s domain, check all variables connected to it. If any of them change, delete all inconsistent values connected to them, etc.

In the above example, nothing changes

Arc Consistency

• Simplest form of propagation makes each arc consistent
• \( X \rightarrow Y \) is consistent iff for every value \( x \) at var \( X \) there is some allowed \( y \), i.e., there is at least 1 value of \( Y \) that is consistent with \( x \) at \( X \)

• If \( X \) loses a value, all neighbors of \( X \) need to be rechecked!

• Arc consistency detects failure earlier than forward checking
• Use as a preprocessor and after each assignment during search
Arc Consistency Algorithm “AC-3”

```
function AC-3(csp) // returns false if inconsistency is found and true otherwise
    // input: csp, a binary CSP with components (X, D, C)
    // local variables: queue, a queue of arcs; initially all arcs in csp
    while queue not empty do
        (X_i, X_j) = Remove-First(queue)
        if Revise(csp, X_i, X_j) then // make arc consistent
            if size of D_i = 0 then return false
            foreach X_k in X_i.Neighbors – {X_j} do // propagate changes to neighbors
                add (X_k, X_i) to queue
        return true

function Revise(csp, X_i, X_j) // returns true iff we revise the domain of X_i
    revised = false
    foreach x in D_i do
        if no value y in D_j allows (x, y) to satisfy the constraints between X_i and X_j then
            [delete x from D_i; revised := true]
    Check if X_i \rightarrow X_j consistent
    return revised
```
Combining Search with CSP

• Idea: Interleave search and CSP inference

• Perform DFS
  – At each node assign a selected value to a selected variable
  – Run CSP to reduce variables’ domains and check if any inconsistencies arise as a result of this assignment

Combining Backtracking Search with CSP

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
if assignment is complete then return assignment
var = SELECT-UNASSIGNED-VARIABLE(csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
do
  if value is consistent with assignment then
    add { var = value } to assignment
    inferences = AC-3(csp, var, value)
    if inferences != failure then
      add inferences to assignment
      result = BACKTRACK(assignment, csp)
      if result != failure then
        return result
      remove { var = value } and inferences from assignment
  return failure

Conflict-Directed Backjumping

• Backtracking goes back one level in the search tree at a time
  – Called Chronological backtracking
• Not good in cases where the previous step is not a cause of the conflict
• Conflict-Directed Backjumping
  – Go back to a variable involved in the conflict
  – Skip several levels if needed to get there
  – Non-chronological backtracking

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 plus consistency checking

• 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

• Iterative min-conflicts is usually effective in practice