Constraint Satisfaction Problems (Chapter 6)
What is search for?

- Assumptions: single agent, deterministic, fully observable, discrete environment

- **Search for planning**
  - The path to the goal is the important thing
  - Paths have various costs, depths

- **Search for assignment**
  - Assign values to variables while respecting certain constraints
  - The goal (complete, consistent assignment) is the important thing
Constraint satisfaction problems (CSPs)

• Definition:
  – **State** is defined by variables $X_i$ with values from domain $D_i$
  – **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables
  – **Solution** is a complete, consistent assignment

• How does this compare to the “generic” tree search formulation?
  – A more structured representation for states, expressed in a formal representation language
  – Allows useful general-purpose algorithms with more power than standard search algorithms
Example: Map Coloring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** \{red, green, blue\}
- **Constraints:** adjacent regions must have different colors 
  e.g., WA \neq NT, or (WA, NT) in \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}
Example: Map Coloring

- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Example: $n$-queens problem

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
Example: N-Queens

• **Variables:** $X_{ij}$

• **Domains:** \{0, 1\}

• **Constraints:**

  \[ \sum_{i,j} X_{ij} = N \]

  \[ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\} \]

  \[ (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\} \]

  \[ (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\} \]

  \[ (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\} \]
N-Queens: Alternative formulation

- **Variables:** $Q_i$
- **Domains:** $\{1, \ldots, N\}$
- **Constraints:**
  $$\forall i, j \text{ non-threatening } (Q_i, Q_j)$$
Example: Cryptarithmetic

- **Variables:** T, W, O, F, U, R
  
  \[ X_1, X_2 \]

- **Domains:** \{0, 1, 2, ..., 9\}

- **Constraints:**
  
  \[ O + O = R + 10 \times X_1 \]
  
  \[ W + W + X_1 = U + 10 \times X_2 \]
  
  \[ T + T + X_2 = O + 10 \times F \]

  \text{Alldiff}(T, W, O, F, U, R)

  \[ T \neq 0, F \neq 0 \]
Example: Sudoku

- **Variables:** $X_{ij}$
- **Domains:** \{1, 2, ..., 9\}
- **Constraints:**
  \[ \text{Alldiff}(X_{ij} \text{ in the same unit}) \]
Real-world CSPs

• Assignment problems
  – e.g., who teaches what class

• Timetable problems
  – e.g., which class is offered when and where?

• Transportation scheduling

• Factory scheduling

• More examples of CSPs: http://www.csplib.org/
Standard search formulation (incremental)

• **States:**
  – Variables and values assigned so far

• **Initial state:**
  – The empty assignment

• **Action:**
  – Choose any unassigned variable and assign to it a value that does not violate any constraints
  • Fail if no legal assignments

• **Goal test:**
  – The current assignment is complete and satisfies all constraints
Standard search formulation (incremental)

• What is the depth of any solution (assuming $n$ variables)?
  \[ n \] (this is good)

• Given that there are $m$ possible values for any variable, how many paths are there in the search tree?
  \[ n! \cdot m^n \] (this is bad)

• How can we reduce the branching factor?
Backtracking search

• In CSP’s, variable assignments are **commutative**
  – For example, \([WA = \text{red} \text{ then } NT = \text{green}]\) is the same as \([NT = \text{green} \text{ then } WA = \text{red}]\)

• We only need to consider assignments to a single variable at each level (i.e., we fix the order of assignments)
  – Then there are only \(m^n\) leaves

• Depth-first search for CSPs with single-variable assignments is called **backtracking search**
Example
Example
Example
Example
Backtracking search algorithm

```function Recursive-Backtracking (assignment, csp)
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable (Variables[csp], assignment, csp)
    for each value in Order-Domain-Values (var, assignment, csp)
        if value is consistent with assignment given Constraints[csp]
            add \{var = value\} to assignment
            result ← Recursive-Backtracking (assignment, csp)
            if result ≠ failure then return result
            remove \{var = value\} from assignment
    return failure```

• Making backtracking search efficient:
  – Which variable should be assigned next?
  – In what order should its values be tried?
  – Can we detect inevitable failure early?
Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. *minimum remaining values* (MRV) heuristic
Which variable should be assigned next?

• **Most constrained variable:**
  – Choose the variable with the fewest legal values
  – A.k.a. *minimum remaining values* (MRV) heuristic
Which variable should be assigned next?

• Most constraining variable:
  – Choose the variable that imposes the most constraints on the remaining variables
  – Tie-breaker among most constrained variables
Which variable should be assigned next?

• **Most constraining variable:**
  – Choose the variable that imposes the most constraints on the remaining variables
  – Tie-breaker among most constrained variables
Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables
Given a variable, in which order should its values be tried?

• Choose the **least constraining value**:
  – The value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?
Early detection of failure

function Recursive-Backtracking (assignment, csp)
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable (Variables[csp], assignment, csp)
    for each value in Order-Domain-Values (var, assignment, csp)
        if value is consistent with assignment given Constraints[csp]
            add \( \{ \text{var} = \text{value} \} \) to assignment
            result ← Recursive-Backtracking (assignment, csp)
            if result ≠ failure then return result
            remove \( \{ \text{var} = \text{value} \} \) from assignment
    return failure

Apply inference to reduce the space of possible assignments and detect failure early
Early detection of failure

Apply *inference* to reduce the space of possible assignments and detect failure early
Early detection of failure: Forward checking

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

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

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

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

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
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 repeatedly enforces constraints locally.
Arc consistency

• Simplest form of propagation makes each pair of variables consistent:
  – $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$

Consistent!
Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  - When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$

- If $X$ loses a value, all pairs $Z \rightarrow X$ need to be rechecked
Arc consistency

• Simplest form of propagation makes each pair of variables consistent:
  – $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  – When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$

• If $X$ loses a value, all pairs $Z \rightarrow X$ need to be rechecked
Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  - When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$

- If $X$ loses a value, all pairs $Z \rightarrow X$ need to be rechecked
Arc consistency

• Simplest form of propagation makes each pair of variables consistent:
  – $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  – When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$
Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  - When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$

- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment
Arc consistency algorithm AC-3

function AC-3( csp) returns the CSP, possibly with reduced domains

inputs: csp, a binary CSP with variables \( \{X_1, X_2, \ldots, X_n\} \)

local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty

\((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)\)

if \text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j) then

for each \( X_k \) in Neighbors[X_i] do

add \((X_k, X_i)\) to queue

function \text{REMOVE-INCONSISTENT-VALUES}( X_i, X_j) returns true iff succeeds

removed \leftarrow false

for each \( x \) in Domain[X_i]

if no value \( y \) in Domain[X_j] allows \((x,y)\) to satisfy the constraint \( X_i \leftrightarrow X_j \)
then delete \( x \) from Domain[X_i]; removed \leftarrow true

return removed
Does arc consistency always detect the lack of a solution?

- There exist stronger notions of consistency (path consistency, k-consistency), but we won't worry about them.
Tree-structured CSPs

- Certain kinds of CSPs can be solved without resorting to backtracking search!

- **Tree-structured CSP**: constraint graph does not have any loops

Source: P. Abbeel, D. Klein, L. Zettlemoyer
Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs
Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards

http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs
Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering.
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards.
- Forward assignment phase: select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is consistent.

http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs
Algorithm for tree-structured CSPs

• If $n$ is the number of variables and $m$ is the domain size, what is the running time of this algorithm?
  
  – $O(nm^2)$: we have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values
Nearly tree-structured CSPs

• **Cutset conditioning:** find a subset of variables whose removal makes the graph a tree, instantiate that set in all possible ways, prune the domains of the remaining variables and try to solve the resulting tree-structured CSP

• Cutset size $c$ gives runtime $O(m^c (n – c)m^2)$

Source: P. Abbeel, D. Klein, L. Zettlemoyer
Algorithm for tree-structured CSPs

• Running time is $O(nm^2)$
  (n is the number of variables, m is the domain size)
    – We have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values

• What about backtracking search for general CSPs?
  – Worst case $O(m^n)$

• Can we do better?
Computational complexity of CSPs

• The satisfiability (SAT) problem:
  – Given a Boolean formula, is there an assignment of the variables that makes it evaluate to true?

\[(X_1 \lor \overline{X}_7 \lor X_{13}) \land (\overline{X}_2 \lor X_{12} \lor X_{25}) \land \ldots\]

• SAT is **NP-complete**
  – NP: class of decision problems for which the “yes” answer can be verified in polynomial time
  – An **NP-complete** problem is in NP and every other problem in NP can be efficiently reduced to it (Cook, 1971)
  – Other NP-complete problems: graph coloring, n-puzzle, generalized sudoku
  – It is not known whether P = NP, i.e., no efficient algorithms for solving SAT in general are known
Local search for CSPs

- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints

\[ h = \text{number of conflicts} \]
Local search for CSPs

• Start with “complete” states, i.e., all variables assigned
• Allow states with unsatisfied constraints
• Attempt to improve states by reassigning variable values
• Hill-climbing search:
  – In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  – I.e., attempt to greedily minimize total number of violated constraints
  – Problem: local minima

$h = 1$
Local search for CSPs

- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
- Problem: local minima

- For more on local search, see ch. 4
CSP in computer vision: Line drawing interpretation

An example polyhedron:

Variables: edges
Domains: +, -, →, ←

Desired output:
CSP in computer vision: Line drawing interpretation

Four vertex types:

- L
- Y
- T
- Arrow

Constraints imposed by each vertex type:

- L
- Y
- T
- Arrow

David Waltz, 1975
CSP in computer vision: 4D Cities

1. When was each photograph taken?
2. When did each building first appear?
3. When was each building removed?

Set of Photographs:

Set of Objects: Buildings


http://www.cc.gatech.edu/~phlosoft/
CSP in computer vision: 4D Cities

- Observed
- Missing
- Occluded

Columns: images
Rows: points

Satisfies constraints:

Violates constraints:

- Goal: reorder images (columns) to have as few violations as possible
CSP in computer vision: 4D Cities

- **Goal:** reorder images (columns) to have as few violations as possible
- **Local search:** start with random ordering of columns, swap columns or groups of columns to reduce the number of conflicts

- Can also reorder the rows to group together points that appear and disappear at the same time – that gives you buildings
Summary

• CSPs are a special kind of search problem:
  – States defined by values of a fixed set of variables
  – Goal test defined by constraints on variable values
• **Backtracking** = depth-first search where successor states are generated by considering assignments to a single variable
  – **Variable ordering** and **value selection** heuristics can 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
• Complexity of CSPs
  – NP-complete in general (exponential worst-case running time)
  – Efficient solutions possible for special cases (e.g., tree-structured CSPs)
• Alternatives to backtracking search: local search