

# Lecture 6: Constraint Satisfaction

10:30 AM, Feb 10, 2009

## Contents

<b>1 Overview</b>	<b>1</b>
<b>2 Constraint Satisfaction Problems</b>	<b>2</b>
<b>3 Systematic Search</b>	<b>3</b>
3.1 Variable Ordering . . . . .	4
3.2 Value Ordering . . . . .	4
3.3 Constraint Propagation (Rough Draft) . . . . .	6

## 1 Overview

Our next series of lectures is concerned with search algorithms for solving constraint satisfaction and constraint optimization problems. Constraint satisfaction problems (*e.g.*,  $n$ -queens and map coloring) are problems in which the search is for a **feasible** solution, that is, one which satisfies all the constraints. Optimization problems (*e.g.*, traveling salesperson problem) are problems in which the search is for an optimal solution. Constraint optimization problems are optimization problems in which the search for an optimal solution is confined to the set of feasible solutions.

Specifically, we study:

1. satisfiability: systematic search algorithms, namely Davis–Putnam and Davis, Putnam, Logemann, and Loveland, and local search algorithms, namely GSAT, and WALKSAT
2. constraint satisfaction problems (CSPs): backtracking, variable and value ordering heuristics, consistency techniques, and constraint propagation
3. mathematical programming: linear and integer linear programming

By definition, constraint satisfaction involves “hard” constraints—constraints that either pass or fail, with no middle ground. Nonetheless, constraint satisfaction problems are often solved as optimization problems with “soft” constraints. Here, the objective is to maximize the number of constraints satisfied; or equivalently, to minimize the number of constraints violated.

On the other hand, optimization problems can be solved using routines designed to solve constraint satisfaction problems. In a maximization (minimization) problem, add the constraint that the value of the solution must be greater (less) than  $k$ ; solve; if a feasible solution exists, increment (decrement)  $k$  by  $\epsilon > 0$  and repeat; otherwise, return  $k$ . The procedure returns an  $\epsilon$ -optimal solution: i.e., one that is within  $\epsilon$  of the optimal.

The topic of this lecture is constraint satisfaction problems (CSPs). Examples of CSPs, in addition to the  $n$ -queens problem and map coloring, include cryptarithmic puzzles and crossword puzzles. In Lectures 3 and 4, we discussed ways to extend blind search methods with domain-specific heuristics to yield heuristic search techniques. In this lecture, we show how blind search methods can be extended with CSP-specific heuristics.

## 2 Constraint Satisfaction Problems

Following the pattern of our search and optimization lectures, let's begin this lecture with a formal definition of CSPs. A (finite) **constraint satisfaction problem** is a triple  $\langle X, \mathcal{D}, \mathcal{C} \rangle$ , where

- a (finite) set of variables  $X = \{x_1, \dots, x_n\}$
- a set of (finite) domains  $\mathcal{D} = \{D_1, \dots, D_n\}$ ,  
with  $D_i = \{v_{i1}, \dots, v_{ik_i}\}$ , where  $k_i$  is the cardinality of domain  $D_i$  and  $1 \leq i \leq n$
- a (finite) set of constraints  $\mathcal{C} = \{C_1, \dots, C_m\}$

In (finite) CSPs, constraints can be expressed either **extensionally** or **intensionally**. Given a CSP with two variables  $x_1$  and  $x_2$ , if  $D_1 = \{A, B\}$  and  $D_2 = \{B, C\}$ , then the constraint “the value of  $x_1$  cannot equal the value of  $x_2$ ” is expressed extensionally as  $\{(A, B), (A, C), (B, C)\} \subseteq \{(A, B), (A, C), (B, B), (B, C)\}$  and intensionally as  $x_1 \neq x_2$ .

**Unary** constraints restrict the value of a single variable: *e.g.*,  $x \neq 0$ . **Binary** constraints relate two variables: *e.g.*,  $x_1 \neq x_2$ . **Higher-order** constraints relate three or more variables.

Binary CSPs are typically visualized using **constraint graphs**,  $G(X, \mathcal{D}, \mathcal{C})$ : the nodes correspond to the variables, and the edges correspond to the constraints between the nodes/variables at the endpoints of the edges.

An **assignment** is a mapping from variables to values. A **consistent** assignment does not violate any constraints. A **complete** assignment assigns a value to all variables. A **solution** to a CSP is a complete and consistent assignment.

**Example:** A classic example of a CSP is the  $n$ -queens problem. In this problem,  $n$ -queens are to be placed on an  $n \times n$  chess board such that no two queens threaten each other: *i.e.*, no two queens occupy the same row, column, or diagonal.

In the case of 4-queens, the problem can be represented as follows.

- variables:  $x_1, x_2, x_3, x_4$ , where variable  $x_i$  denotes the row of the queen in the  $i$ th column
- domains:  $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$
- constraints:
  - no two queens can occupy the same column: this constraint is automatically satisfied in this formulation of the problem
  - no two queens can occupy the same row: *i.e.*,  $x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4$

- no two queens can occupy the same diagonal: *i.e.*,  $|x_1 - x_2| \neq 1$ ,  $|x_1 - x_3| \neq 2$ ,  $|x_1 - x_4| \neq 3$ ,  $|x_2 - x_3| \neq 1$ ,  $|x_2 - x_4| \neq 2$ ,  $|x_3 - x_4| \neq 1$

The assignment  $\{x_1 \mapsto 3, x_2 \mapsto 1, x_3 \mapsto 4, x_4 \mapsto 2\}$  solves the 4-queens problem. A solution to the 8-queens problem is depicted in Figure 1.

				Q			
		Q					
Q							
						Q	
	Q						
							Q
					Q		
			Q				

Figure 1: A solution to the 8-queens problem.

### 3 Systematic Search

There are two basic approaches to solving CSPs: the first is as search; the second is as optimization. We have already alluded to how CSPs can be posed as optimization problems (and hence solved using, for example, local search techniques). This lecture focuses on solving CSPs using systematic search. To begin, we formulate a CSP as a basic search problem:

- $X$ , the set of states, is the set of partial assignments
- $G$ , the set of goal states, is the set of complete (and hence, consistent) assignments
- $S = \{\}$  is the start state
- $\delta : X \rightarrow 2^X$ , the successor relation, extends a partial assignment of length  $k$  to one of length  $k + 1$  in a consistent manner

Given this formulation, we can apply depth-first search to try to find a solution to the CSP. Unlike the depth-first search pseudocode presented in Lecture 2, we present recursive depth-first search below (see Table 1), which generates only one successor node at a time rather than all successors. This algorithm facilitates another memory-saving trick (beyond the usual savings achieved by depth-first search over breadth-first search) because successors are generated by modifying, rather than making copies of, the current node.

Translating back into the language of CSPs, recursive depth-first search is called **backtracking**. The **backtracking** pseudocode (see Table 2) leaves two things unspecified: how to order variables for assignment (*i.e.*, what variable should be assigned a value next?), and how to order the values for that variable. Next, we describe variable and value ordering heuristics that guide these choices.

<p>DEPTHFIRSTSEARCH(<math>X, S, G, \delta</math>)</p> <p>Inputs search problem</p> <p>Output (path to) goal node or failure</p>
<p>1. for all <math>s \in S</math></p> <p style="padding-left: 40px;">(a) RECURSIVEDFS(<math>X, S, G, \delta, s</math>)</p>
<p>RECURSIVEDFS(<math>X, S, G, \delta, n</math>)</p> <p>Inputs search problem</p> <p style="padding-left: 40px;">search node <math>n</math></p> <p>Output (path to) goal node or failure</p>
<p>1. if <math>n \in G</math>, return (path to) <math>n</math></p> <p>2. for all <math>m \in \delta(n)</math></p> <p style="padding-left: 40px;">(a) <math>Z = \text{RECURSIVEDFS}(X, S, G, \delta, m)</math></p> <p style="padding-left: 40px;">(b) if <math>Z \neq \text{fail}</math>, then return <math>Z</math></p> <p>3. fail</p>

Table 1: Recursive Depth-First Search.

### 3.1 Variable Ordering

If a branch of backtracking search is destined to fail, it is preferable for it to fail sooner rather than later. Hence, a sensible way to order variables is from most-constrained to least-constrained. There are two popular heuristics that achieve this ordering. The first is the **degree heuristic**, which simply counts up the number of constraints that each variable is a part of, and chooses one for which this count is maximal. The second is the **minimum remaining values (MRV)** heuristic, which counts up the size of each variable's domain during each recursive call, and chooses one for which this count is minimal. This heuristic is also called **fail first** because failure is detected immediately if ever there is a variable whose domain becomes empty.

The MRV (or fail-first) heuristic is often used in conjunction with **forward checking**. With each variable assignment, forward checking deletes any inconsistent values in the domains of neighboring variables in the constraint graph. A value  $v_{jk}$  for variable  $x_j$  is **inconsistent** with a value  $v_{ik}$  for variable  $x_i$  if assigning  $v_{jk}$  to  $x_j$  and  $v_{ik}$  to  $x_i$  violates a constraint. A subroutine to check for and then remove any inconsistencies is shown in Table 3. Note that this subroutine is asymmetric: it assumes that variable  $x_i$  is assigned a value and checks for and then removes any inconsistent values in the domain of variable  $x_j$ . Forward checking is called in between steps 1 and 2 in the recursive backtracking pseudocode.

### 3.2 Value Ordering

Once a variable is chosen, the next obvious question is: what value should it be assigned? That is, in what order should values be assigned to variables? A solution to a CSP is more likely to be found if the domains of subsequent variables are bigger rather than smaller. Hence, it is reasonable to

<p>BACKTRACKING(<math>X, \mathcal{D}, \mathcal{C}</math>)</p> <p>Inputs   CSP</p> <p>Output   satisfying assignment or failure</p>
<p>RECURSIVEBT(<math>X, \mathcal{D}, \mathcal{C}, \{\}</math>)</p>
<p>RECURSIVEBT(<math>X, \mathcal{D}, \mathcal{C}, A</math>)</p> <p>Inputs   CSP</p> <p>          assignment <math>A</math></p> <p>Output   satisfying assignment or failure</p>
<ol style="list-style-type: none"> <li>1. if <math>A</math> is complete, <b>return</b> <math>A</math></li> <li>2. <i>select</i> unassigned variable <math>x_i</math></li> <li>3. <i>order</i> domain values <math>D_i</math></li> <li>4. for all <math>v_{ik} \in D_i</math> <ol style="list-style-type: none"> <li>(a) if <math>v_{ik}</math> is consistent with <math>A</math> according to <math>\mathcal{C}</math> <ol style="list-style-type: none"> <li>i. add <math>\{x_i \rightarrow v_{ik}\}</math> to <math>A</math></li> <li>ii. <math>Z = \text{RECURSIVEBT}(X, \mathcal{D}, \mathcal{C}, A)</math></li> <li>iii. if <math>Z \neq \text{fail}</math>, then <b>return</b> <math>Z</math></li> <li>iv. delete <math>\{x_i \rightarrow v_{ik}\}</math> from <math>A</math></li> </ol> </li> </ol> </li> <li>5. <b>fail</b></li> </ol>

Table 2: Recursive Backtracking for CSPs.

<p>RESTRICTDOMAIN(<math>X, \mathcal{D}, \mathcal{C}, v_{ik}, x_j</math>)</p> <p>Inputs   CSP</p> <p>          assignment <math>A</math></p> <p>          variable <math>x_j</math></p> <p>          value <math>v_{ik}</math></p> <p>Output   CSP</p>
<ol style="list-style-type: none"> <li>1. for all values <math>v_{jk} \in D_j</math> <ol style="list-style-type: none"> <li>(a) if <math>v_{jk}</math> is inconsistent with <math>v_{ik}</math> according to <math>\mathcal{C}</math> <ol style="list-style-type: none"> <li>i. <math>D_j = D_j \setminus \{v_{jk}\}</math> /* delete <math>v_{jk}</math> from <math>x_j</math>'s domain */</li> </ol> </li> </ol> </li> <li>2. <b>return</b> <math>(X, \mathcal{D}, \mathcal{C})</math></li> </ol>

Table 3: Inconsistency Check.

order values for the current variable from least-constraining to most-constraining. This is precisely what the **least-constraining value** (LCV) heuristic does: it chooses a value that rules out the fewest values for neighboring variables in the constraint graph. See Table 5.

FORWARDCHECKING( $X, \mathcal{D}, \mathcal{C}, A, x_i$ )	
Inputs	CSP current assignment $A$ most-recently-assigned variable $x_i$
Output	CSP
<ol style="list-style-type: none"> <li>1. for all unassigned variables <math>x_j</math> whose values are constrained by the value of <math>x_i</math> <ol style="list-style-type: none"> <li>(a) <math>(X, \mathcal{D}, \mathcal{C}) = \text{RESTRICTDOMAIN}(X, \mathcal{D}, \mathcal{C}, A(x_i), x_j)</math></li> </ol> </li> <li>2. <b>return</b> <math>(X, \mathcal{D}, \mathcal{C})</math></li> </ol>	

Table 4: Forward Checking.

LCV( $X, \mathcal{D}, \mathcal{C}, A, x_i$ )	
Inputs	CSP current assignment $A$ about-to-be assigned variable $x_i$
Output	sorted list of values in domain $D_i$
Initialize	for all $1 \leq k \leq k_i, r_k = 0$
<ol style="list-style-type: none"> <li>1. for all <math>v_{ik} \in D_i</math> <ol style="list-style-type: none"> <li>(a) let <math>A' = A \cup \{x_i \mapsto v_{ik}\}</math></li> <li>(b) for all unassigned variables <math>x_j</math> whose value is constrained by the value of <math>x_i</math> <ol style="list-style-type: none"> <li>i. <math>(X, \mathcal{D}', \mathcal{C}) = \text{RESTRICTDOMAIN}(X, \mathcal{D}, \mathcal{C}, A'(x_i), x_j)</math></li> <li>ii. <math>r_k = r_k + ( D_j  -  D'_j )</math></li> </ol> </li> </ol> </li> <li>2. <b>return</b> <math>v_{ik}</math>s sorted in nondecreasing order by <math>r_k</math> values</li> </ol>	

Table 5: Least-Constraining Value.

### 3.3 Constraint Propagation (Rough Draft)

Backtracking can be further enhanced with constraint propagation techniques, such as validating node, arc, and path consistency. In this section, we restrict our attention to CSPs with only unary and binary constraints, so called binary CSPs. (**Fact:** Any CSP with  $n$ -ary constraints can be converted to a binary CSP.)

A graph is said to be **node consistent** iff for all variables  $x_i$ , any unary constraints on the values of  $D_i$  are satisfied. Unary constraints are satisfied by simply restricting domains.

A graph is said to be **arc consistent** iff for all variables  $x_i, x_j$  for which there exists (directed) arc  $(x_i, x_j)$ , for all  $d \in D_i$ , there exists  $d' \in D_j$  that is consistent with the constraint represented by the arc  $(x_i, x_j)$ .

<p>BACKTRACKING*(<math>X, \mathcal{D}, \mathcal{C}</math>)</p> <p>Inputs CSP</p> <p>Output satisfying assignment or failure</p>
<p>RECURSIVEBT*(<math>X, \mathcal{D}, \mathcal{C}, \{\}</math>)</p>
<p>RECURSIVEBT*(<math>X, \mathcal{D}, \mathcal{C}, A</math>)</p> <p>Inputs CSP</p> <p>assignment <math>A</math></p> <p>Output satisfying assignment or failure</p>
<ol style="list-style-type: none"> <li>1. if <math>A</math> is complete, <b>return</b> <math>A</math></li> <li>2. <i>select</i> unassigned variable <math>x_i</math> /* e.g., using MRV heuristic */</li> <li>3. <i>order</i> domain values <math>D_i</math> /* e.g., using LCV heuristic */</li> <li>4. for all <math>v_{ik} \in D_i</math> <ol style="list-style-type: none"> <li>(a) if <math>v_{ik}</math> is consistent with <math>A</math> according to <math>\mathcal{C}</math> <ol style="list-style-type: none"> <li>i. let <math>A' = A \cup \{x_i \rightarrow v_{ik}\}</math>: i.e., let <math>D'_i = \{v_{ik}\}</math></li> <li>ii. <math>(X, \mathcal{D}'', \mathcal{C}) = \text{FORWARDCHECKING}(X, \mathcal{D}', \mathcal{C}, A', x_i)</math></li> <li>iii. <math>Z = \text{RECURSIVEBT}^*(X, \mathcal{D}'', \mathcal{C}, A')</math></li> <li>iv. if <math>Z \neq \text{fail}</math>, then <b>return</b> <math>Z</math></li> </ol> </li> </ol> </li> <li>5. <b>return fail</b></li> </ol>

Table 6: Recursive Backtracking with Forward Checking.

ARCCONSISTENCY( $X, \mathcal{D}, \mathcal{C}, A$ )	
Inputs	CSP current assignment $A$
Output	CSP
Initialize	queue $Q$ of edges among unassigned variables
<p><b>while</b> <math>Q</math> is not empty</p> <ol style="list-style-type: none"> <li>1. delete edge <math>(x_i, x_j)</math> from <math>Q</math></li> <li>2. for all values <math>v_{ik} \in D_i</math> <ol style="list-style-type: none"> <li>(a) CONSISTENT := false</li> <li>(b) for all values <math>v_{jk} \in D_j</math> <ol style="list-style-type: none"> <li>i. if <math>v_{ik}</math> is consistent with <math>v_{jk}</math>, then CONSISTENT = true and break out of this loop</li> </ol> </li> <li>(c) if CONSISTENT == false           <ol style="list-style-type: none"> <li>i. <math>D_i = D_i \setminus \{v_{ik}\}</math> /* delete <math>v_{ik}</math> from <math>x_i</math>'s domain */</li> <li>ii. for all unassigned variables <math>x_j</math> whose value is constrained by <math>x_i</math>, insert edge <math>(x_j, x_i)</math> into <math>Q</math>, ignoring duplicates</li> </ol> </li> </ol> </li> </ol> <p><b>return</b> <math>(X, \mathcal{D}, \mathcal{C})</math></p>	

Table 7: Arc Consistency.

<p>BACKTRACKING**(X, D, C)  Inputs CSP  Output satisfying assignment or failure</p>
<ol style="list-style-type: none"> <li>1. (X, D, C) = NODECONSISTENCY(X, D, C)</li> <li>2. (X, D, C) = ARCCONSISTENCY(X, D, C, {})</li> <li>3. RECURSIVEBT**(X, D, C, {})</li> </ol>
<p>RECURSIVEBT**(X, D, C, A)  Inputs CSP  assignment A  Output satisfying assignment or failure</p>
<ol style="list-style-type: none"> <li>1. if A is complete, <b>return</b> A</li> <li>2. <i>select</i> unassigned variable <math>x_i</math> /* e.g., using MRV heuristic */</li> <li>3. <i>order</i> domain values <math>D_i</math> /* e.g., using LCV heuristic */</li> <li>4. for all <math>v_{ik} \in D_i</math> <ol style="list-style-type: none"> <li>(a) if <math>v_{ik}</math> is consistent with A according to C <ol style="list-style-type: none"> <li>i. let <math>A' = A \cup \{x_i \rightarrow v_{ik}\}</math>: i.e., let <math>D'_i = \{v_{ik}\}</math></li> <li>ii. (X, D'', C) = FORWARDCHECKING(X, D', C, A', <math>x_i</math>)</li> <li>iii. (X, D''', C) = ARCCONSISTENCY(X, D'', C, A')</li> <li>iv. Z = RECURSIVEBT**(X, D''', C, A')</li> <li>v. if <math>Z \neq \text{fail}</math>, then <b>return</b> Z</li> </ol> </li> </ol> </li> <li>5. <b>return fail</b></li> </ol>

Table 8: Recursive Backtracking with Node and Arc Consistency.