Constraint Programming

Introduction (2)
In This Lecture

• Lecture 1: Recap
• The Constrained Optimisation Problem.
• Classes vs. Instances.
• Representing Constraints.
• Constraint Languages.
• Constraint Models:
  • Crystal Maze
  • Sudoku
  • Kakuro
Lecture 1: Recap
A constraint model maps the features of a combinatorial problem onto the features of a constraint satisfaction problem (CSP).
The (finite-domain) Constraint Satisfaction Problem

• **Given:**
  1. A finite set of *decision variables*.
  2. For each decision variable, a finite *domain* of potential values.
  3. A finite set of *constraints* on the decision variables.

• **Find:**
  • An assignment of values to variables such that all constraints are satisfied.
Constraint Solving

• Typically interleaves 2 components:
  1. Systematic **Search** through a space of partial assignments.
     - Extend an assignment to a subset of the variables incrementally.
     - Backtrack if establish that current partial assignment **cannot** be extended to a solution.
  2. Constraint **Propagation**.
     - Deduction based on constraints, current domains.
     - Usually recorded as reductions in domains.
The Crystal Maze Puzzle

- We saw how a little bit of search and some propagation can solve this puzzle.
- Also talked about heuristics, symmetry.
The Sudoku Puzzle

- We saw that this puzzle has a very natural constraint model.
- And that, in most cases, propagation alone is enough to solve this puzzle.
The Constrained Optimisation Problem
The Constrained Optimisation Problem (COP)

Given:
• A CSP + an **objective function**.
  • E.g. maximise/minimise value of some variable/expression.

Find:
• An assignment of values to variables such that:
  • All constraints are satisfied.
  • The objective is optimised.
The Constrained Optimisation Problem (COP)

• What kind of problem do we need COPs to model?
• Very common, e.g.
  • Minimising waste or cost.
  • Minimising time taken (e.g. for scheduling).
  • Maximising profit.
The Steel Mill
Slab Design Problem

• NB This is a simplification of a real problem that IBM solved for a Korean Steel manufacturer.
• The mill can make different slab sizes.
• **Given** the set of slab sizes and input orders with:
  • A *colour* (route through the mill).
  • A *weight*.
• **Pack** orders onto slabs such that the total steel made is minimised, subject to:
  • Capacity constraints.
  • Colour constraints.
Slab Design Constraints

• **Capacity:**
  • Total weight of orders assigned to a slab cannot exceed slab capacity.

• **Colour:**
  • Each slab can contain at most 2 colours.
  • Reason: expensive to cut slabs up to send them to different parts of the mill.
An Example

- Slab Sizes: 1, 3, 4
- Orders: a, …, i
- Colours: red, green, blue, orange, brown
A Solution

• 6 Slabs:

- (size 4)
- (size 3)
- (size 3)
- (size 1)
- (size 1)
- (size 1)
Slab Design Decision

Variables

• What do we need to decide to solve this problem?
  1. How many slabs we are going to use.
  2. How big each slab is.
  3. Which orders are on which slabs.
Decision Variables: Slab Sizes

- Worst case: put each order on its own slab.
- Introduce decision variables $s_1$-$s_9$.
- Domain: set of sizes the mill can make.
- Objective: minimise sum of $s_1$-$s_9$.

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
<th>$s_8$</th>
<th>$s_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
<td>{0,1, 3,4}</td>
</tr>
</tbody>
</table>

What’s the “0” for?
A Solution Revisited

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
<th>$s_8$</th>
<th>$s_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${4}$</td>
<td>${3}$</td>
<td>${3}$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${0}$</td>
<td>${0}$</td>
<td>${0}$</td>
</tr>
</tbody>
</table>
### Decision Variables: Orders on Slabs

<table>
<thead>
<tr>
<th>Slabs</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4...</td>
<td>0</td>
</tr>
</tbody>
</table>

Each column sums to 1:
each order assigned to one slab.

Weighted sum of each row is at most slab size.
**Decision Variables: Colours on Slabs**

<table>
<thead>
<tr>
<th>Slabs</th>
<th>Colours</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Each row sums to at most 2.

If \( \text{Orders}[i,j] = 1 \) then \( \text{Colours}[	ext{colour_of}(i), j] = 1 \)
## Example

### Table 1

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Orange</th>
<th>Brown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Solution

\[ s_1 = 4, \ s_2 = 3, \ s_3 = 3, \ s_4 = 3, \ s_5 \ldots 9 = 0 \]

\[
\begin{array}{cccccccc}
\toprow
a & b & c & d & e & f & g & h & i \\
\midrow
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{ccccc}
\toprow
\text{Red} & \text{Green} & \text{Blue} & \text{Orange} & \text{Brown} \\
\midrow
1 & 0 & 0 & 0 & 1 \\
2 & 1 & 1 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 1 & 1 \\
\ldots & 0 & 0 & 0 & 0 \\
\end{array}
\]
Problem Classes vs Problem Instances
Problem Classes

- A problem class describes a family of problems, related by a common set of parameters.
- Obtain an instance: give values for the parameters.
- Example: $n$-queens problem class. Place $n$ queens on an $n \times n$ chess board such that no pair of queens attack each other.
- Here is a solution to the 4-queens instance.
The Sudoku Problem Class

- Sudoku is parameterised by the set of filled-in cells in the grid:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7</td>
<td>8</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Slab Design Problem Class

• In Steel Mill Slab Design, the parameters are:
  • the set of slab sizes the mill can make
  • and the set of orders.

• Slab Sizes: \{1, 3, 4\}
Crystal Maze?

- What about the Crystal Maze puzzle?

There are no parameters. This is a single problem instance.
A Crystal Maze Problem Class?

• How might we turn the crystal maze into a problem class?
• We need to **parameterise** it.
• What might we parameterise?

The graph!
Important: Constraint Solvers Solve Instances

- A **problem class** is specified by one or more parameters.
- An **instance** is specified from a class by instantiating the parameters to particular values.
- An individual CSP/COP represents a problem instance.
- Constraint solvers solve individual instances.
Representing Constraints
Recall: 3. Constraints

- **scope**: subset of the decision variables a constraint involves.
- Of the possible combinations of assignments to the variables in its scope, a constraint specifies:
  - Which are allowed. Assignments that **satisfy** the constraint.
  - Which are disallowed. Assignments that **violate** the constraint.
- I.e. can think of a constraint as a **relation**.
One Constraint to Rule Them All: Table

- The **table** constraint:
  - The most basic constraint available to us.
  - Consists of literally listing the satisfying combinations of assignments.
- Also known as the **extensional** representation.
Table Constraint: The Good News

- We can represent any constraint we wish to model like this.
- Example: Let’s say I have two variables.
  - X, with domain \{1, 2, 3\}
  - Y, with domain \{1, 2\}
- And I wish to constrain X to be greater than Y.

<table>
<thead>
<tr>
<th>Table(X, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨2, 1⟩</td>
</tr>
<tr>
<td>⟨3, 1⟩</td>
</tr>
<tr>
<td>⟨3, 2⟩</td>
</tr>
</tbody>
</table>
Table Constraint: The Bad News

• The table constraint can be very cumbersome.
• Think about modelling AllDifferent on 9 variables like this (e.g. for Sudoku).
  • That’s 9! Tuples…
The Solution: Intensional Representation

- Constraint solvers provide a library of commonly-occurring constraints that can be specified much more concisely.
  - E.g. AllDifferent.
- Internally, the solver usually represents these constraints **Intensionally**:
  - An expression that can be evaluated:
    - E.g. =, <, ≤, ≠.
  - An algorithm that can be executed:
    - AllDifferent, various kinds of counting constraints.
Constraints: Intensional Representation

• This can be a big win.
• Consider again $X > Y$:
  • Simple to check whether an assignment satisfies this constraint.
  • But consider how large the set of allowed tuples might be if $X$ and $Y$ both had domains with 100 elements…
  • Much more space required, and searching through all of those tuples would take much more time.
Why Bother with the Extensional/Table Representation?

• Sometimes it is the only sensible option.
• E.g. Table($p_1$, $p_2$),
  • where $p_1$ and $p_2$ are variables whose domains are people,
  • and the constraint being represented is $p_1$ likes $p_2$.
  • No obvious algorithmic representation for that!
Constraint Languages
Constraint Languages and Constraint Programs

• We do not usually work directly with CSP/COPs, which can be large and cumbersome.
• Instead we write constraint programs (also known as constraint models) in constraint languages.
• A constraint program/model is a recipe.
  • When followed, produces a CSP/COP.
Constraint Languages: Common Features

• Allow us to declare decision variables, and their domains.
• Often support arrays of variables.
  • Remember the order and colour grids from the Steel Mill example.
  • And iteration over these arrays for concision.
• Allow us to model problem classes.
Constraint Languages: Common Features

• All support **extensional/table** constraints.
  • These are our basic building blocks.
• Equality, disequality, inequality.
• Operators allow us to build constraint expressions:
  • Arithmetic: $+$, $-$, $\times$, absolute value.
  • Logical: AND, OR, NOT
• These constraints are represented intensionally.
• Four components:
  1. Given parameters
  2. Find decision variables
  3. Such That constraints
  4. Min/Maximising objective
The Story So Far

- Constraint Models are recipes written in constraint languages.
  - Follow the recipe to get a CSP/COP.
- The model can be parameterised to represent a problem class.
  - Give values for the parameters to obtain an instance.
  - A CSP/COP corresponds to a single instance.
- Feed the CSP/COP to a constraint solver to get a solution to that instance.
Examples: The Crystal Maze
The Crystal Maze Puzzle

- **Find** $x_1 \ldots x_8$ **each with domain**
  \{1,2,3,4,5,6,7,8\}
The Crystal Maze Puzzle: Adjacency Constraint

- What is the arity of the adjacency constraint?
The Crystal Maze Puzzle: Adjacency Constraint

- This is a binary constraint on pairs of variables, e.g. $x_1$, $x_2$, both of which have domain:
  - $\{1, 2, 3, 4, 5, 6, 7, 8\}$
Adjacency: Extensional/Table Representation

- Domains: \{1, 2, 3, 4, 5, 6, 7, 8\}

<table>
<thead>
<tr>
<th>Table(x1,x2)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;1, 2&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;1, 3&gt;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>&lt;2, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;2, 2&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;2, 3&gt;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>&lt;8, 8&gt;</td>
<td></td>
</tr>
</tbody>
</table>

64 possible combinations
Adjacency: Extensional/Table Representation

- Constraint:
  $x_1, x_2$ are not assigned consecutive numbers.

Notice that $\langle 1, 2 \rangle$ is missing:

Notice that $\langle 2, 1 \rangle, \langle 2, 3 \rangle$ are missing:

50 combinations allowed
Adjacency: Intensional Representation

- Constraint:
  \(x_1, x_2\) are not assigned consecutive numbers.
- Intensionally: \(|x_1 - x_2| > 1\), …
  - Solver has to support:
    - “greater-than” constraint.
    - Absolute value operator.
    - Subtraction operator.
The Crystal Maze Puzzle: Distinctness Constraint

• What is the arity of this constraint?
Distinctness: Intensional Representation

- Intensionally: `AllDifferent(x_1, ..., x_8)`.
- NB This is a very common & useful constraint.
Distinctness: Extensional Representation

- Extensionally: \( \text{Table}(x_1, \ldots, x_8) = \{\langle 1,2,3,4,5,6,7,8 \rangle, \langle 1,2,3,4,5,6,8,7 \rangle, \ldots \} \).

- NB There are 8! of these tuples.
Distinctness: Pairwise Intensional

- Pairwise intensionally:
  - $x_1 \neq x_2$, $x_1 \neq x_4$, $x_2 \neq x_4$, ...

Qn: What do we lose by doing this?
Distinctness: Pairwise Intensional

- Pairwise extensionally:
  - Table($x_1$, $x_2$) = \{\langle1, 2\rangle, \langle1, 3\rangle, \langle1, 4\rangle, \langle1, 5\rangle, \langle1, 6\rangle, \langle1, 7\rangle, \\ \langle1, 8\rangle, \langle2, 1\rangle, \langle2, 3\rangle, \langle2, 4\rangle, \langle2, 5\rangle, \langle2, 6\rangle, \langle2, 7\rangle, \langle2, 8\rangle \ldots\}
The Crystal Maze Puzzle: A Constraint Model

- Find $x_1, \ldots, x_8$, all with domain $\{1, \ldots, 8\}$.
- Such That $\text{AllDifferent}(x_1, \ldots, x_8)$,
  
  \begin{align*}
  |x_1 - x_2| & > 1, \ |x_1 - x_3| > 1, \ |x_1 - x_4| > 1, \ |x_1 - x_5| > 1, \\
  |x_2 - x_4| & > 1, \ |x_2 - x_5| > 1, \ |x_2 - x_6| > 1, \\
  |x_3 - x_4| & > 1, \ |x_3 - x_7| > 1 \\
  |x_4 - x_5| & > 1, \ |x_4 - x_7| > 1, \ |x_4 - x_8| > 1, \\
  |x_5 - x_6| & > 1, \ |x_5 - x_7| > 1, \ |x_5 - x_8| > 1, \\
  |x_6 - x_8| & > 1, \\
  |x_7 - x_8| & > 1
  \end{align*}
Examples: Sudoku
Sudoku: A Constraint Model

- **Given** a set of \( \langle \text{cell}, \text{digit} \rangle \) pairs.
- **Find** \( x_1, \ldots, x_{81} \), all with domain \{1, \ldots, 9\}.
- **Such That**
  - **ForEach** \( \langle \text{cell}, \text{digit} \rangle \) pair \( x_{\text{cell}} = \text{digit} \)
  - \text{AllDifferent}(x_1, \ldots, x_9),
  - \text{AllDifferent}(x_{10}, \ldots, x_{18}),
  - \text{AllDifferent}(x_{19}, \ldots, x_{27}),
  - \ldots
Examples: Kakuru
Kakuru: Problem Definition

- [http://4c.ucc.ie/~hcambaza/page1/page7/page7.html](http://4c.ucc.ie/~hcambaza/page1/page7/page7.html)
- The puzzle is to fill a grid with numbers between 1 and 9 such that:
  - The sum of a continuous block of white cell in horizontal (or vertical) direction must be equal to the hint given in the black cell to the left (above).
Kakuru: Problem Definition

• [http://4c.ucc.ie/~hcambaza/page1/page7/page7.html](http://4c.ucc.ie/~hcambaza/page1/page7/page7.html)

• The puzzle is to fill a grid with numbers between 1 and 9 such that:
  – All numbers in a continuous block of cells must be pairwise different.
Kakuru: Instance
Kakuru: Solution
Kakuru: A Constraint Model

\[
\forall i \leq n, \ x_i \in [1, 9] \\
\forall S_j, \ \text{alldifferent} \left( \{x_i | i \in S_j \} \right) \\
\forall S_j, \ h_j \ \sum_{i \in S_j} x_i = h_j
\]
Summary

• The Constrained Optimisation Problem.
• Classes vs. Instances.
• Representing Constraints.
• Constraint Languages.
• Constraint Models & OPL demo for:
  • Crystal Maze
  • Sudoku
  • Kakuru
Summary

• By now you should have a feel for what a constraint satisfaction problem looks like.
• And be starting to understand what modelling is.
• You should also be able to distinguish problems from instances.