In This Lecture

• Lecture 3: Recap
• Graphical Representations.
• Constraint Propagation.
  • Node Consistency.
  • Arc Consistency.
  • Combining Constraint Propagation and Search.
Lecture 3: Recap
Search

• To **search** is to make an (educated) guess among several alternatives…

• …But be prepared to undo that guess and try a different alternative if the guess does not lead to a solution.

• We will focus on **systematic** search. Given sufficient time:
  • If there is a solution, it will be found.
  • If there is no solution, the search space will be exhausted and the search will report that there is no solution.
d-way Vs 2-way Branching

• To begin with, we will focus on d-way.
  • Makes presentation of simpler algorithms easier.

• However, most modern constraint solvers use 2-way branching
  • Why? Stay Tuned.
  • We will return to this branching style later.
The *n*-Queens Puzzle: Row-based Model

- No queens on the same column:
  - \text{AllDifferent}(r_1, r_2, r_3, r_4).
- No queens on the same diagonal:
  - \( r_1 + 1 \neq r_2 \)
  - \( r_1 - 1 \neq r_2 \)
  - ...
Generate and Test

• Brute force: only checks constraints after a complete assignment has been generated.
• Systematic: guaranteed to find a solution if one exists (eventually).
• Never used in practice.
The Backtrack Algorithm

• Systematic: guaranteed to find a solution if one exists.
• Checks a constraint as soon as all of the variable that it constrains are instantiated.
• Can spot dead-ends much faster than G&T.
  • In general, the sooner you can spot a dead-end, the more search you will save.
Backtrack for 4-Queens

This is a domain wipeout. We have tried all possible values for $r_3$ none were feasible, so we must backtrack.
Branch and Bound

• Systematic: guaranteed to find an optimal solution.
• Like BT, checks a constraint as soon as all of the variable that it constrains are instantiated.
• Also maintains the value of the objective associated with the current best solution.
  • Backtracks if current solution cannot be better.
  • Sooner we find a good solution, sooner we can prune.
Constraint Graphs

Dechter 2.1.3
A Graphical Representation

• Why is it useful to view a CSP as a graph?
A Graphical Representation

• Why is it useful to view a CSP as a graph?
  • To guide Heuristics.
  • To guide Propagation.
Binary Constraint Graphs

- Each **node** represents a variable.
- Constraints are represented by **edges**.
- Also known as the **primal** graph representation.

- Variables \( \{x_1, x_2, x_3\} \)
- Domain of each is \( \{0,1\} \)
- Constraints: \( \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\} \)

What does the constraint graph look like?
Binary Constraint Graphs

- Each **node** represents a variable.
- Constraints are represented by **edges**.
- Also known as the **primal** graph representation.

- Variables \( \{x_1, x_2, x_3\} \)
- Domain of each is \( \{0,1\} \)
- Constraints: \( \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\} \)
You’ve Already Seen a Constraint Graph

• For the crystal maze, the problem description & constraint graph coincide.
Non-binary Constraints: Dual Representation

• A means of transforming non-binary instances into binary instances.
• Why? There is a lot of work on algorithms for binary CSP:
  • This is a way of applying it to non-binary CSPs.
Non-binary Constraints: Dual Representation

- Transform each constraint into a dual variable.
  - Has an associated domain of the allowed tuples.
- Add binary constraints between any two dual variables that share an original variable.
  - Constraint insists that the assignments to the two nodes that it connects agree for the shared original variables.
Dual Representation: Example

• $x_1, x_2, x_3, x_4, x_5$
• $D_1, D_2, D_3, D_4, D_5 = \{0,1\}$
• $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5$.  

Original instance

• All three constraints have the same set of satisfying tuples:
  • Which are?

NB Using $D_i$ to denote domain of $x_i$. 
Dual Representation: Example

- $x_1, x_2, x_3, x_4, x_5$
- $D_1, D_2, D_3, D_4, D_5 = \{0,1\}$  
  Original instance
- $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5.$

- All three constraints have the same set of satisfying tuples:
  - Which are?
  - $\{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}$
  - So what next?
Dual Representation: Example

- \(x_1, x_2, x_3, x_4, x_5\)
- \(D_1, D_2, D_3, D_4, D_5 = \{0, 1\}\) (Original instance)
- \(x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5\).

- All three constraints have the same set of satisfying tuples:
  - Which are?
  - \(\{\langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 0\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 1\rangle\}\)
  - So what next?
  - We add **three** variables, each with this domain: \(y_1, y_2, y_3\)
Dual Representation: Example

• $x_1, x_2, x_3, x_4, x_5$
• $D_1, D_2, D_3, D_4, D_5 = \{0, 1\}$
• $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5$.

Original instance

• $y_1, y_2, y_3$
• $D_1, D_2, D_3 = \{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}$
• How many constraints?

Dual representation
**Dual Representation: Example**

- $x_1, x_2, x_3, x_4, x_5$
- $D_1, D_2, D_3, D_4, D_5 = \{0,1\}$
- $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5.$

**Original instance**

**Dual representation**

- $y_1, y_2, y_3$
- $D_1, D_2, D_3 = \{\langle 0,0,1 \rangle, \langle 0,1,0 \rangle, \langle 1,0,0 \rangle, \langle 1,1,0 \rangle, \langle 1,1,1 \rangle\}$
- How many constraints?
  - 3: $c(y_1, y_2), c(y_1, y_3), c(y_2, y_3)$
- What is the extensional representation of these constraints?
Dual Representation: Example

- \( x_1, x_2, x_3, x_4, x_5 \)
- \( D_1, D_2, D_3, D_4, D_5 = \{0,1\} \)
- \( x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5 \).

\( y_1, y_2, y_3 \)

- \( D_1, D_2, D_3 = \{\langle0, 0, 1\rangle, \langle0, 1, 0\rangle, \langle1, 0, 0\rangle, \langle1, 1, 0\rangle, \langle1, 1, 1\rangle\} \)
- \( c(y_1, y_2)\): \{\langle\langle0, 0, 1\rangle, \langle0, 1, 0\rangle\rangle, \langle\langle0, 1, 0\rangle, \langle1, 0, 0\rangle\rangle, \langle\langle1, 0, 0\rangle, \langle0, 0, 1\rangle\rangle, \langle\langle1, 1, 0\rangle, \langle1, 0, 0\rangle\rangle, \langle\langle1, 1, 1\rangle, \langle1, 1, 0\rangle\rangle, \langle\langle1, 1, 1\rangle, \langle1, 1, 1\rangle\rangle\} \)
- \( c(y_1, y_3)\): \{\langle\langle0, 0, 1\rangle, \langle1, 0, 0\rangle\rangle, \langle\langle0, 0, 1\rangle, \langle1, 1, 0\rangle\rangle, \langle\langle0, 0, 1\rangle, \langle1, 1, 1\rangle\rangle, \langle\langle0, 1, 0\rangle, \langle0, 0, 1\rangle\rangle, \langle\langle0, 1, 0\rangle, \langle0, 1, 0\rangle\rangle, \langle\langle0, 1, 0\rangle, \langle0, 1, 1\rangle\rangle, \langle\langle1, 0, 0\rangle, \langle0, 1, 0\rangle\rangle, \langle\langle1, 1, 0\rangle, \langle0, 0, 1\rangle\rangle, \langle\langle1, 1, 0\rangle, \langle0, 1, 0\rangle\rangle, \langle\langle1, 1, 0\rangle, \langle0, 1, 1\rangle\rangle, \langle\langle1, 1, 1\rangle, \langle1, 0, 0\rangle\rangle, \langle\langle1, 1, 1\rangle, \langle1, 1, 0\rangle\rangle, \langle\langle1, 1, 1\rangle, \langle1, 1, 1\rangle\rangle\} \)
- \( c(y_2, y_3)\): (as \( c(y_1, y_2)\)).
Dual Representation: Example

- \(x_1, x_2, x_3, x_4, x_5\)
- \(D_1, D_2, D_3, D_4, D_5 = \{0,1\}\)
- \(x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5\).

- \(y_1, y_2, y_3\)
- \(D_1, D_2, D_3 = \{\langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 0\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 1\rangle\}\)
- \(c(y_1, y_2)\): \{\langle\langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle\rangle, \langle\langle 0, 1, 0\rangle, \langle 1, 0, 0\rangle\rangle, \langle\langle 1, 0, 0\rangle, \langle 0, 0, 1\rangle\rangle, \langle\langle 1, 0, 1\rangle, \langle 1, 1, 0\rangle\rangle, \langle\langle 1, 1, 1\rangle, \langle 1, 1, 1\rangle\rangle\}\)
- \(c(y_1, y_3)\): \{\langle\langle 0, 0, 1\rangle, \langle 1, 0, 0\rangle\rangle, \langle\langle 0, 0, 1\rangle, \langle 1, 1, 0\rangle\rangle, \langle\langle 0, 0, 1\rangle, \langle 1, 1, 1\rangle\rangle, \langle\langle 0, 1, 0\rangle, \langle 0, 0, 1\rangle\rangle, \langle\langle 0, 1, 0\rangle, \langle 0, 1, 0\rangle\rangle, \langle\langle 0, 1, 0\rangle, \langle 0, 1, 0\rangle\rangle, \langle\langle 1, 0, 0\rangle, \langle 1, 1, 0\rangle\rangle, \langle\langle 1, 1, 0\rangle, \langle 0, 0, 1\rangle\rangle, \langle\langle 1, 1, 1\rangle, \langle 1, 0, 0\rangle\rangle, \langle\langle 1, 1, 1\rangle, \langle 1, 1, 0\rangle\rangle, \langle\langle 1, 1, 1\rangle, \langle 1, 1, 1\rangle\rangle\}\)
- \(c(y_2, y_3)\): (as \(c(y_1, y_2)\)).
Dual Representation: Example

- $x_1, x_2, x_3, x_4, x_5$
- $D_1, D_2, D_3, D_4, D_5 = \{0, 1\}$
- $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5$.

- $y_1, y_2, y_3$
- $D_1, D_2, D_3 = \{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\} $
- $c(y_1, y_2)$: \{\langle \langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle \rangle, \\
\langle \langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 1, 0, 0 \rangle \rangle, \\
\langle \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle \rangle\} $
- $c(y_1, y_3)$: \{\langle \langle 0, 0, 1 \rangle, \langle 1, 0, 0 \rangle \rangle, \langle \langle 0, 0, 1 \rangle, \langle 1, 1, 0 \rangle \rangle, \\
\langle \langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \\
\langle \langle 1, 0, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \\
\langle \langle 1, 1, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \\
\langle \langle 1, 1, 1 \rangle, \langle 1, 0, 0 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 1, 1, 0 \rangle \rangle, \\
\langle \langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle \rangle\} $
- $c(y_2, y_3)$: (as $c(y_1, y_2)$).
Can Tuples be Domain Elements?

- $D_1, D_2, D_3 = \{\langle0, 0, 1\rangle, \langle0, 1, 0\rangle, \langle1, 0, 0\rangle, \langle1, 1, 0\rangle, \langle1, 1, 1\rangle\}$
- No: for the purposes of this module we think of domains as atomic (indivisible).
- But we can transform each tuple into an atomic value:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>
Can Tuples be Domain Elements?

- $D_1, D_2, D_3 = \{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}$
- No: for the purposes of this course we think of domains as **atomic** (indivisible).
- But we can transform each tuple into an atomic value:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Dual Representation: Example

- $x_1, x_2, x_3, x_4, x_5$
- $D_1, D_2, D_3, D_4, D_5 = \{0, 1\}$
- $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5$.

- $y_1, y_2, y_3$
- $D_1, D_2, D_3 = \{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}$
- $c(y_1, y_2)$: $\{\langle \langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle \rangle, \langle \langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 1, 0, 0 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 1, 1, 0 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle \rangle\}$
- $c(y_1, y_3)$: $\{\langle \langle 0, 0, 1 \rangle, \langle 1, 0, 0 \rangle \rangle, \langle \langle 0, 0, 1 \rangle, \langle 1, 1, 0 \rangle \rangle, \langle \langle 0, 0, 1 \rangle, \langle 1, 1, 1 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 0, 1, 0 \rangle \rangle, \langle \langle 1, 1, 0 \rangle, \langle 1, 1, 0 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 0, 0, 1 \rangle \rangle, \langle \langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle \rangle\}$
- $c(y_2, y_3)$: (as $c(y_1, y_2)$).
Dual Representation: Example

- $x_1, x_2, x_3, x_4, x_5$
- $D_1, D_2, D_3, D_4, D_5 = \{0,1\}$
- $x_1 + x_2 \neq x_3, x_2 + x_3 \neq x_4, x_3 + x_4 \neq x_5$.

- $y_1, y_2, y_3$
- $D_1, D_2, D_3 = \{1, 2, 4, 6, 7\}$
- $c(y_1, y_2): \{\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 4, 1\rangle, \langle 6, 4\rangle, \langle 7, 6\rangle, \langle 7, 7\rangle\}$
- $c(y_1, y_3): \{\langle 1, 4\rangle, \langle 1, 6\rangle, \langle 1, 7\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 4, 1\rangle, \langle 4, 2\rangle, \langle 6, 1\rangle, \langle 6, 2\rangle, \langle 7, 4\rangle, \langle 7, 6\rangle, \langle 7, 7\rangle\}$
- $c(y_2, y_3): (as \ c(y_1, y_2))$.

- Given a solution, reverse the transformation to find a solution to the original non-binary problem.
Non-binary Constraints: Dual Graph

- Each **node** represents a dual variable (i.e. an original constraint).
- There are **edges** between nodes that share an original variable.
  - Corresponding to the new binary constraints.
Dual Graph: Example

- $y_1, y_2, y_3$
- $D_1, D_2, D_3 = \{\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}$
- $c(y_1, y_2): \{\langle\langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle\rangle, \langle\langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle\rangle, \langle\langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle\rangle, \langle\langle 1, 1, 0 \rangle, \langle 0, 0, 0 \rangle\rangle, \langle\langle 1, 1, 0 \rangle, \langle 1, 0, 0 \rangle\rangle\}$
- $c(y_1, y_3): \{\langle\langle 0, 0, 1 \rangle, \langle 1, 0, 0 \rangle\rangle, \langle\langle 0, 0, 1 \rangle, \langle 1, 1, 0 \rangle\rangle, \langle\langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle\rangle, \langle\langle 0, 1, 0 \rangle, \langle 0, 1, 0 \rangle\rangle, \langle\langle 1, 0, 0 \rangle, \langle 0, 0, 1 \rangle\rangle, \langle\langle 1, 1, 0 \rangle, \langle 0, 1, 0 \rangle\rangle, \langle\langle 1, 1, 0 \rangle, \langle 0, 1, 0 \rangle\rangle, \langle\langle 1, 1, 0 \rangle, \langle 1, 1, 0 \rangle\rangle\}$
- $c(y_2, y_3): (as \ c(y_1, y_2))$. 

![Diagram with nodes y1, y2, y3 connected as a triangle]
Dual Representation: Summary

• Constraints become nodes in the graph, essentially become the variables in an equivalent CSP.
  • Domains are tuples (or atomic equivalents).
Constraint Propagation
How Can We Do Better than Backtrack?

• Backtrack just looks **backwards**.
Constraint Propagation

• We can use constraints for more than just checking.
• We have already seen this:
  • Crystal Maze.
  • Sudoku.

\{1,2,3,4,5,6,7,8\}
Constraint Propagation

• Definition: constraint propagation.
  • Deduction via a subset of the constraints.
  • Deduced information recorded as changes to the problem.
    • Most often: pruning values from domains (i.e. unary constraints).
    • Sometimes: higher arity constraints.
• Local changes form the basis for further deductions.
  • Hence the result of a change is gradually propagated through the constraint network.
• Consider this constraint graph.
  • Let’s say we propagate $c(x_1, x_2)$ and remove one or more domain elements from $D_2$...
Propagation?

• Local changes form the basis for further deductions.
  • Hence the result of a change is gradually propagated through the constraint network.

• Consider this constraint graph.
  • Now we must propagate \( c(x_2, x_3) \) in case we can deduce something new.
  • Let’s say we can remove at least one element from \( D_3 \).
Propagation?

• Local changes form the basis for further deductions.
  • Hence the result of a change is gradually propagated through the constraint network.

• Consider this constraint graph.
  • Similarly, we must propagate $c(x_3, x_4)$.
  • Let’s say we can remove at least one element from $D_4$…
**Propagation?**

- Local changes form the basis for further deductions.
  - Hence the result of a change is gradually *propagated* through the constraint network.
- Consider this constraint graph.
  - Finally, we must propagate $c(x_4, x_5)$.
Consistency Properties

Dechter 3.2
Consistency Properties

• A consistency property holds when constraint propagation of a certain kind reaches a fixpoint.
  • We can deduce nothing new.

• Consistency may be:
  • **local**: one (or a subset of) constraint/arc/variable
  • **global**: all constraints/arcs/variables.
Local Node Consistency

• Given a unary constraint $c(x_i)$, local node consistency holds if:
  • For every $d$ in $D_i$, $c(x_i)$ is satisfied when $x_i = d$.

• Consider $x_1 < 5$, when $D_1 = \{3, 4, 5, 6\}$.
  • Is this node consistent?
Local Node Consistency

• Given a unary constraint $c(x_i)$, local node consistency holds if:
  • For every $d$ in $D_i$, $c(x_i)$ is satisfied when $x_i = d$.
• Consider $x_1 < 5$, when $D_1 = \{3, 4, 5, 6\}$.
  • This is not node consistent.
  • If we propagate $x_1 < 5$ to obtain $D_1 = \{3, 4\}$, it is.
Global Node Consistency

- Holds if local node consistency holds for all unary constraints.
- Can also think of it as follows:
  - Any single variable can be assigned a value from its domain consistently.
- Can be enforced simply by enforcing node consistency for each unary constraint once.
  - Can you see why once is enough?
Binary vs Non-binary Constraints

• Some consistency properties are defined for **binary** constraints only, for historical reasons.

• For now, we will limit ourselves to binary constraints and return to non-binary constraints later.
Arc Consistency

- Preliminaries: we said that a constraint is represented in a constraint graph as an undirected edge/arc between nodes:

  \[
  x_2 \rightarrow x_3 \quad \text{c}(x_2, x_3)
  \]

- We are now going to consider splitting this up into two directional arcs:

  \[
  x_2 \rightarrow x_3 \quad \text{arc}(x_2, x_3)
  \]

  \[
  x_2 \leftarrow x_3 \quad \text{arc}(x_3, x_2)
  \]
Local Arc Consistency

- Given arc \((x_i, x_j)\), local arc consistency holds if:
  - \(c(x_i)\) and \(c(x_j)\) are both node consistent.
  - For each \(d_i\) in \(D_i\), there is at least one corresponding \(d_j\) in \(D_j\), such that \(c(x_i, x_j)\) is satisfied when \(x_i = d_i\) and \(x_j = d_j\).

- Arc revision.
  - Revise arc \((x_1, x_2)\) below:

\[
\text{arc}(x_1, x_2):
\]
\[
x_1, D_1 = \{2, 11, 16\} \quad \xleftarrow{\text{\ }} \quad x_2, D_2 = \{2, 5, 10, 11\}
\]
Local Arc Consistency

• Given arc($x_i$, $x_j$), local arc consistency holds if:
  • $c(x_i)$ and $c(x_j)$ are both node consistent.
  • For each $d_i$ in $D_i$, there is at least one corresponding $d_j$ in $D_j$, such that $c(x_i, x_j)$ is satisfied when $x_i = d_i$ and $x_j = d_j$.

• Arc revision.
  • Revise arc($x_1$, $x_2$) below:

arc($x_1, x_2$):

$x_1, D_1 = \{2, 11, 16\}$ \(\xrightarrow{<} \) \(\bullet \) $x_2, D_2 = \{2, 5, 10, 11\}$
Support

• \{5, 10, 11\} in the domain of \(x_2\) support \{2\} in the domain of \(x_1\).

\[
\text{arc}(x_1, x_2):
\]
\[
x_1, D_1 = \{2\} \quad \bullet \quad \text{<} \quad \bullet \quad x_2, D_2 = \{2, 5, 10, 11\}
\]

• Now revise \(\text{arc}(x_2, x_1)\):

\[
\text{arc}(x_1, x_2):
\]
\[
x_1, D_1 = \{2\} \quad \bullet \quad \text{>} \quad \bullet \quad x_2, D_2 = \{2, 5, 10, 11\}
\]
Why Do We Split A Constraint Into 2 Arcs?

• Efficiency.
• Sometimes we can revise arc($x_i, x_j$) without revising arc($x_j, x_i$).
Global Arc Consistency

• Holds if local arc consistency holds for all arcs.
• Can also think of it as follows:
  • Any assignment to a single variable can be extended to an assignment to two variables consistently.
  • (remember that global node consistency is a precondition).
Global Arc Consistency

- *Cannot* be enforced in general simply by enforcing local arc consistency for all arcs once:
- Revise arc($x_1$, $x_3$), arc($x_3$, $x_1$):

\[
x_1, D_1 = \{5, 6, 7\} \quad \Rightarrow \quad x_2, D_2 = \{5, 6, 7\}
\]

\[
x_3, D_3 = \{5, 6, 7\} \quad = \quad x_1, D_1 = \{5, 6, 7\}
\]
Global Arc Consistency

- Cannot be enforced in general simply by enforcing local arc consistency for all arcs once.
- Revise arc($x_1$, $x_3$), arc($x_3$, $x_1$): No pruning.

\[ x_1, D_1 = \{5, 6, 7\} \quad \Rightarrow \quad x_2, D_2 = \{5, 6, 7\} \]

\[ x_3, D_3 = \{5, 6, 7\} \]
Global Arc Consistency

- *Cannot* be enforced in general simply by enforcing local arc consistency for all arcs once:
  - Revise arc($x_1$, $x_3$), arc($x_3$, $x_1$): No pruning.
  - Now revise arc($x_1$, $x_2$):

\[
x_1, D_1 = \{5, 6, 7\} \quad \rightarrow \quad x_2, D_2 = \{5, 6, 7\}
\]

\[
x_3, D_3 = \{5, 6, 7\}
\]
Global Arc Consistency

- Cannot be enforced in general simply by enforcing local arc consistency for all arcs once:
  - Revise arc($x_1$, $x_3$), arc($x_3$, $x_1$): No pruning.
  - Now revise arc($x_1$, $x_2$):

\[
x_1, D_1 = \{5, 6, 7\} \quad \Rightarrow \quad x_2, D_2 = \{5, 6, 7\}
\]

\[
x_3, D_3 = \{5, 6, 7\}
\]
Global Arc Consistency

- **Cannot** be enforced in general simply by enforcing local arc consistency for all arcs once:
  - Revise arc($x_1, x_3$), arc($x_3, x_1$): No pruning.
  - Now revise arc($x_1, x_2$).
  - What do you notice about arc($x_3, x_1$)?

\[
x_1, D_1 = \{6, 7\} \quad \rightarrow \quad x_2, D_2 = \{5, 6, 7\}
\]

\[
x_3, D_3 = \{5, 6, 7\}
\]
Global Arc Consistency

- *Cannot* be enforced in general simply by enforcing local arc consistency for all arcs once:
  - Revise $\text{arc}(x_1, x_3)$, $\text{arc}(x_3, x_1)$: No pruning.
  - Now revise $\text{arc}(x_1, x_2)$.
  - What do you notice about $\text{arc}(x_3, x_1)$?

\[ x_1, D_1 = \{6, 7\} \quad \rightarrow \quad x_2, D_2 = \{5, 6, 7\} \]

\[ x_3, D_3 = \{5, 6, 7\} \]
Global Arc Consistency

• A great deal of effort has been put into enforcing global arc consistency efficiently.
• We will look at one of these in detail, but first …
Combining Search and Propagation

Forward Checking
Dechter 5.3.1
Combining Search and Propagation

- This is the fundamental way in which systematic constraint solvers work:
  1. Guess an assignment.
  2. Propagate consequences of that assignment.
  3. If all is well (and not done) goto 1.
Forward Checking


• Basic idea:
  • After an assignment to $x_i$, revise arcs from every future variable to $x_i$ once.
  • I.e. Enforce local arc consistency on a subset of the arcs.

• How does this help?
  • Spot dead-ends earlier, reduce search.
The Forward Checking Algorithm

Procedure ForwardChecking(depth)
    Foreach d in $D_{\text{depth}}$
        assign($x_{\text{depth}}$, d)
        consistent = true
    For future = depth+1 To n While consistent
        consistent = revise(arc($x_{\text{future}}$, $x_{\text{depth}}$))
    If (consistent)
        If (depth = n) ShowSolution()
    Else ForwardChecking(depth+1)
    Undo Pruning
The Forward Checking Algorithm

Procedure ForwardChecking(depth)

Foreach d in $D_{depth}$
assign($x_{depth}$, d)
consistent = true
For future = depth + 1 To n While consistent
consistent = revise(arc($x_{future}$, $x_{depth}$))
If (consistent)
  If (depth = n) ShowSolution()
  Else ForwardChecking(depth + 1)
Undo Pruning

• We assume a fixed, increasing instantiation order.
**The Forward Checking Algorithm**

Procedure ForwardChecking(\texttt{depth})

Foreach \texttt{d} in \(D_{\text{depth}}\)

- \texttt{assign}(\texttt{x}_{\text{depth}}, \texttt{d})
- \texttt{consistent} = true

For \texttt{future} = \texttt{depth}+1 To \(n\) While \texttt{consistent}

- \texttt{consistent} = \texttt{revise}(\texttt{arc}(\texttt{x}_{\text{future}}, \texttt{x}_{\text{depth}}))

If (\texttt{consistent})

- If (\texttt{depth} = \(n\)) \texttt{ShowSolution()}
- Else ForwardChecking(\texttt{depth}+1)

Undo Pruning

- After each assignment we revise all arcs from future variables to the current variable.
FC Doesn’t Need to Check Backwards

• Notice that FC does not check against the past variables.
• Consider a past variable $x_p$, and the current variable $x_c$.
• When was $x_p$ assigned, $\text{arc}(x_c, x_p)$ was revised:
  - $x_p = a$
  - $x_c \neq x_p$
  - $\ldots$
  - $x_c \quad D_c = \{a, b, c, d\}$
FC Doesn’t Need to Check Backwards

• When was $x_p$ assigned, $\text{arc}(x_c, x_p)$ was revised:
  
  ● $x_p = a$ \hspace{1cm} $x_c \neq x_p$

  ...

  ● $x_c$ $D_c = \{b, c, d\}$

• This removed all values in $D_c$ incompatible with the assignment to $x_p$.

• When later considering an assignment to $x_c$, all possible assignments are guaranteed to be compatible with the assignment to $x_p$. 
Procedure ForwardChecking(depth)
    Foreach d in D_{depth}
        assign(x_{depth}, d)
    consistent = true
    For future = depth + 1 To n While consistent
        consistent = revise(arc(x_{future}, x_{depth}))
        If (consistent)
            If (depth = n) ShowSolution()
            Else ForwardChecking(depth + 1)
    Undo Pruning

- If all is well, either recurse or we are done.
The Forward Checking Algorithm

Procedure ForwardChecking(depth)

Foreach $d$ in $D_{\text{depth}}$

assign($x_{\text{depth}}$, $d$)

consistent = true

For future = depth+1 To $n$ While consistent

consistent = revise(arc($x_{\text{future}}$, $x_{\text{depth}}$))

If (consistent)

If (depth = $n$) ShowSolution()

Else ForwardChecking(depth+1)

Undo Pruning

• Notice that we now have the overhead of undoing revisions when we backtrack over an assignment.
When Does Forward Checking Beat Backtrack?

• FC is not just BT with constraint checks in a different order:

\[ \bullet x_p = a \quad \text{if} \quad x_c \neq x_p \]

\[ \bullet x_c \quad D_c = \{a\} \]

• This is a dead-end.

• We have saved all this search.
### Forward Checking: 6-Queens

- There is only one queen per row.
- Use one variable per row:
  - \( X = \{ r_1, r_2, r_3, r_4, r_5, r_6 \} \).
  - Each \( D_i \) is \( \{1, 2, 3, 4, 5, 6\} \), denoting each of the available columns.
- No queens on the same column:
  - \( r_1 \neq r_2, r_1 \neq r_3, \ldots \)
- No queens on the same diagonal:
  - \( r_1 + 1 \neq r_2 \)
  - \( r_1 - 1 \neq r_2 \)
  - \( \ldots \)
- Notice: using a **binary** formulation.
Forward Checking: Example

Assign $r_1 = 1$
Forward Checking: Example

Assign $r_1 = 1$

… and propagate
Forward Checking: Example

Assign $r_2 = 3$
Forward Checking: Example

Assign $r_2 = 3$

…and propagate
Forward Checking: Example

Assign $r_3 = 5$
Forward Checking: Example

Assign $r_3 = 5$

…and propagate
Forward Checking: Example

Assign $r_4 = 2$
Forward Checking: Example

- So backtrack.
- Note how the pruning is undone.

Assign $r_4 = 2$

... and propagate:

Wipeout $r_6$
Forward Checking: Example

Assign $r_3 = 6$
Forward Checking: Example

Assign $r_3 = 6$

... and propagate
Forward Checking: Example

Assign $r_4 = 2$
Forward Checking: Example

Assign $r_4 = 2$

... and propagate:

Wipeout $r_5$
Forward Checking: Example

Assign $r_2 = 4$
Forward Checking: Example

Assign $r_2 = 4$
...and propagate
Forward Checking: Example

Assign $r_3 = 2$
Forward Checking: Example

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assign $r_3 = 2$
... and propagate
Forward Checking: Example

Assign $r_4 = 5$
Forward Checking: Example

Assign \( r_4 = 5 \)

...and propagate:

Wipeout \( r_6 \)
Some time Later
Forward Checking and Branch and Bound

• We can combine FC and BB.
• Result is:
  • Better information about the objective:
  • Better propagation of current “best” value.
• Spot dead-ends earlier.
Forward Checking: What You Need to Know

• Complete: Guaranteed to find a solution if one exists.
• Guaranteed to explore a search tree **smaller than or equal-sized** to that of Backtrack.
Summary

• Graphical Representations.
• Constraint Propagation.
  • Node Consistency.
  • Arc Consistency.
  • Combining Constraint Propagation and Search.