Constraint Programming

Lecture 3:
Basic Solution Procedures
In This Lecture

• Lecture 2: Recap
• Search:
  • Definitions
  • Methods
Lecture 2: Recap
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 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.
An Instance & Solution

\[ a = 2, \quad b = 3, \quad c = 1, \quad d = 1, \quad e = 1, \quad f = 1, \quad g = 1, \quad h = 2, \quad i = 1 \]

\[ s_1 = 4, \quad s_2 = 3, \quad s_3 = 3, \quad s_4 = 3, \quad s_5 \ldots s_9 = 0 \]

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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**.
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.
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.
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.
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 Models in Lectures

• Four components:
  1. **Given** parameters
  2. **Find** decision variables
  3. **Such That** constraints
  4. **Min/Maximising** objective
Search: Definitions
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.
Searching Through a Space of Partial Assignments

• Recall that a partial assignment is an assignment to one or more decision variables.
• Begin with an empty assignment, incrementally attempt to extend it into a solution.
• We will assume a backtracking search style:
  • If discover the current partial assignment cannot be extended to a solution (a dead end):
    • Backtrack over last decision made.
    • Try an alternative.
The search for a solution to a CSP may be viewed as exploring a tree.

Root represents the CSP before any search choices.

Choices made correspond to branches in the tree.

Descendants of root node correspond to sub-CSPs:
- Original CSP + partial assignment leaves simplified problem.

Root: original CSP

Branching decisions,
- e.g. \( x = v \)

sub-CSPs
d-way Branching

- There are two common branching styles used.
- In d-way branching, each branch under a parent node represents the assignment of one of d domain values from the domain of a particular variable.
- E.g. if x in \{1, 2, 3\}:

```
x = 1
x = 2
x = 3
```

\{ sub-CSPs \}
2-way/Binary Branching

- Try extending partial assignment with $x = v$ first. If no solution remove $v$ from consideration before continuing.
  - **Qn:** Can you see why this is justified?
2-way/Binary Branching

- Each pair of branches divides the search space in half.
- If we explore the whole tree, we will explore the whole search space.
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 **instantiation order** is the order in which assignments are made.

• The instantiation order can be **fixed** or **dynamic**.
Search: Definitions

• We will speak of a **level** in a search tree.
  • Corresponds to the number of assignments made.
  • Also known as search **depth**.
Search: Definitions

• During search, we will call those variables that have not yet been assigned **future** variables.

• Similarly, those variables that have been assigned are **past** variables.
• Each **branch** represents a partial assignment.

\[
x_1 = a \quad x_2 = a
\]

Diagram:
```
            Root
             |
       x_1    |
       /  \   |
   x_2   x_2
```

25
Search: Definitions

• Each **branch** represents a partial assignment.

\[ x_1 = a \]

\[ x_2 = b \]
Search: Definitions

- Each **branch** represents a partial assignment.

\[ x_1 = b \]
\[ x_2 = a \]
Search: Definitions

- A branch that reaches depth $n$, is a complete assignment.
Modelling the $n$-Queens Problem

A Short Digression
Recall: The $n$-Queens Puzzle

- A problem class, parameterised by $n$.
- Given $n$, put $n$ queens on a $n \times n$ chess board such that no two queens attack each other.
The 4-Queens Puzzle

- Put 4 queens on a $4 \times 4$ chess board such that no two queens attack each other:

```
 Q Q
 Q
 Q
 Q
```

Have a go at modelling this as a CSP
The 4-Queens Puzzle: Variables & Domains

• One option is to have one variable per queen:
  • $q_1, q_2, q_3, q_4$
  • Each domain is $\{1, \ldots, 16\}$, denoting each of the available squares:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
The 4-Queens Puzzle: Constraints

• All constraints look the same. E.g:
  • \textbf{Table}(q_1, q_2) = \{\langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 1, 10 \rangle, \langle 1, 12 \rangle, \langle 1, 14 \rangle, \\
  \langle 1, 15 \rangle, \langle 2, 8 \rangle, \langle 2, 9 \rangle, \langle 2, 11 \rangle, \langle 2, 13 \rangle, \langle 2, 15 \rangle, \\
  \langle 2, 16 \rangle, \langle 3, 5 \rangle, \langle 3, 10 \rangle, \langle 3, 12 \rangle, \langle 3, 13 \rangle, \ldots \}\}

\begin{array}{|c|c|c|c|}
\hline
 1 & 2 & 3 & 4 \\
\hline
 5 & 6 & 7 & 8 \\
\hline
 9 & 10 & 11 & 12 \\
\hline
13 & 14 & 15 & 16 \\
\hline
\end{array}
The 4-Queens Puzzle: Variables & Domains(2)

- There is only one queen per row (or column).
- Another option is to have one variable per row (or column):
  - $r_1, r_2, r_3, r_4$
  - Each domain is \{1, 2, 3, 4\}, denoting each of the available columns:
The 4-Queens Puzzle: Constraints(2)

- No queens on the same column:
  - AllDifferent($r_1$, $r_2$, $r_3$, $r_4$).
- No queens on the same diagonal:
  - $r_1 + 1 \neq r_2$
  - $r_1 - 1 \neq r_2$
  - ...
The 4-Queens Puzzle: Alternatives

• We have seen two alternative models of this puzzle.

• The second is much better than the first.
  • Smaller domains.
  • Less symmetry.

• Recognising this is part of the “art” of modelling.

• These are by no means the only alternatives!
Search Methods:

Generate & Test
Generate and Test

• A simple (but very expensive) method of solving a CSP.
• Each possible complete assignment is generated...
• …and then tested to see if it satisfies all the constraints.
Generate and Test: 4-queens

- This is equivalent to extending each branch of the search tree to level $n$. 

```
Q
Q
Q
Q
```

```
1 1 1
1 1
1
1
```

Root

$r_1$

$r_2$

$r_3$

$r_4$
Generate and Test

\[ Q \]

\[ Q \]

\[ Q \]

\[ Q \]

Root

\( r_1 \)

\( r_2 \)

\( r_3 \)

\( r_4 \)
Generate and Test
### Generate and Test

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

- **Root**
  - $r_1$
  - $r_2$
  - $r_3$
  - $r_4$
Generate and Test

\[
\begin{array}{|c|c|c|c|}
\hline
Q & & & \\
\hline
Q & & & \\
\hline
Q & & & \\
\hline
Q & & & \\
\hline
\end{array}
\]

Root

\[r_1\]

\[r_2\]

\[r_3\]

\[r_4\]
Generate and Test

\[
\begin{array}{c c c c c}
Q & Q & Q & Q \\
Q & & & \\
Q & & & \\
Q & & & \\
\end{array}
\]

Root

\(r_1\)

\(r_2\)

\(r_3\)

\(r_4\)
Generate and Test

\[ \begin{array}{|c|c|c|c|}
\hline
Q & Q & Q & Q \\
\hline
\end{array} \]

Root

\[ r_1 \]

\[ r_2 \]

\[ r_3 \]

\[ r_4 \]
Generate and Test

\[
\begin{array}{|c|c|c|c|}
\hline
Q & & & \\
\hline
Q & & & \\
\hline
Q & & & \\
\hline
Q & & & \\
\hline
\end{array}
\]
Generate and Test

\[ \begin{array}{c c c c c}
Q & & & & \\
Q & & & & \\
& Q & & & \\
Q & & & & \\
\end{array} \]
Generate and Test

\[
\begin{array}{|c|c|}
\hline
Q & Q \\
\hline
Q & Q \\
\hline
\end{array}
\]

Root

\[r_1\]

\[r_2\]

\[r_3\]

\[r_4\]
Generate and Test

Q
Q
Q
Q

Root
r_1
r_2
r_3
r_4
Generate and Test
Generate and Test: Much Later

\[
\begin{array}{ccc}
Q & & Q \\
& Q & \\
Q & & Q \\
\end{array}
\]

\[
\begin{array}{ccc}
& & 1 \\
& 2 & \\
& 3 & \\
\end{array}
\]

Root
\[r_1\]
\[r_2\]
\[r_3\]
\[r_4\]
Generate and Test: Constrained Optimisation Problems

- Generate & Test is even worse in this case.
- We can’t stop when we find a satisfying assignment - it might not be the ‘best’ one.
- We have to examine all complete assignments.
  - Except?

Except: when we know the current solution cannot be improved. E.g. a steel mill solution with no waste.
Generate and Test: What You Need to Know

• 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.
Search Methods:

(Chronological) Backtracking

Dechter 5.2
The Backtrack Algorithm

- Improves on G&T by incrementally extending partial solutions.
- Every time we make an assignment, we check to see if a constraint has been violated.
The Backtrack Algorithm

Procedure Backtrack(*depth*)
    Foreach *d* in *D*\textsubscript{depth}
        assign(*x*\textsubscript{depth}, *d*)
        consistent = true
    For *past* = 1 To *depth*-1 While consistent
        consistent = test(c(*x*\textsubscript{past}, *x*\textsubscript{depth}))
        If (consistent)
            If (*depth* = *n*) ShowSolution()
            Else Backtrack(*depth*+1)

NB Using *D*\textsubscript{depth} to mean the domain of the variable being assigned at level *depth*. 
The Backtrack Algorithm

Procedure Backtrack($depth$)

Foreach $d$ in $D_{depth}$
    assign($x_{depth}$, $d$)
    consistent = true

For $past = 1$ To $depth-1$ While consistent
    consistent = test(c($x_{past}$, $x_{depth}$))
    If (consistent)
        If ($depth = n$) ShowSolution()
        Else Backtrack($depth$+1)

• We assume a fixed, increasing instantiation order.
The Backtrack Algorithm

Procedure Backtrack(depth)
   Foreach d in D_{depth}
      assign(x_{depth}, d)
      consistent = true
   For past = 1 To depth-1 While consistent
      consistent = test(c(x_{past}, x_{depth}))
      If (consistent)
         If (depth = n) ShowSolution()
      Else Backtrack(depth+1)

• After each assignment we check that all of the constraints with assigned variables are satisfied.
The Backtrack Algorithm

Procedure Backtrack(depth)

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

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

\( \text{consistent} = \text{true} \)

For \( \text{past} = 1 \) To \( \text{depth} - 1 \) While \( \text{consistent} \)

\( \text{consistent} = \text{test}(c(\( x_{past} \), \( x_{depth} \))) \)

If (consistent)

If (depth = \( n \)) ShowSolution()

Else Backtrack(depth+1)

• If all is well, either display the solution or recurse.
• This algorithm will display all solutions.
Backtrack for 4-Queens

• Backtrack checks a constraint when all variables in its scope are assigned.
• For AllDiff this means all four variables.
  • We will see how to do much better with AllDiff later.
• So, for the following we will assume that we are using binary not-equals constraints.
Backtrack for 4-Queens

- BT backtracks here immediately: no two queens on the same column.
- Already a huge saving over G&T.
Backtrack for 4-Queens

Q
Q
Backtrack for 4-Queens
Backtrack for 4-Queens

Root

\( r_1 \)

\( r_2 \)

\( r_3 \)

\( r_4 \)
Backtrack for 4-Queens
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.
Backtrack for 4-Queens
Backtrack for 4-Queens

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

Root

- $r_1$
- $r_2$
- $r_3$
- $r_4$
Backtrack for 4-Queens

```
Q   Q
Q   Q
```

Root

\[ r_1 \]

\[ r_2 \]

\[ r_3 \]

\[ r_4 \]
Backtrack for 4-Queens
Backtrack for 4-Queens

Wipeout
Backtrack for 4-Queens
Backtrack for 4-Queens

Q
Q
Q

Wipeout

Root

\(r_1\)

\(r_2\)

\(r_3\)

\(r_4\)
Backtrack for 4-Queens

\[ \begin{array}{|c|c|c|c|c|} 
\hline
Q & & & & \\
Q & & & & \\
\hline
\end{array} \]

Root

\[ r_1 \]

\[ r_2 \]

\[ r_3 \]

\[ r_4 \]
Backtrack for 4-Queens

Root

\( r_1 \)

\( r_2 \)

\( r_3 \)

\( r_4 \)
Backtrack for 4-Queens

\[
\begin{array}{|c|c|c|c|c|}
\hline
Q & & & & \\
Q & & & & \\
\hline
\end{array}
\]

Root

\[r_1\]

\[r_2\]

\[r_3\]

\[r_4\]
Backtrack for 4-Queens

Q
Q
Q
Q

Q

1 2 4
1 4
1

Root

r_1

r_2

r_3

r_4
Backtrack for 4-Queens
Backtrack for 4-Queens

Every time we noticed failure above level 4, we made a significant saving over G&T
Backtrack: What You Need to Know

• Systematic: guaranteed to find a solution if one exists.
• Checks a constraint as soon as all of the variables 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.
Search Methods:

Solving COPs
Branch and Bound
Branch & Bound

• Augments Backtrack.
• When we reach a solution we record the value of the objective.
• As soon as a partial assignment cannot improve over the current “best” value, we backtrack.
The Branch and Bound Algorithm

Procedure BranchAndBound(depth)
    Foreach d in D_{depth}
        assign(x_{depth}, d)
    objVal = calculateObjective()
    If (objVal < α)
        consistent = true
        For past = 1 To depth-1 While consistent
            consistent = test(c(x_{past}, x_{depth}))
        If (consistent)
            If (depth = n)
                α = objVal
                ShowSolution()
            Else BranchAndBound(depth+1)

• Assume α is global, and initialised to $\infty$.
• Assume that we are minimising.
The Branch and Bound Algorithm

Procedure BranchAndBound(depth)

- For each $d$ in $D_{depth}$
  - assign($x_{depth}$, $d$)
  - objVal = calculateObjective()
  - If (objVal < $\alpha$)
    - consistent = true
    - For past = 1 To depth-1 While consistent
      - consistent = test(c($x_{past}$, $x_{depth}$))
    - If (consistent)
      - If (depth = $n$)
        - $\alpha$ = objVal
        - ShowSolution()
      - Else BranchAndBound(depth+1)

- Again, assume a fixed increasing instantiation order.
The Branch and Bound Algorithm

**Procedure** BranchAndBound\((\text{depth})\)

- **Foreach** \(d\) in \(D_{\text{depth}}\)
  - assign\((x_{\text{depth}}, d)\)
  - \(\text{objVal} = \text{calculateObjective()}\)
  - **If** \((\text{objVal} < \alpha)\)
    - \(\text{consistent} = \text{true}\)
    - For \(\text{past} = 1\) To \(\text{depth}-1\) While \(\text{consistent}\)
      - \(\text{consistent} = \text{test}(c(x_{\text{past}}, x_{\text{depth}}))\)
    - **If** \((\text{consistent})\)
      - **If** \((\text{depth} = n)\)
        - \(\alpha = \text{objVal}\)
        - ShowSolution()
      - Else BranchAndBound\((\text{depth}+1)\)

- Given an assignment, compare current objective with best solution.
- The more accurate \(\text{calculateObjective()}\) is, sooner we spot dead-ends.
The Branch and Bound Algorithm

Procedure BranchAndBound(depth)

Foreach $d$ in $D_{depth}$
    assign($x_{depth}, d$)
    objVal = calculateObjective()
    If (objVal < $\alpha$)
        consistent = true
        For past = 1 To depth-1 While consistent
            consistent = test(c($x_{past}, x_{depth}$))
        If (consistent)
            If (depth = $n$)
                $\alpha = objVal$
                ShowSolution()
            Else BranchAndBound(depth+1)

• If objective still viable, check constraints as before.
The Branch and Bound Algorithm

Procedure BranchAndBound(\textbf{depth})

Foreach \textbf{d} in \textbf{D}_{\text{depth}}

assign(\textbf{x}_{\text{depth}}, \textbf{d})

\textbf{objVal} = \text{calculateObjective()}

If (\textbf{objVal} < \alpha)

\hspace{1cm} \textbf{consistent} = true

\hspace{1cm} For \textbf{past} = 1 \text{ To } \text{depth}-1 \text{ While } \textbf{consistent}

\hspace{2cm} \textbf{consistent} = \text{test}(c(\textbf{x}_{\text{past}}, \textbf{x}_{\text{depth}}))

\hspace{1cm} If (\textbf{consistent})

\hspace{2cm} \bullet \hspace{1cm} If (\textbf{depth} = \textit{n})

\hspace{3cm} \alpha = \textbf{objVal}

\hspace{3cm} ShowSolution()

\hspace{2cm} \bullet \hspace{1cm} Else BranchAndBound(\textbf{depth}+1)

• If all is well, and have reached depth \textit{n}, update \alpha.
• Otherwise, recurse.
A Small Steel Mill Instance

- The mill can make two slab sizes: 4 and 5.
- There are three orders: $a$, $b$, $c$.
Steel Mill Instance: COP

- Three slab variables: \( s_1, s_2, s_3 \).
- Domain: \( \{0, 4, 5\} \)
- The order matrix:

<table>
<thead>
<tr>
<th></th>
<th>Order\textsubscript{a}</th>
<th>Order\textsubscript{b}</th>
<th>Order\textsubscript{c}</th>
<th>Weighted sum ( \leq s_1 )</th>
<th>Weighted sum ( \leq s_2 )</th>
<th>Weighted sum ( \leq s_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slab\textsubscript{1}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
</tr>
<tr>
<td>Slab\textsubscript{2}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
</tr>
<tr>
<td>Slab\textsubscript{3}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
<td>( \sum = 1 )</td>
</tr>
</tbody>
</table>
Steel Mill Instance: COP

• We ignore the colour matrix for this simple example.

• Objective:
  • Minimise $s_1 + s_2 + s_3$. 
Branch and Bound: Small Steel Mill Instance

- At this point the objective is 0.
- This is clearly less than $\alpha = \infty$.
- But of course the sum constraints cannot be satisfied.
At this point the objective is 4.
This is less than $\alpha = \infty$.
But the sum constraints still cannot be satisfied.
Branch and Bound: Small Steel Mill Instance

- At this point the objective is 5.
- This is less than $\alpha = \infty$.
- But the sum constraints still cannot be satisfied.

\[
\begin{align*}
\text{Root} & : s_1, s_2, s_3 \\
O_{a,1} & : O_{a,2}, O_{a,3} \\
O_{b,1} & : O_{b,2}, O_{b,3} \\
O_{c,1} & : O_{c,2}, O_{c,3}
\end{align*}
\]
Branch and Bound: Small Steel Mill Instance

- At this point the objective is 4.
- This is less than $\alpha = \infty$.
- But the sum constraints still cannot be satisfied.
What does this repeated failure tell us?

- At this point the objective is 8.
- This is less than $\alpha = \infty$.
- But the sum constraints still cannot be satisfied.
Branch and Bound: Small Steel Mill Instance

- At this point the objective is 9.
- This is less than $\alpha = \infty$.
- There is now a solution:

<table>
<thead>
<tr>
<th></th>
<th>Order$_a$</th>
<th>Order$_b$</th>
<th>Order$_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slab$_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Slab$_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Slab$_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Set $\alpha = 9$
Branch and Bound: Small Steel Mill Instance

- New value of $\alpha$ means we must backtrack to $s_3$: wipeout.
Branch and Bound: Small Steel Mill Instance

• Value of objective is $5 < \alpha = 9$. 
Branch and Bound: Small Steel Mill Instance

- Value of objective is $9 = \alpha$:
  - Backtrack (assuming not looking for all optimal solutions).
• Value of objective is $10 > \alpha = 9$
  • Backtrack: wipeout.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $4 < \alpha = 9$
  - But no feasible solutions.
Branch and Bound:
Small Steel Mill Instance

- Value of objective is $8 < \alpha = 9$
  - But no feasible solutions.
Branch and Bound:
Small Steel Mill Instance

• Value of objective is $9 = \alpha$
• Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $8 < \alpha = 9$
- But no feasible solutions.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $12 > \alpha = 9$
  - Backtrack.
Branch and Bound: Small Steel Mill Instance

• Value of objective is $13 > \alpha = 9$
  • Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $9 = \alpha$
- Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $5 < \alpha = 9$
  - But no feasible solutions.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $9 = \alpha$
- Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $10 > \alpha = 9$
  - Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $9 = \alpha$
- Backtrack.
Branch and Bound: Small Steel Mill Instance

- Value of objective is $10 > \alpha = 9$
  - Backtrack.
- We have exhausted the search space.
- This is a proof that 9 is the optimum amount of steel to create.
- We have an assignment of orders to slabs for this optimum.
Branch and Bound: What You Need to Know

• 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.
Lecture 3: Summary

• Search:
  • d-way vs 2-way

• Methods:
  • Generate & Test
  • Chronological Backtracking.
  • Branch and Bound.