Operations Research Techniques in Constraint Programming

Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University

ACP Summer School on Theory and Practice of Constraint Programming
September 24-28, 2012, Wrocław, Poland

Motivation

Benefits of CP
- Modeling power
- Inference methods
- Advanced search
- Exploits local structure

Benefits of OR
- Optimization algorithms
- Relaxation methods
- Duality theory
- Exploits global structure

Integrated methods can combine these complementary strengths
Can lead to several orders of magnitude of computational advantage

Some additional references

- Conference series CPAIOR
  - integration of techniques from CP, AI, and OR
  - http://www.andrew.cmu.edu/user/vanhoeve/cpaior/
  - online master classes/tutorials

- Tutorials by John Hooker
  - CP summer school 2011: ‘Integrating CP and mathematical programming’
  - http://ba.gsia.cmu.edu/jnh/slides.html
Outline

- Global constraint propagation
  - matching theory for *alldifferent*
  - network flow theory for *cardinality* constraint
- Integrating relaxations
  - Linear Programming relaxation
  - Lagrangean relaxation
- Decomposition methods
  - logic-based Benders
  - column generation

Matchings in graphs

- **Definition:** Let $G = (V, E)$ be a graph with vertex set $V$ and edge set $E$. A matching in $G$ is a subset of edges $M$ such that no two edges in $M$ share a vertex.
- A **maximum matching** is a matching of maximum size
- **Definition:** An $M$-augmenting path is a vertex-disjoint path with an odd number of edges whose endpoints are $M$-free
- **Theorem:** Either $M$ is a maximum-size matching, or there exists an $M$-augmenting path

Finding a maximum matching

- The augmenting path theorem can be used to iteratively find a maximum matching in a graph $G$:
  - given $M$, find an $M$-augmenting path $P$
  - if $P$ exists, augment $M$ along $P$ and repeat
  - otherwise, $M$ is maximum
- For a **bipartite** graph $G = (V_1, V_2, E)$, an $M$-augmenting path can be found in $O(|E|)$ time
  - finding a maximum matching can then be done in $O(|V_1| \cdot |E|)$, as we need to compute at most $|V_1|$ paths (assume $|V_1| \leq |V_2|$)
  - this can be improved to $O(\sqrt{|V_1| \cdot |E|})$ time  
  - [Hopcroft & Karp, 1973]
- For general graphs this is more complex, but still tractable
  - can be done in $O(\sqrt{|V| \cdot |E|})$ time  
  - [Micali & Vazirani, 1980]
Alldifferent Propagation

• Goal: establish domain consistency on \textit{alldifferent}
  – Guarantee that each remaining domain value participates in at least one solution
  – Can we do this in polynomial time?

• We already saw that the decomposition is not sufficient to establish domain consistency

  \[ x_1 \in \{a,b\}, x_2 \in \{a,b\}, x_3 \in \{a,b,c\} \]
  \[ x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_3 \text{ \ versus \ } \text{alldifferent}(x_1,x_2,x_3) \]

Value Graph Representation

• Definition: The \textit{value graph} of a set of variables \(X\) is a bipartite graph \((X, D, E)\) where
  – node set \(X\) represents the variables
  – node set \(D\) represents the union of the variable domains
  – edge set \(E\) is \(\{ (x,d) \mid x \in X, d \in D(x) \} \)

• Example:

  \text{alldifferent}(x_1,x_2,x_3)
  \[ x_1 \in \{a,b\} \]
  \[ x_2 \in \{a,b\} \]
  \[ x_3 \in \{b,c\} \]

From alldifferent to matchings

\textbf{Observation} [Régin, 1994]:
solution to \textit{alldifferent}(X) \iff
matching in value graph covering \(X\)

Example:

\[ x_1 \in \{a,b\}, x_2 \in \{a,b\}, x_3 \in \{b,c\} \]
\text{alldifferent}(x_1,x_2,x_3)

Domain consistency for \textit{alldifferent}:
remove all edges (and corresponding domain values) that are not in any maximum matching
**Filtering Algorithm**

1. Verify consistency of the constraint
   - find maximum matching \( M \) in value graph
   - if \( M \) does not cover all variables: inconsistent
2. Verify consistency of each edge
   - for each edge \( e \) in value graph:
     - fix \( e \) in \( M \), and extend \( M \) to maximum matching
     - if \( M \) does not cover all variables: remove \( e \) from graph

Total runtime: \( O(\sqrt{|X| \cdot |E|^2}) \)
- Establishes domain consistency in polynomial time
- But not efficient in practice... can we do better?

---

**A useful theorem**

- **Theorem** [Petersen, 1891] [Berge, 1970]: Let \( G \) be graph and \( M \) a maximum matching in \( G \). An edge \( e \) belongs to a maximum-size matching if and only if
  - it either belongs to \( M \)
  - or to an even \( M \)-alternating path starting at an \( M \)-free vertex
  - or to an \( M \)-alternating circuit

---

**A Better Filtering Algorithm**

1. compute a maximum matching \( M \): covering all variables \( X \)
2. direct edges in \( M \) from \( X \) to \( D \), and edges not in \( M \) from \( D \) to \( X \)
3. compute the strongly connected components (SCCs)
4. edges in \( M \), edges within SCCs and edges on path starting from \( M \)-free vertices are all consistent
5. all other edges are not consistent and can be removed

- SCCs can be computed in \( O(|E|+|V|) \) time [Tarjan, 1972]
- consistent edges can be identified in \( O(|E|) \) time
- filtering in \( O(|E|) \) time
Important aspects

- Separation of consistency check \( O(\sqrt{|X| \cdot |E|}) \) and domain filtering \( O(|E|) \)
- Incremental algorithm
  - Maintain the graph structure during search
  - When \( k \) domain values have been removed, we can repair the matching in \( O(km) \) time
  - Note that these algorithms are typically invoked many times during search / constraint propagation, so being incremental is very important in practice

Network Flows

Let \( G=(V,A) \) be a directed graph with vertex set \( V \) and arc set \( A \). To each arc \( a \in A \) we assign a capacity function \( [d(a),c(a)] \) and a weight function \( w(a) \).

Let \( s,t \in V \). A function \( f: A \to \mathbb{R} \) is called an \( s-t \) flow (or a flow) if

- \( f(a) \geq 0 \) for all \( a \in A \)
- \( \sum_{a \text{ enters } v} f(a) = \sum_{a \text{ leaves } v} f(a) \) for all \( v \in V \) (flow conservation)
- \( d(a) \leq f(a) \leq c(a) \) for all \( a \in A \)

Define the cost of flow \( f \) as \( \sum_{a \in A} w(a)f(a) \). A minimum-cost flow is a flow with minimum cost.

Example: Network flow for alldifferent

Fact: matching in bipartite graph \( \iff \) integer flow in directed bipartite graph

Step 1: direct edges from \( X \) to \( D(X) \)
Step 2: add a source \( s \) and sink \( t \)
Step 3: connect \( s \) to \( X \), and \( D(X) \) to \( t \)
Step 4: add special arc \((t,s)\)

all arcs have capacity \([0,1]\) and weight 0
except arc \((t,s)\) with capacity \([0, \min(|X|,|D(X)|)]\)
Cardinality constraints

- The global cardinality constraint restricts the number of times certain values can be taken in a solution.
- Example: We need to assign 75 employees to shifts. Each employee works one shift. For each shift, we have a lower and upper demand.

<table>
<thead>
<tr>
<th>Shift</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>10</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td>14</td>
<td>14</td>
<td>20</td>
<td>14</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ D(x_i) = \{1, 2, 3, 4, 5, 6\} \text{ for } i = 1, 2, \ldots, 75 \]

\[ \text{gcc}(x_1, \ldots, x_{75}, \text{min}, \text{max}) \]

Filtering for cardinality constraints

**Definition:** Let \( X \) be a set of variables with \( D(x) \subseteq V \) for all \( x \in X \) (for some set \( V \)). Let \( L \) and \( U \) be vectors of non-negative integers over \( V \) such that \( L(v) \leq U(v) \) for all \( v \in V \). The \( \text{gcc}(X, L, U) \) is defined as the conjunction

\[ \land_{v \in V} \left( L(v) \leq \sum_{x \in X} (x=v) \leq U(v) \right) \]

**Questions:**
1. Can we determine in polynomial time whether the constraint is consistent (satisfiable)?
2. Can we establish domain consistency (remove all inconsistent domain values) in polynomial time?

Network representation

- **Observation** [Regin, 1996]: Solution to \( \text{gcc} \) is equivalent to particular network flow
  - similar to bipartite network for \( \text{alldifferent} \)
  - node set defined by variables and domain values, one source \( s \) and one sink \( t \)
  - define arc \((x,v)\) for all \( x \in X, v \in D(x) \) with capacity \([0,1]\)
  - define arcs from \( s \) to \( x \) for all \( x \in X \) with capacity \([1,1]\)
  - define arcs from \( v \) to \( t \) for all \( v \in V \) with capacity \([U(v),L(v)]\)
- Feasible flow corresponds to solution to \( \text{gcc} \)
- **Note:** If \( L(v)=0, U(v)=1 \) for all \( v \in V \) then \( \text{gcc} \) is equivalent to \( \text{alldifferent} \)
Filtering for cardinality constraints

- Determining consistency: compute network flow
  - Using Ford & Fulkerson's augmenting path algorithm, this can be done in $O(mn)$ time for $n$ is number of variables, $m$ is number of edges in the graph
  - Can be improved to $O(m\sqrt{n})$ [Quimper et al., 2004]

- Naive domain consistency
  - Fix flow of each arc to 1, and apply consistency check. Remove arc if no solution. $O(m^2\sqrt{n})$ time.

- More efficient algorithm: use residual network

Residual network

Given network $G=(V,A)$ and a flow $f$ in $G$, the residual network $G_f$ is defined as $(V,A_f)$ where for all $a \in A$,
- $a \in A_f$ if $f(a) < c(a)$ with capacity $\max\{d(a) - f(a), 0\}$, $c(a) - f(a)$ and weight $w(a)$
- $a^{-1} \in A_f$ if $f(a) > d(a)$ with capacity $\{0, f(a) - d(a)\}$ and weight $-w(a)$

New capacities express how much more flow we can put on arc $a$ or subtract from it (via $a^{-1}$)
Example for GCC

\[ \begin{align*}
D(x_1) &\quad D(x_2) &\quad D(x_3) &\quad D(x_4) &\quad D(x_5) \\
1 &\quad 1.2 &\quad 1.2 &\quad 2 &\quad 2.3 &\quad 3.4
\end{align*} \]

### Benefits of residual network

- Fact from flow theory:
  
  **Theorem 9.** Let \( G \) be a graph and \( f \) a feasible flow in \( G \). An arc belongs to some feasible flow in \( G \) if and only if it belongs to \( f \) or both of its endpoints belong to the same SCC of the residual graph of \( G \) with respect to \( f \).

- We can compute all strongly connected components in \( O(m+n) \) time [Tarjan, 1972]

- Therefore, given a consistent gcc, domain consistency can be established in \( O(m) \) time

- Other benefits
  - maintain data structures and flow incrementally
  - compute initial network flow only once at the root of the search tree (similar to the alldifferent algorithm)

### Optimization Constraints

- In the CP literature, ‘optimization’ constraints refer to constraints that represent a structure commonly identified with optimization
  - usually linked to the objective function (e.g., minimize cost)
  - sometimes standalone structure (budget limit, risk level, etc.)

- For any constraint, a weighted version can be obtained by applying a weight measure on the variable assignments, and restricting the total weight to be within a threshold
GCC with costs

- The classical weighted version of the gcc is obtained by associating a weight \( w(x,v) \) to each pair \( x \in X, v \in V \).
- Let \( z \) be a variable representing the total weight. Then
  \[
  \text{cost\_gcc}(X, L, U, z, w) = \text{gcc}(X, L, U) \land \sum_{x \in X \& x=v} w(x,v) \leq z
  \]
- In other words, we restrict the solutions to those that have a weight at most \( \max(D(z)) \).

Domain filtering for weighted gcc

1. Determine consistency of the constraint
2. Remove all domain values from \( X \) that do not belong to a solution with weight \( \leq \max(D(z)) \)
3. Filter domain of \( z \)
   - i.e., increase \( \min(D(z)) \) to the minimum weight value over all solutions, if applicable

Determining consistency of cost\_gcc

- Once again, we exploit the correspondence with a (weighted) network flow [Regin 1999, 2002]:
  A solution to cost\_gcc corresponds to a weighted network flow with total weight \( \leq \max(D(z)) \)
- We can test consistency of the cost\_gcc by computing a minimum-cost flow
Time complexity

- A minimum-cost flow can be found with the classical 'successive shortest paths' algorithm of Ford & Fulkerson
  - The flow is successively augmented along the shortest path in the residual network
  - Finding the shortest path takes $O(m + n \log n)$ time (for $m$ edges, $n$ variables)
  - In general, this yields a pseudo-polynomial algorithm, as it depends on the cost of the flow. However, we compute at most $n$ shortest paths (one for each variable)
  - Overall running time is $O(n(m + n \log n))$ time
- Naive domain consistency in $O(nm(m + n \log n))$

Taking advantage of residual graph

Theorem (e.g., Ahuja et al., 1993)\]

For each arc $(x, d)$ with $d$ in $D(x)$ for which $f(x,d)=0$, we compute the shortest $d$-$x$ path $P$ in the residual graph

- If $\text{cost}(f) + \text{cost}(P) + \text{cost}(x,d) > \max(D(z))$, remove $d$ from $D(x)$
  - Gives domain consistency in $O((m-n)(m + n \log n))$ time
  - Can be improved to $O(\min(n, |V|)(m + n \log n))$ time by computing all shortest paths from variable (or value) vertices in one shot

Domain consistency again

- For each arc $(x, d)$ with $d$ in $D(x)$ for which $f(x,d)=0$, we compute the shortest $d$-$x$ path $P$ in the residual graph
- Maintain flow incrementally. Upon $k$ domain changes, update flow in $O(k(m + n \log n))$ time
Other optimization constraints

- Weighted network flows have been applied to several other global constraints
  - weighted alldifferent
  - soft alldifferent
  - soft cardinality constraint
  - soft regular constraint
  - cardinality constraints in weighted CSPs
  - ...
  
  see [v. H. “Over-Constrained Problems”, 2011] for an overview

- Very powerful and generic technique for handling global constraints

Outline

- Global constraint propagation
  - matching theory for alldifferent
  - network flow theory for cardinality constraint

- Integrating relaxations
  - Linear Programming relaxation
  - Lagrangean relaxation

- Decomposition methods
  - logic-based Benders
  - column generation

Integrating relaxations

- Linear Programming
  - duality
  - LP-based domain filtering
  - application: routing

- Lagrangean Relaxations
  - domain filtering
  - application: routing
Linear Programming

- LP model is restricted to linear constraints and continuous variables
- Linear programs can be written in the following standard form:

\[
\begin{align*}
\min & \quad c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \\
\text{subject to} & \quad a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n = b_1 \\
& \quad a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n = b_2 \\
& \quad \vdots \\
& \quad a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n = b_m \\
& \quad x_1, \ldots, x_n \geq 0
\end{align*}
\]

or, using matrix notation:

\[
\min \{ c^T x \mid Ax = 0, x \geq 0 \}
\]

Benefits of Linear Programming

- Solvable in polynomial time
  - very scalable (millions of variables and constraints)
- Many real-world applications can be modeled and solved using LP
  - from production planning to data mining
- LP models are very useful as relaxation for integer decision problems
  - LP relaxation can be strengthened by adding constraints (cuts) based on integrality
- Well-understood theoretical properties
  - e.g., duality theory

Solving LP models: Example

Maximize

8x_1 + 5x_2

Subject to

x_1 + x_2 \leq 10
4x_1 - x_2 \leq 0
x_1, x_2 \geq 0

Optimal Solution: x_1 = 2, x_2 = 8 with value 56
### Solving LP models: Standard form

Maximize

\[ 8x_1 + 5x_2 \]

Subject to

\[ x_1 + x_2 \leq 10 \]
\[ 4x_1 - x_2 \leq 0 \]
\[ x_1, x_2 \geq 0 \]

Minimize

\[ -8x_1 - 5x_2 \]

Subject to

\[ x_1 + x_2 \leq 10 \]
\[ 4x_1 - x_2 \leq 0 \]
\[ x_1, x_2 \geq 0 \]

Minimize

\[ -8x_1 - 5x_2 \]

Subject to

\[ x_1 + x_2 + x_3 = 10 \]
\[ 4x_1 - x_2 + x_4 = 0 \]
\[ x_1, x_2, x_3, x_4 \geq 0 \]

\[ \min \{ c^T x \mid Ax = b, x \geq 0 \} \]

(x_2 and x_4 are called 'slack' variables)

### Algebraic analysis

- Rewrite \( Ax = b \) as \( A_B x_B + A_N x_N = b \), where \( A = [A_B \mid A_N] \)
- \( A_B \) is any set of \( m \) linearly independent columns of \( A \)
  - these form a basis for the space spanned by the columns
- We call \( x_B \) the basic variables and \( x_N \) the non-basic variables

- Solving \( Ax = b \) for \( x_B \) gives \( x_B = A_B^{-1} b - A_B^{-1} A_N x_N \)
- We obtain a basic solution by setting \( x_N = 0 \)
  - so, \( x_B = A_B^{-1} b \)
  - this is a basic feasible solution if \( x_B \geq 0 \)

### Example

Minimize

\[ -8x_1 - 5x_2 \]

Subject to

\[ x_1 + x_2 + x_3 = 10 \]
\[ 4x_1 - x_2 + x_4 = 0 \]
\[ x_1, x_2, x_3, x_4 \geq 0 \]
Optimality condition

- Recall solution: \( x_B = A_B^{-1}b - A_B^{-1}A_N x_N \)
- Express objective \( c_B x_B + c_N x_N \) in terms of non-basic variables:
  \[ c_B A_B^{-1}b + (c_N - c_B A_B^{-1}A_N) x_N \]
  vector of reduced costs
- Since \( x_N \geq 0 \), basic solution \( (x_B, 0) \) is optimal if reduced costs are nonnegative
- (In fact, the Simplex method moves from one basic solution to another improving one until all reduced costs are nonnegative)

LP Duality

Every (primal) LP model has an associated dual model:

(P) \[
\begin{align*}
\min & \quad c^T x \\
\text{s.t.} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

(D) \[
\begin{align*}
\max & \quad \lambda^T b \\
\text{s.t.} & \quad \lambda^T A \leq c \\
& \quad \lambda \geq 0
\end{align*}
\]

- Each constraint in (P) has an associated dual variable in (D)
- these are also called the shadow prices of the constraints
- The dual of the dual is the primal
- Every feasible solution to an LP gives a bound on its dual
- If (P) is feasible, then optimum(P) = optimum(D)
  (this is called strong duality)

LP dual for standard form

(P) \[
\begin{align*}
\min & \quad c_B^T x_B + c_N^T x_N \\
\text{s.t.} & \quad A_B x_B + A_N x_N = b \\
& \quad x_B, x_N \geq 0
\end{align*}
\]

(D) \[
\begin{align*}
\max & \quad \lambda^T b \\
\text{s.t.} & \quad \lambda^T A_B \leq c_B \quad (x_B) \\
& \quad \lambda^T A_N \leq c_N \quad (x_N) \\
& \quad \lambda \text{ free}
\end{align*}
\]

If \( (x_B, 0) \) solves the primal, then \( \lambda^T = c_B A_B^{-1} \) solves the dual

Recall: reduced cost vector is
\[ c_N - c_B A_B^{-1}A_N = c_N - \lambda^T A_N \]
In other words, the reduced cost for \( x_i \) is
\[ \bar{c}_i = c_i - \sum_j \lambda_j a_{ij} \]
**Economic Interpretation**

- **Reduced costs**
  - by definition, represents the marginal change in the objective if variable enters the basis
  - changing \( x_i \) by \( \Delta \) will change objective by at least \( c^T \Delta \)

- **Shadow prices**
  - by dual perspective, represents the marginal change in the objective if the RHS changes
  - changing \( b_j \) by \( \Delta \) will change objective by at least \( \lambda \Delta \)

\[
\begin{align*}
\textbf{(P)} & \quad \min & c^T x \\
\text{s.t.} & \quad Ax \geq b \\
& \quad x \geq 0 \\
\end{align*}
\]

\[
\begin{align*}
\textbf{(D)} & \quad \max & \lambda^T b \\
\text{s.t.} & \quad \lambda^T A \leq c \\
& \quad \lambda \geq 0 \\
\end{align*}
\]

**Graphical representation of shadow price**

What happens if we increase the RHS of \( x_1 + x_2 \leq 10 \) with 1 unit to \( x_1 + x_2 \leq 11 \)?

- Basis remains optimal
- Objective decreases by 5.6 to value -61.6

So, the shadow price of this constraint is -5.6

**LP-based domain filtering**

- Suppose we have a LP relaxation available for our problem
  \[
  \min \{ c^T x \mid Ax = b, x \geq 0 \}
  \]

- Can we establish 'LP bounds consistency' on the domains of the variables?

  For each variable \( x_i \):
  - change objective to \( \min x_i \) and solve LP: lower bound \( \text{LB}_i \)
  - change objective to \( \max x_i \) and solve LP: upper bound \( \text{UB}_i \)

  \[ x_i \in [\text{LB}_i, \text{UB}_i] \]

- Very time-consuming (although it can pay off, e.g., in nonlinear programming problems)
LP-based domain filtering

• Instead of min/max of each variable, exploit reduced costs as more efficient approximation
  [Focacci, Lodi, and Milano, 1999, 2002]

• In the following, we assume for simplicity an ‘optimization constraint’ of the form:

\[
\text{opt. } \mathcal{L}(x_1, \ldots, x_n, z, c) = \{(d_1, \ldots, d_n) \in C(x_1, \ldots, x_n) : \\
\forall i \in D(x_i), d \in D(z), \sum_{i=1}^{n} c_{i,d_i} \leq d\}.
\]

Creating an LP relaxation

• Create mapping between linear model and CP model by introducing binary variables \( y_{ij} \) for all \( i \in \{1, \ldots, n\} \) and \( j \in D(x_i) \) such that

\[
x_i = j \Leftrightarrow y_{ij} = 1 \\
x_i \neq j \Leftrightarrow y_{ij} = 0
\]

• To ensure that each variable \( x_i \) is assigned a value, we add the following constraints to the linear model:

\[
\sum_{j \in D(x_i)} y_{ij} = 1 \text{ for } i = 1, \ldots, n
\]

• The objective is naturally stated as

\[
\sum_{i=1}^{n} \sum_{j \in D(x_i)} c_{ij} y_{ij}
\]

LP relaxation (cont’d)

• The next task is to represent the actual constraint, and this depends on the combinatorial structure

• For example, if the constraint contains a permutation structure (such as the \textit{alldifferent}), we can add the constraints:

\[
\sum_{i=1}^{n} y_{ij} \leq 1 \text{ for all } j \in \bigcup_{i=1}^{n} D(x_i)
\]

• (Note that specific cuts known from MIP may be added to strengthen the LP)

• After the linear model is stated, we obtain the natural LP relaxation by removing the integrality condition on \( y_{ij} \):

\[
0 \leq y_{ij} \leq 1 \text{ for } i \in \{1, \ldots, n\}, j \in D(x_i)
\]
Reduced-cost based filtering

- The output of the LP solution is an optimal solution value $z^*$, a (fractional) value for each variable $y_{ij}$, and an associated reduced cost $\bar{c}_{ij}$.
- Recall that $\bar{c}_{ij}$ represents the marginal change in the objective value when variable $y_{ij}$ is forced in the solution.
- But $y_{ij}$ represents $x_i = j$.
- Reduced-cost based filtering:
  
  if $z^* + \bar{c}_{ij} > \max D(z_i)$ then $D(x_i) \leftarrow D(x_i) \setminus \{j\}$

  (This is a well-known technique in OR, called 'variable fixing')

Pros and Cons

- Potential drawbacks:
  - The filtering power depends directly on the quality of the LP relaxation, and it may be hard to find an effective relaxation.
  - Solving a LP using the simplex method may take much more time than propagating the constraint using combinatorial filtering algorithm.
- Potential benefits:
  - It's very generic; it works for any LP relaxation of a single constraint, a combination of constraints, or for the entire problem.
  - New insights in LP solving can have immediate impact.
  - For several constraint types, there exist fast and incremental combinatorial techniques to solve the LP relaxation.
  - This type optimality-based filtering complements nicely the feasibility-based filtering of CP; several applications cannot be solved with CP otherwise.

Example Application: TSP

- CP model
- LP relaxation
  - Assignment Problem
- Impact of reduced-cost based filtering

Graph $G = (V,E)$ with vertex set $V$ and edge set $E$

$|V| = n$

Let distance between $i$ and $j$ be represented by 'weight' function $w(i,j)$.
CP models for the TSP

- Permutation model
  - variable pos_i represents the i-th city to be visited
  - (can introduce dummy node pos_{n+1} = pos_1)
  \[
  \begin{align*}
  \min & \quad \sum_i w(pos_i, pos_{i+1}) \\
  \text{s.t.} & \quad \text{alldifferent}(pos_1, \ldots, pos_n) \\
  \end{align*}
  \]
  both models decouple the objective and the circuit

- Successor model
  - variable next_i represents the immediate successor of city i
  \[
  \begin{align*}
  \min & \quad \sum_i w(i, next_i) \\
  \text{s.t.} & \quad \text{alldifferent}(next_1, \ldots, next_n) \\
  & \quad \text{path}(next_1, \ldots, next_n) \quad \text{(Hamiltonian Path, not always supported by the CP solver)}
  \end{align*}
  \]

More CP models

- Combined model (still decoupled)
  \[
  \begin{align*}
  \min & \quad \sum_{i \in V} w(i, next_i) \\
  \text{s.t.} & \quad \text{alldifferent}(next_1, \ldots, next_n) \\
  & \quad \text{alldifferent}(pos_1, \ldots, pos_n) \\
  & \quad pos_j = next_{pos_j-1} \quad \forall j \in \{2, \ldots, n\} \\
  & \quad pos_1 = 1 
  \end{align*}
  \]

- Integrated model
  \[
  \begin{align*}
  \min & \quad z \\
  \text{s.t.} & \quad \text{alldifferent}(next_1, \ldots, next_n) \\
  & \quad \text{WeightedPath}(next, w, z) \quad \text{[Focacci et al., 1999, 2002]}
  \end{align*}
  \]
  (Note: most CP solvers do not support this constraint)

Relaxations for TSP

- An integrated model using \text{WeightedPath}(next, w, z) allows to apply an LP relaxation and perform reduced-cost based filtering

- Observe that the TSP is a combination of two constraints
  - The degree of each node is 2
  - The solution is connected (no sub tours)

- Relaxations:
  - relax connectedness: \text{Assignment Problem}
  - relax degree constraints: 1-Tree Relaxation
Benefits of AP relaxation

- Continuous relaxation provides integer solutions (total unimodularity)
- Specialized $O(n^3)$ algorithm (Hungarian method)
- Incremental $O(n^2)$ running time
- Reduced costs come for free
- Works well on asymmetric TSP

Assignment Problem (see introduction)

Binary variable $y_{ij}$ represents whether the tour goes from $i$ to $j$

\[
\begin{align*}
\min \; z &= \sum_{(i,j) \in E} w_{ij} y_{ij} \\
\text{s.t.} & \quad \sum_{j \in V} y_{ij} = 1, \forall j \in V \\
& \quad \sum_{i \in V} y_{ij} = 1, \forall i \in V \\
& \quad 0 \leq y_{ij} \leq 1, \forall i, j \in V
\end{align*}
\]

Computational results for TSP-TW

<table>
<thead>
<tr>
<th>Instance</th>
<th>Dyn.Prog.</th>
<th>Branch&amp;Cut</th>
<th>CP+LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>n</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>rbg201.2</td>
<td>21</td>
<td>9.60</td>
<td>0.22</td>
</tr>
<tr>
<td>rbg201.3</td>
<td>21</td>
<td>9.60</td>
<td>27.15</td>
</tr>
<tr>
<td>rbg201.4</td>
<td>21</td>
<td>11.52</td>
<td>5.82</td>
</tr>
<tr>
<td>rbg201.5</td>
<td>21</td>
<td>127.97</td>
<td>6.83</td>
</tr>
<tr>
<td>rbg201.6</td>
<td>21</td>
<td>161.66</td>
<td>1.38</td>
</tr>
<tr>
<td>rbg201.7</td>
<td>21</td>
<td>N.A.</td>
<td>4.30</td>
</tr>
<tr>
<td>rbg201.8</td>
<td>21</td>
<td>N.A.</td>
<td>17.49</td>
</tr>
<tr>
<td>rbg201.9</td>
<td>21</td>
<td>N.A.</td>
<td>26.12</td>
</tr>
<tr>
<td>rbg031a</td>
<td>36</td>
<td>18.63</td>
<td>0.98</td>
</tr>
<tr>
<td>rbg031a.3</td>
<td>37</td>
<td>7.67</td>
<td>1.83</td>
</tr>
<tr>
<td>rbg031a.4</td>
<td>40</td>
<td>8.64</td>
<td>423.23</td>
</tr>
<tr>
<td>rbg031a.5</td>
<td>42</td>
<td>20.98</td>
<td>751.42</td>
</tr>
<tr>
<td>rbg031a.6</td>
<td>43</td>
<td>24.57</td>
<td>N.A.</td>
</tr>
<tr>
<td>rbg031a.7</td>
<td>44</td>
<td>47.38</td>
<td>N.A.</td>
</tr>
<tr>
<td>rbg020a</td>
<td>52</td>
<td>N.A.</td>
<td>18.82</td>
</tr>
<tr>
<td>rbg067a</td>
<td>69</td>
<td>29.11</td>
<td>5.95</td>
</tr>
<tr>
<td>rbg123</td>
<td>152</td>
<td>37.90</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Langrangean Relaxation for LP

Move subset (or all) of constraints into the objective with 'penalty' multipliers $\mu$:

\[
\begin{align*}
\min \; c^T x & \quad \rightarrow \quad L(\mu) = \min \; c^T x + \mu^T (b_2 - A_2 x) \\
\text{s.t.} & \quad A_1 x = b_1 \\
& \quad A_2 x = b_2 \\
& \quad x \geq 0
\end{align*}
\]

Weak duality: for any choice of $\mu$, Lagrangean $L(\mu)$ provides a lower bound on the original LP

Goal: find optimal $\mu$ (providing the best bound) via

\[
\max_{\mu \geq 0} L(\mu)
\]
Motivation for using Lagrangeans

- Lagrangean relaxations can be applied to nonlinear programming problems (NLPs), LPs, and in the context of integer programming
- Lagrangean relaxation can provide better bounds than LP relaxation
- The Lagrangean dual generalizes LP duality
- It provides domain filtering analogous to that based on LP duality
- Lagrangean relaxation can dualize ‘difficult’ constraints
  - Can exploit the problem structure, e.g., the Lagrangean relaxation may decouple, or $L(\mu)$ may be very fast to solve combinatorially
- Next application: Lagrangean relaxation for TSP

Recall: Relaxations for TSP

- An integrated model using WeightedPath(next, w, z) allows to apply an LP relaxation and perform reduced-cost based filtering
- Observe that the TSP is a combination of two constraints
  - The degree of each node is 2
  - The solution is connected (no sub tours)
- Relaxations:
  - relax connectedness: Assignment Problem
  - relax degree constraints: 1-Tree Relaxation

The 1-Tree Relaxation for TSP

- Held and Karp [1970, 1971] proposed a lower bound based on a relaxation of the degree constraints
- A minimum spanning tree gives such a relaxation
- A 1-tree is a stronger relaxation, which can be obtained by:
  - Choosing any node v (which is called the 1-node)
  - Building a minimum spanning tree T on $G = (\mathcal{V}(v), E)$
  - Adding the smallest two edges linking v to T
- For n vertices, a 1-tree contains n edges

P.S. an MST can be found in $O(m \alpha(m,n))$ time
The Held and Karp bound for TSP

The 1-tree can be tightened through the use of Lagrangean relaxation by relaxing the degree constraints in the TSP model:

Let binary variable $x_e$ represent whether edge $e$ is used

\[
\min \sum_{e \in E} w(e)x_e \\
\text{s.t.} \sum_{e \in \mathcal{E}(i)} x_e = 2 \quad \forall i \in V \\
\sum_{i,j \in S, i < j} x_{(i,j)} \leq |S| - 1 \quad \forall S \subset V, |S| \geq 3 \\
x_e \in \{0, 1\} \quad \forall e \in E
\]

Lagrangean relaxation with multipliers $\pi$ (penalties for node degree violation):

\[
\min \sum_{e \in E} w(e)x_e + \sum_{i \in V \setminus \{1\}} \pi_i \left(2 - \sum_{e \in \mathcal{E}(i)} x_e \right) \\
\text{s.t.} \quad \sum_{i,j \in S, i < j} x_{(i,j)} \leq |S| - 1 \quad \forall S \subset V \setminus \{1\}, |S| \geq 3 \\
\sum_{e \in \mathcal{E}(i)} x_e = 2 \quad \forall i \in V \setminus \{1\} \\
\sum_{e \in E} x_e = |V| \\
x_e \in \{0, 1\}
\]

How to find the best penalties $\pi$?
- In general, subgradient optimization
- But here we can exploit a combinatorial interpretation
- No need to solve LP

Held-Karp iteration

- Solve 1-tree w.r.t. updated edge weights $w'(e) = w(e) - \pi_i - \pi_j$
- Optimal 1-tree $T$ gives lower bound: $\text{cost}(T) + 2 \sum \pi_i$
- If $T$ is not a tour, then we iteratively update the penalties as $\pi_i = (2 \cdot \deg(i)) \beta$ (step size $\beta$ different per iteration) and repeat

$w(2,4) = w(2,4) - \pi_2 - \pi_1$
How can we exploit 1-tree in CP?

- We need to reason on the graph structure
  - manipulate the graph, remove costly edges, etc.
- Not easily done with ‘next’ and ‘pos’ variables
  - e.g., how can we enforce that a given edge e=(i,j) is mandatory?
    - (next = j or next = i) ?
    - \( pos_i = i \Rightarrow (pos_{i+1} = j) \) or \( pos_{i+1} = j) \) ?
- Ideally, we want to have access to the graph rather than local successor/predecessor information
  - modify definition of global constraint

One more CP model for the TSP

Integrated model based on graph representation

\[
\begin{align*}
\min & \quad z \\
\text{s.t.} & \quad \text{weighted-circuit}(X, G, z)
\end{align*}
\]

- \( G=\{V,E,w\} \) is the graph with vertex set \( V \), edge set \( E \), weights \( w \)
- \( X \) is a set variable representing the set of edges that will form the circuit
  - Domain \( D(X) = \{ (X) \cup \{U\} \} \), with fixed cardinality \( |V| \) in this case
  - Lower bound \( LX \) is set of mandatory edges
  - Upper bound \( UX \) is set of possible edges
- \( z \) is a variable representing the total edge weight
Domain Filtering

- Given constraint
  \[ \text{weighted-circuit}(X, G=(V,E,w), z) \]
- Apply the 1-tree relaxation to
  - remove sub-optimal edges from \( U(X) \)
  - force mandatory edges into \( L(X) \)
  - update bounds of \( z \)
- For simplicity, the presentation of the algorithms are restricted to \( G = (V \setminus \{1\}, E) \)

Removing non-tree edges

- The marginal cost of a non-tree edge \( e \) is the additional cost of forcing \( e \) in the solution:
  \[ c'_e = \text{cost}(T(e)) - \text{cost}(T) \]
- Given a current best solution UB, edge \( e \) can be removed if
  \[ \text{cost}(T(e)) > UB, \text{ or } c'_e + \text{cost}(T) > UB \]

Replacement cost of
- \((1,2)\) is \(4 - 2 = 2\)
- \((6,7)\) is \(5 - 5 = 0\)

Computing marginal costs

Basic algorithm for computing marginal edge costs:
- For each non-tree edge \( e=(i,j) \)
  - find the unique i-j path \( P_e \) in the tree
  - the marginal cost of \( e \) is \( c_e = \max(c_a \mid a \in P_e) \)

Complexity: \(O(mn)\), since \( P_e \) can be found in \(O(n)\) time by DFS

Can be further improved to \(O(m + n + n \log n)\) [Regin, 2008]
Impact of edge filtering

Forcing tree edges

- The replacement cost of a tree edge $e$ is the additional cost when $e$ is removed from the tree:
  \[ c'_e = \text{cost}(T \setminus e) - \text{cost}(T) \]
- Given a current best solution $UB$, edge $e$ is mandatory if $\text{cost}(T \setminus e) > UB$, or $c'_e + \text{cost}(T) > UB$

replacement cost of (1,4)?
we need to find the cheapest edge to reconnect: 3 - 1 = 2

Computing replacement costs

1. Compute minimum spanning tree $T$ in $G$
2. Mark all edges in $T$ as ‘unmarked’
3. Consider non-tree edges, ordered by non-decreasing weight:
   - For non-tree edge $(i,j)$, traverse the i-j path in $T$
   - Mark all unmarked edges $e$ on this path, and assign $c'_e = c_{ij} - c_e$
4. Basic time complexity $O(mn)$, or, at no extra cost if performed together with the computation of marginal costs

<table>
<thead>
<tr>
<th>non-tree edge</th>
<th>mark edge</th>
<th>replacement cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,4)</td>
<td>(1,2)</td>
<td>3 - 2 = 1</td>
</tr>
<tr>
<td>(1,4)</td>
<td>(2,4)</td>
<td>4 - 2 = 2</td>
</tr>
<tr>
<td>(1,3)</td>
<td></td>
<td>3 - 2 = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(edge (1,4) already marked)</td>
</tr>
</tbody>
</table>
Improving the time complexity

- We can improve this complexity by 'contracting' the marked edges (that is, we merge the extremities of the edge)
  - First, root the minimum spanning tree
  - Apply Tarjan's 'path compression' technique during the algorithm
  - This leads to a time complexity of $O(m \alpha(m,n))$
Outline

• Global constraint propagation
  – matching theory for *alldifferent*
  – network flow theory for *cardinality* constraint

• Integrating relaxations
  – Linear Programming relaxation
  – Lagrangean relaxation

• *Decomposition methods*
  – logic-based Benders
  – column generation

Motivation

• Many practical applications are composed of several subproblems
  – *facility location*: assign orders to facilities with minimum cost, but respect facility constraints
  – *vehicle routing*: assign pick-up locations to trucks, while respecting constraints on truck (capacity, driver time, …)

• By solving subproblems separately we can
  – be more scalable (decrease solving time)
  – exploit the subproblem structure

• OR-based decomposition methods can preserve optimality

Motivation for integrated approach

Example: *airline crew rostering*

• Crew members are assigned a schedule from a huge list of possible schedules
  – this is a ‘set covering’ problem: relatively easy for IP/LP

• New schedules are added to the list as needed
  – many challenging scheduling constraints – difficult for MIP, but doable for CP

• *Integrated OR/CP decompositions* broaden the applicability to more complex and larger applications
Benders Decomposition

Benders decomposition can be applied to problems of the form:

\[ \min \ v = f(x, y) \quad \text{s.t.} \quad S(x, y) \in D_x, y \in D_y \]

When fixing variables \( x \), the resulting problem may become much simpler:

\[ \min \ f(x) \quad \text{s.t.} \quad S(x) \in D_x \]

Example: multi-machine scheduling
- variables \( x \) assign tasks to machines
- variables \( y \) give feasible/optimal schedules per machine
- when fixing \( x \), the problem decouples into independent single-machine scheduling problems on \( y \)

Benders Decomposition (cont’d)

Iterative process
- Master problem: search over variables \( x \)
  - optimal solution \( x^k \) in iteration \( k \)
- Subproblems: search over variables \( y \), given fixed \( x^k \)
  - optimal objective value \( v^k \)
- Add Benders cut to master problem
  \[ v \geq B_k(x) \quad \text{such that } B_k(x^k) = v^k \]

Bounding
- Master is relaxation: gives lower bound
- Subproblem is restriction: gives upper bound
- Process repeats until the bounds meet

Logic-based Benders

- Original Benders decomposition applies to LP and NLP problems
  - Based on duality theory to obtain Benders cuts
- However, the concept is more general
  - Logic-based Benders: generalizes LP-based Benders to other types of inference methods, using 'inference duality'
  - In particular, CP can be applied to solve the subproblems
  - Also allows additional types of 'feasibility' cuts (nogoods)

[Jain & Grossmann, 2001] [Hooker & Ottoson, 2003]
Example: Task-Facility Allocation

Benders Scheme

Pros and Cons

- Benefits of Logic-based Benders
  - reported orders of magnitude improvements in solving time
    [Jain & Grossmann, 2001], [Hooker, 2007]
  - CP models very suitable for more complex subproblems such as scheduling, rostering, etc.

- Potential drawbacks
  - finding good Benders cuts for specific application may be challenging
  - feasible solution may be found only at the very end of the iterative process
**Column Generation**

- One of the most important techniques for solving very large scale linear programming problems
  - perhaps too many variables to load in memory

\[
\begin{align*}
\min & \quad c^T x \\
\text{s.t.} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

- Delayed column generation (or variable generation):
  - start with subset of variables ("restricted master problem")
  - iteratively add variables to model until optimality condition is met

**Column Generation (cont’d)**

Delayed column generation:
- Solve for subset of variables \( S \) (assume feasible)
- This gives shadow prices \( \lambda \) for the constraints
- Use reduced costs to price the variables not in \( S \)
  \[
  \tilde{c}_i = c_i - \sum j \lambda_j a_{ij}
  \]
- If \( \tilde{c}_i < 0 \), variable \( x_i \) may improve the solution:
  - add \( x_i \) to \( S \) and repeat
- Otherwise, we are optimal (since all reduced costs are nonnegative)

How can we find the best variable to add?

**Pricing Problem**

- Solve optimization problem to find the variable (column) with the minimum reduced cost:
  \[
  \begin{align*}
  \min & \quad c^T x \\
  \text{s.t.} & \quad Ax \geq b \\
  & \quad x \geq 0
  \end{align*}
  \]
  \[
  \begin{align*}
  \min & \quad c^T x \\
  \text{s.t.} & \quad Ax \geq b \\
  & \quad x \geq 0
  \end{align*}
  \]

- In many cases, columns of \( A \) can be described using a set of (complicated) constraints
- Remarks:
  - any negative reduced cost column suffices (need not be optimal)
  - CP can be suitable method for solving pricing problem
Application: Capacitated Vehicle Routing

- Set of clients $V$, depot $d$
- Set of trucks (unlimited, equal)
- Parameters:
  - distance matrix $D$
  - load $w_j$ for each client $j$ in $V$ (unsplitable)
  - truck capacity $Q$
- Goal:
  - find an allocation of clients to trucks
  - and a route for each truck
  - respecting all constraints
  - with minimum total distance

Problem Formulation: Restricted Master

- Let $R$ be (small) set of feasible individual truck routes
  - parameter $a_{rj} = 1$ if client $j$ is on route $r \in R$
  - parameter $c_r$ represent the length of route $r \in R$
- Let binary variable $x_r$ represent whether we use route $r \in R$
- Set covering formulation:

\[
\begin{align*}
\min & \quad \sum_{r \in R} c_r x_r \\
\text{s.t.} & \quad \sum_{r \in R} a_{rj} x_r \geq 1 \quad \forall j \in V \\
& \quad 0 \leq x_r \leq 1 \quad \forall r \in R
\end{align*}
\]

- shadow price $\lambda_j$ for all $j$
- continuous LP relaxation

Pricing Problem

- Truck route similar to TSP, but
  - not all locations need to be visited
  - there is a capacity constraint on the trucks
- We can solve this problem in different ways
  - shortest path problem in a layered graph
  - single machine scheduling problem
**Pricing as shortest path**

Binary variable $y_{ijk}$: travel from location $i$ to $j$ in step $k$

Constraints:
- variables $y_{ijk}$ must represent a path from and to the depot
- we can visit each location at most once
- total load cannot exceed capacity $Q$

This model can be solved by IP (or dedicated algorithms)

---

**Benefit of using CP**

- We can use CP to solve the pricing problem:
  - represent the constrained shortest path as CP model,
  - or we can view the pricing problem as a single machine scheduling problem
- A major advantage is that CP allows to add many more side constraints:
  - time window constraints for the clients
  - precedence relations due to stacking requirements
  - union regulations for the drivers
  - ...
- In such cases, other methods such as IP may no longer be applicable

---

**From TSP to machine scheduling**

- Vehicle corresponds to 'machine' or 'resource'
- Visiting a location corresponds to 'activity'

- Sequence-dependent setup times
  - Executing activity $j$ after activity $i$ induces setup time $D_{ij}$ (distance)
  - Minimize 'makespan' (or sum of the setup times)
  - Activities cannot overlap (disjunctive resource)
**CP Model**

- **Activities**:
  - Optional activity \( \text{visit}[j] \) for each client \( j \) (duration: 0)
  - \( \text{StartAtDepot} \)
  - \( \text{EndAtDepot} \)

- **Transition times** between two activities \( i \) and \( j \)
  - \( T_{i,j} = D(i,j) - \lambda_i \)

**CP Model (cont'd)**

\[
\begin{align*}
\text{minimize} & \quad \text{EndAtDepot}.\text{end} - \sum_j \lambda_j(\text{Visit}[j].\text{present}) \\
\text{s.t.} & \quad \text{DisjunctiveResource} \\
& \quad \text{Activities:} \quad \text{Visit}[j], \text{StartAtDepot}, \text{EndAtDepot} \\
& \quad \text{Transition:} \quad T_{i,j} \\
& \quad \text{First:} \quad \text{StartAtDepot} \\
& \quad \text{Last:} \quad \text{EndAtDepot} \\
& \quad \sum_j w_j(\text{Visit}[j].\text{present}) \leq Q
\end{align*}
\]

- Observe that this model naturally allows to add time windows (on Visit[j]) precedence relations, etc

**Discussion**

- **Benefits of column generation**
  - A small number of variables may suffice to prove optimality of a problem with exponentially many variables
  - Complicated constraints can be moved to subproblem
  - Can stop at any time and have feasible solution (not the case with Benders)

- **Potential drawbacks / challenges**
  - LP-based column generation still fractional: need branch-and-price method to be exact (can be challenging)
  - For degenerate LPs, shadow prices may be non-informative
  - Difficult to replace single columns: need sets of new columns which are hard to find simultaneously