DIP with CHiPPS: Decomposition Methods for Integer Linear Programming

Ted Ralphs
Lehigh University
Matthew Galati
SAS Institute
Jiadong Wang
Lehigh University

University of Michigan, 2 November 2011

Thanks: Work supported in part by the National Science Foundation

Outline

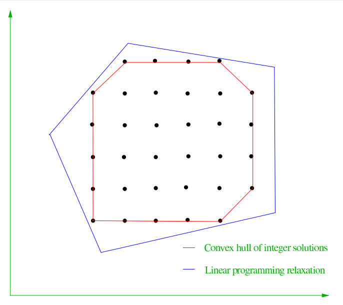
Outline

1. Motivation
2. Methods
   - Cutting Plane Method
   - Dantzig-Wolfe Method
   - Lagrangian Method
   - Integrated Methods
3. Software
   - Implementation and API
   - Algorithmic Details
4. Interfaces
   - DIPPY
   - MILPBlock
5. Current and Future Research

The Basic Setting

Integer Linear Program: Minimize/Maximize a linear *objective function* over a (discrete) set of *solutions* satisfying specified *linear constraints*.

$$
\begin{aligned}
z_{\mathrm{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{\mathrm{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}
\end{aligned}
$$



Branch and Bound

- A *relaxation* of an ILP is an auxiliary mathematical program for which
  - the feasible region contains the feasible region for the original ILP, and
  - the objective function value of each solution to the original ILP is not increased.
  - Relaxations can be used to efficiently get bounds on the value of the original integer program.
- Types of Relaxations
  - Continuous relaxation
  - Combinatorial relaxations
  - Lagrangian relaxations

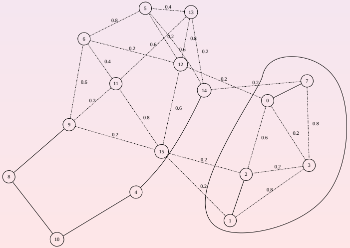
Branch and Bound

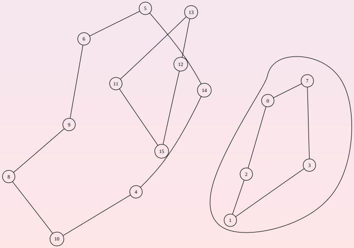
Initialize the queue with the root subproblem. While there are subproblems in the queue, do

1. Remove a subproblem and solve its relaxation.
2. The relaxation is infeasible $\Rightarrow$ subproblem is infeasible and can be pruned.
3. Solution is feasible for the MILP $\Rightarrow$ subproblem solved (update upper bound).
4. Solution is not feasible for the MILP $\Rightarrow$ lower bound.
   - If the lower bound exceeds the global upper bound, we can *prune the node*.
   - Otherwise, we *branch* and add the resulting subproblems to the queue.

What is the Goal of Decomposition?

- Basic Idea: Exploit knowledge of the underlying structural components of model to improve the bound.
- Many complex models are built up from multiple underlying substructures.
  - Subsystems linked by global constraints.
  - Complex combinatorial structures obtained by combining simple ones.
- We want to exploit knowledge of efficient, customized methodology for substructures.
- This can be done in two primary ways (with many variants).
  - Identify independent subsystems.
  - Identify subsets of constraints that can be dealt with efficiently.

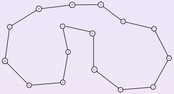




Example: Exposing Combinatorial Structure

Traveling Salesman Problem Formulation

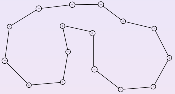
$$
\begin{array}{llll}
x(\delta(\{u\})) & = & 2 & \forall u \in V \\
x(E(S)) & \leq & |S| - 1 & \forall S \subset V,\ 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0, 1\} & & & \forall e \in E
\end{array}
$$



Example: Exposing Combinatorial Structure

Traveling Salesman Problem Formulation

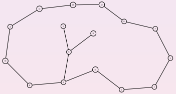
$$
\begin{array}{lll}
x(\delta(\{u\})) & = & 2 \quad \forall u \in V \\
x(E(S)) & \leq & |S| - 1 \quad \forall S \subset V,\ 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0,1\} & & \forall e \in E
\end{array}
$$



Two relaxations

Find a spanning subgraph with $|V|$ edges ($\mathcal{P}' =$ 1-Tree)

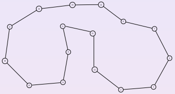
$$
\begin{array}{lll}
x(\delta(\{0\})) & = & 2 \\
x(E(V)) & = & |V| \\
x(E(S)) & \leq & |S| - 1 \quad \forall S \subset V \setminus \{0\}, 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0,1\} & & \forall e \in E
\end{array}
$$



Example: Exposing Combinatorial Structure

Traveling Salesman Problem Formulation

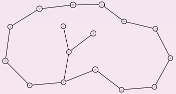
$$\begin{array}{lll} x(\delta(\{u\})) & = & 2 \quad \forall u \in V \\ x(E(S)) & \leq & |S|-1 \quad \forall S \subset V,\ 3 \leq |S| \leq |V|-1 \\ x_e \in \{0,1\} & & \quad \forall e \in E \end{array}$$



Two relaxations

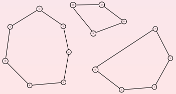
Find a spanning subgraph with $|V|$ edges ($\mathcal{P}' =$ 1-Tree)

$$\begin{array}{lll} x(\delta(\{0\})) & = & 2 \\ x(E(V)) & = & |V| \\ x(E(S)) & \leq & |S|-1 \quad \forall S \subset V \setminus \{0\}, 3 \leq |S| \leq |V|-1 \\ x_e \in \{0,1\} & & \quad \forall e \in E \end{array}$$



Find a 2-matching that satisfies the subtour constraints ($\mathcal{P}' =$ 2-Matching)

$$\begin{array}{lll} x(\delta(\{u\})) & = & 2 \quad \forall u \in V \\ x_e \in \{0,1\} & & \quad \forall e \in E \end{array}$$



Example: Exposing Block Structure

- One motivation for decomposition is to expose *independent subsystems*.
- The key is to identify *block structure* in the constraint matrix.
- The separability lends itself nicely to parallel implementation.

$$\begin{pmatrix} A''_1 & A''_2 & \cdots & A''_\kappa \\ \hline A'_1 & & & \\ & A'_2 & & \\ & & \ddots & \\ & & & A'_\kappa \end{pmatrix}$$

Example: Exposing Block Structure

- One motivation for decomposition is to expose *independent subsystems*.
- The key is to identify *block structure* in the constraint matrix.
- The separability lends itself nicely to parallel implementation.

$$\begin{pmatrix} A''_1 & A''_2 & \cdots & A''_\kappa \\ \hline A'_1 & & & \\ & A'_2 & & \\ & & \ddots & \\ & & & A'_\kappa \end{pmatrix}$$

Example: Exposing Block Structure

- One motivation for decomposition is to expose *independent subsystems*.
- The key is to identify *block structure* in the constraint matrix.
- The separability lends itself nicely to parallel implementation.

Generalized Assignment Problem (GAP)

- The problem is to assign m tasks to n machines subject to capacity constraints.
- An IP formulation of this problem is

$$
\begin{array}{llll}
\min & \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} & & \\
& \sum_{j \in N} w_{ij} x_{ij} \leq & b_i & \forall i \in M \\
& \sum_{i \in M} x_{ij} = & 1 & \forall j \in N \\
& x_{ij} \in & \{0,1\} & \forall i,j \in M \times N
\end{array}
$$

- The variable x_{ij} is one if task i is assigned to machine j.
- The "profit" associated with assigning task i to machine j is c_{ij}.

Example: Eliminating Symmetry

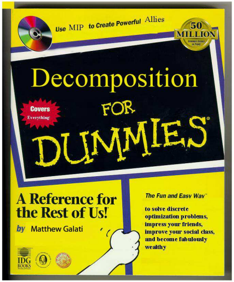
- In some cases, the identified blocks are *identical*.
- In such cases, the original formulation will often be highly symmetric.
- The decomposition eliminates the symmetry by collapsing the identical blocks.

Vehicle Routing Problem (VRP)

$$
\begin{array}{llll}
\min & \sum_{k \in M} \sum_{(i,j) \in A} c_{ij} x_{ijk} & & \\
& \sum_{k \in M} \sum_{j \in N} x_{ijk} & = 1 & \forall i \in V \\
& \sum_{i \in V} \sum_{j \in N} d_i x_{ijk} & \leq C & \forall k \in M \\
& \sum_{j \in N} x_{0jk} & = 1 & \forall k \in M \\
& \sum_{i \in N} x_{ihk} - \sum_{j \in N} x_{hjk} & = 0 & \forall h \in V, k \in M \\
& \sum_{i \in N} x_{i,n+1,k} & = 1 & \forall k \in M \\
& x_{ijk} \in \{0,1\} & & \forall (i,j) \in A, k \in M
\end{array}
$$

DIP and CHiPPS

- The use of decomposition methods in practice is hindered by a number of serious drawbacks.
  - *Implementation is difficult*, usually requiring development of sophisticated customized codes.
  - Choosing an algorithmic strategy requires *in-depth knowledge* of theory and strategies are *difficult to compare empirically*.
  - The powerful techniques modern solvers use to solve integer programs are *difficult to integrate* with decomposition-based approaches.
- DIP and CHiPPS are two frameworks that together allow for easier implementation of decomposition approaches.
  - CHiPPS (COIN High Performance Parallel Search Software) is a flexible library hierarchy for implementing parallel search algorithms.
  - DIP (Decomposition for Integer Programs) is a framework for implementing decomposition-based bounding methods.
  - DIP with CHiPPS is a full-blown branch-and-cut-and-price framework in which details of the implementation are hidden from the user.
- DIP can be accessed through a modeling language or by providing a model with notated structure.



Outline

The Decomposition Principle in Integer Programming

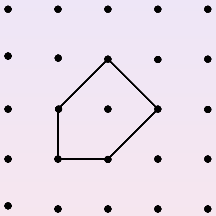
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

$$z_{\mathrm{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



$\mathcal{P} = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b',\ A''x \geq b''\}$

Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

The Decomposition Principle in Integer Programming

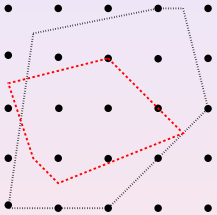
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

$$z_{\mathrm{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



$\mathcal{Q}' = \{x \in \mathbb{R}^n \mid A'x \geq b'\}$

$\mathcal{Q}'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}$

Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

The Decomposition Principle in Integer Programming

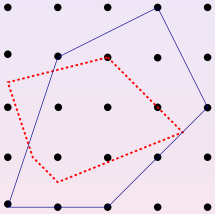
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

$$z_{\mathrm{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



$\mathcal{P}' = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b'\}$

$\mathcal{Q}'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}$

Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

The Decomposition Principle in Integer Programming

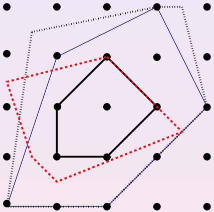
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

$$z_{\mathrm{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



$\mathcal{P} = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b', A''x \geq b''\}$

$\mathcal{P}' = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b'\}$

$\mathcal{Q}' = \{x \in \mathbb{R}^n \mid A'x \geq b'\}$

$\mathcal{Q}'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}$

Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

The Decomposition Principle in Integer Programming

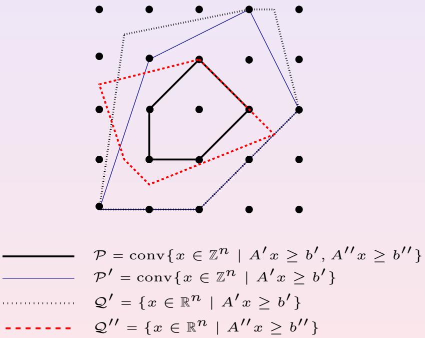
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

$$z_{\mathrm{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

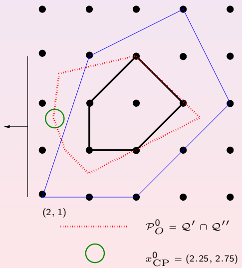
Cutting Plane Method (CPM)

CPM combines an *outer* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{CP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Dx \geq d, A''x \geq b'' \right\}$
- **Subproblem**: $\mathrm{SEP}(\mathcal{P}', x_{\mathrm{CP}})$

$$\mathcal{P}' = \{x \in \mathbb{R}^n \mid Dx \geq d\}$$

Exponential number of constraints



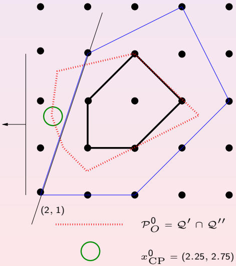
Cutting Plane Method (CPM)

CPM combines an *outer* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{CP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Dx \geq d, A''x \geq b'' \right\}$
- **Subproblem**: $\mathrm{SEP}(\mathcal{P}', x_{\mathrm{CP}})$

$$\mathcal{P}' = \{x \in \mathbb{R}^n \mid Dx \geq d\}$$

Exponential number of constraints



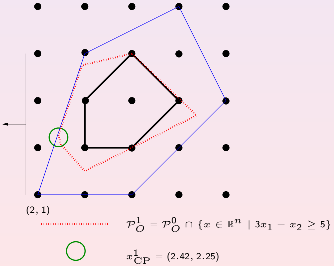
Cutting Plane Method (CPM)

CPM combines an *outer* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{CP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Dx \geq d, A''x \geq b'' \right\}$
- **Subproblem**: $\mathrm{SEP}(\mathcal{P}', x_{\mathrm{CP}})$

$$\mathcal{P}' = \{x \in \mathbb{R}^n \mid Dx \geq d\}$$

Exponential number of constraints



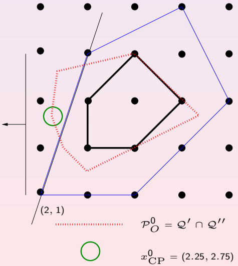
Cutting Plane Method (CPM)

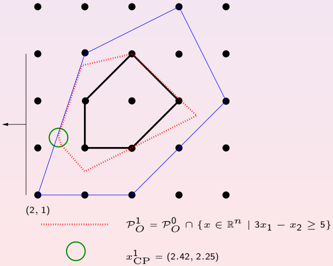
CPM combines an *outer* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{CP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid Dx \geq d, A''x \geq b''\}$
- **Subproblem**: $\mathrm{SEP}(\mathcal{P}', x_{\mathrm{CP}})$

$$\mathcal{P}' = \{x \in \mathbb{R}^n \mid Dx \geq d\}$$

Exponential number of constraints





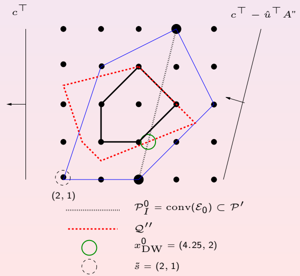
Dantzig-Wolfe Method (DW)

DW combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{DW}} = \min_{\lambda \in \mathbb{R}^{\mathcal{E}}_{+}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \mid A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{DW}}^\top A''\right)$

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{E}} s\lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E} \right\}$$

Exponential number of variables



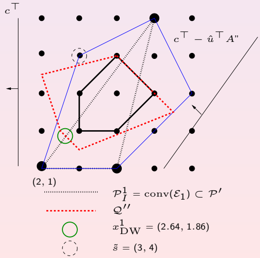
Dantzig-Wolfe Method (DW)

DW combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{DW}} = \min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{DW}}^\top A''\right)$

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{E}} s\lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E} \right\}$$

Exponential number of variables



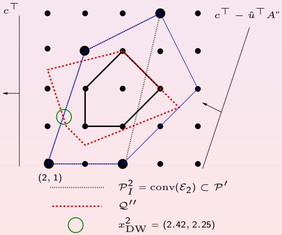
Dantzig-Wolfe Method (DW)

DW combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{DW}} = \min_{\lambda \in \mathbb{R}^{\mathcal{E}}_{+}} \left\{ c^\top \left(\sum_{s\in\mathcal{E}} s\lambda_s \right) \mid A'' \left(\sum_{s\in\mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s\in\mathcal{E}} \lambda_s = 1 \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u^\top_{\mathrm{DW}} A''\right)$

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s\in\mathcal{E}} s\lambda_s, \sum_{s\in\mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E} \right\}$$

Exponential number of variables



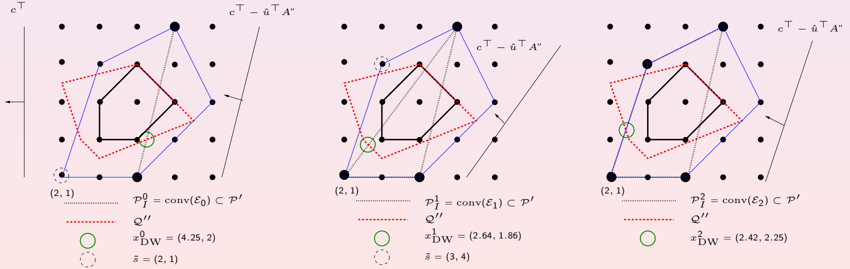
Dantzig-Wolfe Method (DW)

DW combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master**: $z_{\mathrm{DW}} = \min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \mid A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{DW}}^\top A''\right)$

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{E}} s\lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \;\forall s \in \mathcal{E} \right\}$$

Exponential number of variables

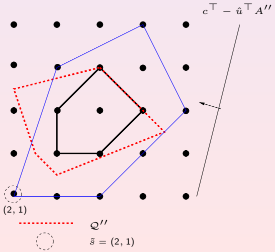


Lagrangian Method (LD)

LD iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

- **Master**: $z_{\mathrm{LD}} = \max_{u \in \mathbb{R}^{m''}_{+}} \left\{ \min_{s \in \mathcal{E}} \left\{ c^\top s + u^\top (b'' - A'' s) \right\} \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{LD}}^\top A''\right)$

$$z_{\mathrm{LD}} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \alpha + b''^\top u \;\middle|\; \left(c^\top - u^\top A''\right) s - \alpha \geq 0 \;\forall s \in \mathcal{E} \right\} = z_{\mathrm{DW}}$$

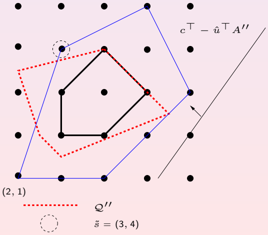


Lagrangian Method (LD)

LD iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

- **Master**: $z_{\mathrm{LD}} = \max_{u \in \mathbb{R}^{m''}_{+}} \left\{ \min_{s \in \mathcal{E}} \left\{ c^\top s + u^\top (b'' - A'' s) \right\} \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{LD}}^\top A''\right)$

$$z_{\mathrm{LD}} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \alpha + b''^\top u \;\middle|\; \left(c^\top - u^\top A''\right) s - \alpha \geq 0 \ \forall s \in \mathcal{E} \right\} = z_{\mathrm{DW}}$$

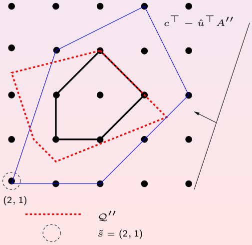


Lagrangian Method (LD)

LD iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

- **Master**: $z_{\text{LD}} = \max_{u \in \mathbb{R}_+^{m''}} \left\{ \min_{s \in \mathcal{E}} \left\{ c^\top s + u^\top (b'' - A'' s) \right\} \right\}$
- **Subproblem**: $\text{OPT}\left(\mathcal{P}', c^\top - u_{\text{LD}}^\top A''\right)$

$$z_{\text{LD}} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}_+^{m''}} \left\{ \alpha + b''^\top u \;\middle|\; \left(c^\top - u^\top A''\right) s - \alpha \geq 0 \; \forall s \in \mathcal{E} \right\} = z_{\text{DW}}$$

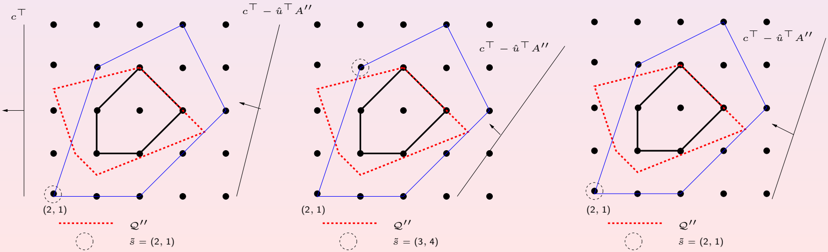


Lagrangian Method (LD)

LD iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

- **Master**: $z_{\mathrm{LD}} = \max_{u \in \mathbb{R}^{m''}_{+}} \left\{ \min_{s \in \mathcal{E}} \left\{ c^\top s + u^\top (b'' - A'' s) \right\} \right\}$
- **Subproblem**: $\mathrm{OPT}\left(\mathcal{P}', c^\top - u_{\mathrm{LD}}^\top A''\right)$

$$z_{\mathrm{LD}} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \alpha + b''^\top u \;\middle|\; \left(c^\top - u^\top A''\right) s - \alpha \geq 0 \; \forall s \in \mathcal{E} \right\} = z_{\mathrm{DW}}$$



Common Threads

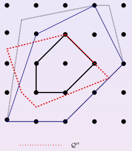
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

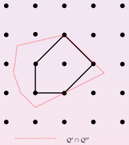
$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

$$z_{\mathrm{CP}} = z_{\mathrm{DW}} = z_{\mathrm{LD}} = z_{\mathrm{D}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{\mathrm{LP}}$$

- Traditional decomp-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual solution information
  - **Subproblem:** Update the approximation of $\mathcal{P}'$: $\mathrm{SEP}(\mathcal{P}', x)$ or $\mathrm{OPT}(\mathcal{P}', c)$
- Integrated decomposition methods further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.
  - Price-and-Cut (PC)
  - Relax-and-Cut (RC)
  - Decompose-and-Cut (DC)





Common Threads

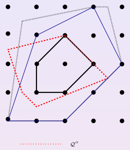
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

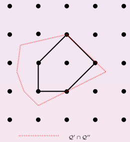
$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

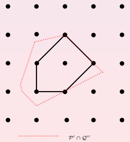
- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

$$z_{\mathrm{CP}} = z_{\mathrm{DW}} = z_{\mathrm{LD}} = z_{\mathrm{D}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{\mathrm{LP}}$$

- Traditional decomp-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual solution information
  - **Subproblem:** Update the approximation of $\mathcal{P}'$: $\mathrm{SEP}(\mathcal{P}', x)$ or $\mathrm{OPT}(\mathcal{P}', c)$
- Integrated decomposition methods further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.
  - Price-and-Cut (PC)
  - Relax-and-Cut (RC)
  - Decompose-and-Cut (DC)







Common Threads

- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

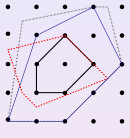
$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

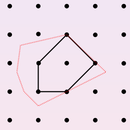
- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

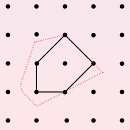
$$z_{\mathrm{CP}} = z_{\mathrm{DW}} = z_{\mathrm{LD}} = z_{\mathrm{D}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{\mathrm{LP}}$$

- Traditional decomp-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual solution information
  - **Subproblem:** Update the approximation of $\mathcal{P}'$: $\mathrm{SEP}(\mathcal{P}', x)$ or $\mathrm{OPT}(\mathcal{P}', c)$

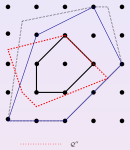
- Integrated decomposition methods further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.
  - Price-and-Cut (PC)
  - Relax-and-Cut (RC)
  - Decompose-and-Cut (DC)





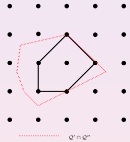


Common Threads



- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

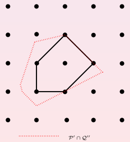
$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$



- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

$$z_{\mathrm{CP}} = z_{\mathrm{DW}} = z_{\mathrm{LD}} = z_{\mathrm{D}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{\mathrm{LP}}$$

- Traditional decomp-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual **solution** information
  - **Subproblem:** Update the **approximation** of $\mathcal{P}'$: $\mathrm{SEP}(\mathcal{P}', x)$ or $\mathrm{OPT}(\mathcal{P}', c)$



- Integrated decomposition methods further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.
  - **Price-and-Cut** (PC)
  - **Relax-and-Cut** (RC)
  - **Decompose-and-Cut** (DC)

Decompose-and-Cut (DC)

Decompose-and-Cut: Each iteration of CPM, decompose into convex combo of e.p.'s of $\mathcal{P}'$

$$\min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}, (x^+, x^-) \in \mathbb{R}_+^n} \left\{ x^+ + x^- \;\middle|\; \sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}_{\mathrm{CP}}, \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$$

Decompose-and-Cut (DC)

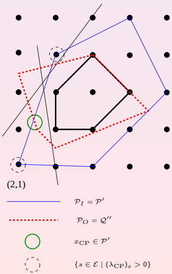
Decompose-and-Cut: Each iteration of CPM, decompose into convex combo of e.p.'s of $\mathcal{P}'$

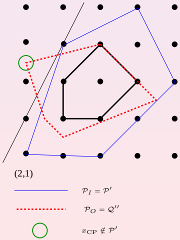
$$\min_{\lambda \in \mathbb{R}^{\mathcal{E}}_{+}, (x^+, x^-) \in \mathbb{R}^n_+} \left\{ x^+ + x^- \;\middle|\; \sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}_{\mathrm{CP}}, \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$$

- If $\hat{x}_{\mathrm{CP}}$ lies outside $\mathcal{P}'$ the decomposition will fail
- By the *Farkas Lemma* the proof of infeasibility provides a valid and violated inequality

Decomposition Cuts

$$\begin{aligned} u^t_{\mathrm{DC}} s + \alpha^t_{\mathrm{DC}} &\leq 0 \;\forall s \in \mathcal{P}' \quad \text{and} \\ u^t_{\mathrm{DC}} \hat{x}_{\mathrm{CP}} + \alpha^t_{\mathrm{DC}} &> 0 \end{aligned}$$





Decompose-and-Cut (DC)

Decompose-and-Cut: Each iteration of CPM, decompose into convex combo of e.p.'s of $\mathcal{P}'$.

$$\min_{\lambda \in \mathbb{R}^{\mathcal{E}}_{+}, (x^+, x^-) \in \mathbb{R}^n_+} \left\{ x^+ + x^- \;\middle|\; \sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}_{\mathrm{CP}}, \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$$

- Original used to solve VRP with TSP as relaxation.
- Essentially, we are transforming an optimization algorithm into a separation algorithm.
- The machinery for solving this already exists (=column generation)
- Much easier than DW problem because it's a *feasibility* problem and
  - $\hat{x}_i = 0 \Rightarrow s_i = 0$, can remove constraints not in support, and
  - $\hat{x}_i = 1 \text{ and } s_i \in \{0, 1\} \Rightarrow$ constraint is redundant with convexity constraint
  - Often gets *lucky* and produces incumbent solutions to original IP

Outline

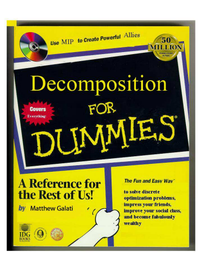
1. Motivation
2. Methods
   - Cutting Plane Method
   - Dantzig-Wolfe Method
   - Lagrangian Method
   - Integrated Methods
3. Software
   - Implementation and API
   - Algorithmic Details
4. Interfaces
   - DIPPY
   - MILPBlock
5. Current and Future Research

DIP Framework

DIP Framework

DIP (**D**ecomposition for **I**nteger **P**rogramming) is an open-source software framework that provides an implementation of various decomposition methods with minimal user responsibility

- Allows direct comparison CPM/DW/LD/PC/RC/DC in one framework
- DIP abstracts the common, generic elements of these methods
- **Key:** The user defines application-specific components in the space of the compact formulation - greatly simplifying the API
  - Define $[A'', b'']$ and/or $[A', b']$
  - Provide methods for $\mathrm{OPT}(\mathcal{P}', c)$ and/or $\mathrm{SEP}(\mathcal{P}', x)$
- Framework handles all of the algorithm-specific reformulation



DIP Framework: Implementation

COmputational INfrastructure for Operations Research
Have some DIP with your CHiPPS?

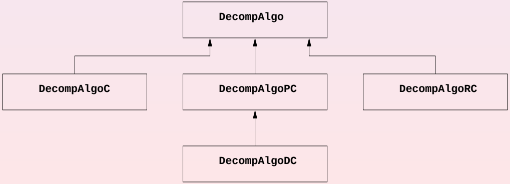
- **DIP** was built around data structures and interfaces provided by COIN-OR
- The **DIP** framework, written in C++, is accessed through two user interfaces:
  - Applications Interface: `DecompApp`
  - Algorithms Interface: `DecompAlgo`
- **DIP** provides the bounding method for branch and bound
- **ALPS** (Abstract Library for Parallel Search) provides the framework for tree search
  - `AlpsDecompModel : public AlpsModel`
    - a wrapper class that calls (data access) methods from `DecompApp`
  - `AlpsDecompTreeNode : public AlpsTreeNode`
    - a wrapper class that calls (algorithmic) methods from `DecompAlgo`

DIP Framework: Applications API

- The base class DecompApp provides an interface for user to define the application-specific components of their algorithm
- Define the model(s)
  - `setModelObjective(double * c)`: define c
  - `setModelCore(DecompConstraintSet * model)`: define $\mathcal{Q}''$
  - `setModelRelaxed(DecompConstraintSet * model, int block)`: define $\mathcal{Q}'$ [optional]
- `solveRelaxed()`: define a method for $\mathrm{OPT}(\mathcal{P}', c)$ [optional, if $\mathcal{Q}'$, **CBC** is built-in]
- `generateCuts()`: define a method for $\mathrm{SEP}(\mathcal{P}', x)$ [optional, **CGL** is built-in]
- `isUserFeasible()`: is $\hat{x} \in \mathcal{P}$? [optional, if $\mathcal{P} = \mathrm{conv}(\mathcal{P}' \cap \mathcal{Q}'' \cap \mathbb{Z})$]
- All other methods have appropriate defaults but are `virtual` and may be overridden

DIP Framework: Algorithm API

- The base class `DecompAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described has derived default implementations `DecompAlgoX : public DecompAlgo` which are accessible by any application class, allowing full flexibility.
- New, hybrid or extended methods can be easily derived by overriding the various subroutines, which are called from the base class. For example,
  - Alternative methods for solving the master LP in DW, such as **interior point methods**
  - Add stabilization to the dual updates in LD (stability centers)
  - For LD, replace subgradient with **volume** providing an approximate primal solution
  - Hybrid init methods like using LD or DC to initialize the columns of the DW master
  - During PC, adding cuts to either master and/or subproblem.
  - ...



DIP Framework: Feature Overview

- One interface to all algorithms: **CP/DC, DW, LD, PC, RC**. Change aapproach by switching parameters.
- Automatic reformulation allows users to specify methods in the compact (original) space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Novel options for cut generation
  - Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Can utilize *structured separation* (efficient algorithms that apply only to vectors with special structure (integer) in various ways.
  - Can separate from $\mathcal{P}'$ using subproblem solver (DC).
- Easy to combine different approaches
  - Column generation based on *multiple algorithms* or *nested subproblems* can be easily defined and employed.
  - Bounds based on *multiple model/algorithm* combinations.
- Provides generic (naive) branching rules,
- Active LP compression, variable and cut pool management. overrides.
- Fully generic algorithm for problems with block structure.
  - Automatic detection of blocks.
  - Threaded oracle.
  - No coding required.

Working in the Compact Space

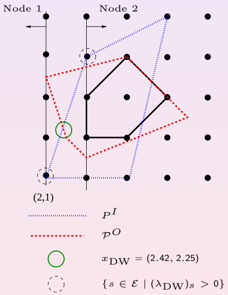
- The key to the implementation of this unified framework is that we always maintain a representation of the problem in the compact space.
- This allows us to employ most of the usual techniques used in LP-based branch and bound without modification, even in this more general setting.
- There are some challenges related to this approach that we are still working on.
  - Gomory cuts
  - Preprocessing
  - Identical subproblems
  - Strong branching
- Allowing the user to express all methods in the compact space is extremely powerful when it comes to modeling language support.
- It is important to note that DIP currently assumes the existence of a formulation in the compact space.
- We are working on relaxing this assumption, but this means the loss of the fully generic implementation of some techniques.

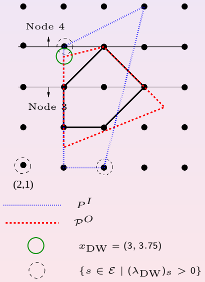
Branching

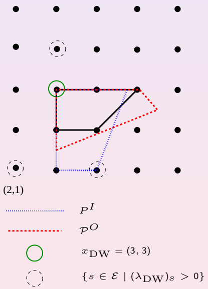
- By default, we branch on variables in the compact space.
- In PC, this is done by mapping back to the compact space $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$.
- Variable branching in the compact space is constraint branching in the extended space
- This idea makes it possible define generic branching procedures.

Branching

- By default, we branch on variables in the compact space.
- In PC, this is done by mapping back to the compact space $\hat{x} = \sum_{s \in \mathcal{E}} s \hat{\lambda}_s$.
- Variable branching in the compact space is constraint branching in the extended space
- This idea makes it possible define generic branching procedures.

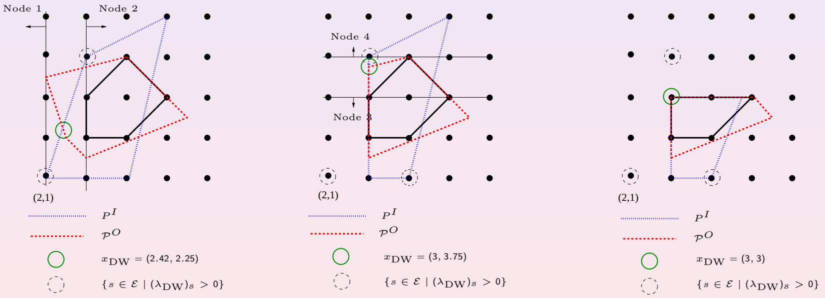






Branching

- By default, we branch on variables in the compact space.
- In PC, this is done by mapping back to the compact space $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$.
- Variable branching in the compact space is constraint branching in the extended space
- This idea makes it possible define generic branching procedures.



$$\begin{array}{lccc} \text{Node 1:} & 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} & \leq & 2 \\ \text{Node 2:} & 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} & \geq & 3 \end{array}$$

Branching for RC

- In general, Lagrangian methods do *not* provide a primal solution λ
- Let $\mathcal{B}$ define the extreme points found in solving subproblems for z_{LD}
- Build an inner approximation using this set, then proceed as in PC

$$\mathcal{P}_I = \left\{ x \in \mathbb{R}^n \,\middle|\, x = \sum_{s \in \mathcal{B}} s\lambda_s, \sum_{s \in \mathcal{B}} \lambda_s = 1, \lambda_s \geq 0 \ \forall s \in \mathcal{B} \right\}$$

$$\min_{\lambda \in \mathbb{R}^{\mathcal{B}}_+} \left\{ c^\top \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \,\middle|\, A'' \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{B}} \lambda_s = 1 \right\}$$

- Closely related to *volume* algorithm and *bundle* methods

Branching for RC

- In general, Lagrangian methods do *not* provide a primal solution λ
- Let $\mathcal{B}$ define the extreme points found in solving subproblems for z_{LD}
- Build an inner approximation using this set, then proceed as in PC

$$\mathcal{P}_I = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{B}} s\lambda_s, \sum_{s \in \mathcal{B}} \lambda_s = 1, \lambda_s \geq 0 \ \forall s \in \mathcal{B} \right\}$$

$$\min_{\lambda \in \mathbb{R}^{\mathcal{B}}_+} \left\{ c^\top \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{B}} \lambda_s = 1 \right\}$$

- Closely related to *volume* algorithm and *bundle* methods

Branching for RC

- In general, Lagrangian methods do *not* provide a primal solution λ
- Let $\mathcal{B}$ define the extreme points found in solving subproblems for z_{LD}
- Build an inner approximation using this set, then proceed as in PC

$$\mathcal{P}_I = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{B}} s\lambda_s, \sum_{s \in \mathcal{B}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{B} \right\}$$

$$\min_{\lambda \in \mathbb{R}_+^{\mathcal{B}}} \left\{ c^\top \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{B}} \lambda_s = 1 \right\}$$

- Closely related to *volume* algorithm and *bundle* methods

Algorithmic Details

- **Performance improvements**
  - Detection and removal of columns that are close to parallel
  - Basic dual stabilization (Wentges smoothing)
  - Redesign (and simplification) of treatment of master-only variables.
- **New features and enhancements**
  - Branching can be auto enforced in subproblem or master (when oracle is MILP)
  - Ability to stop subproblem calculation on gap/time and calculate LB (can branch early)
  - For oracles that provide it, allow multiple columns for each subproblem call
  - Management of compression of columns once master gap is tight
- **Use of generic MILP solution technology**
  - Using the mapping $\hat{x} = \sum_{s \in \mathcal{E}} s \hat{\lambda}_s$ we can import any generic MILP tehcnique to the PC/RC context.
  - Use generic MILP solver to solve subproblems.
  - Hooks to define branching methods, heuristics, etc.
- **Algorithms for generating initial columns**
  - Solve $\mathrm{OPT}(\mathcal{P}', c + r)$ for random perturbations
  - Solve $\mathrm{OPT}(\mathcal{P}_N)$ heuristically
  - Run several iterations of LD or DC collecting extreme points

Algorithmic Details

- **Performance improvements**
  - Detection and removal of columns that are close to parallel
  - Basic dual stabilization (Wentges smoothing)
  - Redesign (and simplification) of treatment of master-only variables.
- **New features and enhancements**
  - Branching can be auto enforced in subproblem or master (when oracle is MILP)
  - Ability to stop subproblem calculation on gap/time and calculate LB (can branch early)
  - For oracles that provide it, allow multiple columns for each subproblem call
  - Management of compression of columns once master gap is tight
- **Use of generic MILP solution technology**
  - Using the mapping $\hat{x} = \sum_{s \in \mathcal{E}} s \hat{\lambda}_s$ we can import any generic MILP tehcnique to the PC/RC context.
  - Use generic MILP solver to solve subproblems.
  - Hooks to define branching methods, heuristics, etc.
- **Algorithms for generating initial columns**
  - Solve $\mathrm{OPT}(\mathcal{P}', c + r)$ for random perturbations
  - Solve $\mathrm{OPT}(\mathcal{P}_N)$ heuristically
  - Run several iterations of LD or DC collecting extreme points

Algorithmic Details

- **Performance improvements**
  - Detection and removal of columns that are close to parallel
  - Basic dual stabilization (Wentges smoothing)
  - Redesign (and simplification) of treatment of master-only variables.
- **New features and enhancements**
  - Branching can be auto enforced in subproblem or master (when oracle is MILP)
  - Ability to stop subproblem calculation on gap/time and calculate LB (can branch early)
  - For oracles that provide it, allow multiple columns for each subproblem call
  - Management of compression of columns once master gap is tight
- **Use of generic MILP solution technology**
  - Using the mapping $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$ we can import any generic MILP tehcnique to the PC/RC context.
  - Use generic MILP solver to solve subproblems.
  - Hooks to define branching methods, heuristics, etc.
- **Algorithms for generating initial columns**
  - Solve $\mathrm{OPT}(\mathcal{P}', c + r)$ for random perturbations
  - Solve $\mathrm{OPT}(\mathcal{P}_N)$ heuristically
  - Run several iterations of LD or DC collecting extreme points

Algorithmic Details

- **Performance improvements**
  - Detection and removal of columns that are close to parallel
  - Basic dual stabilization (Wentges smoothing)
  - Redesign (and simplification) of treatment of master-only variables.
- **New features and enhancements**
  - Branching can be auto enforced in subproblem or master (when oracle is MILP)
  - Ability to stop subproblem calculation on gap/time and calculate LB (can branch early)
  - For oracles that provide it, allow multiple columns for each subproblem call
  - Management of compression of columns once master gap is tight
- **Use of generic MILP solution technology**
  - Using the mapping $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$ we can import any generic MILP tehcnique to the PC/RC context.
  - Use generic MILP solver to solve subproblems.
  - Hooks to define branching methods, heuristics, etc.
- **Algorithms for generating initial columns**
  - Solve $\mathrm{OPT}(\mathcal{P}', c + r)$ for random perturbations
  - Solve $\mathrm{OPT}(\mathcal{P}_N)$ heuristically
  - Run several iterations of LD or DC collecting extreme points

Algorithmic Details (cont.)

- **Choice of master LP solver**
  - Dual simplex after adding rows or adjusting bounds (warm-start dual feasible)
  - Primal simplex after adding columns (warm-start primal feasible)
  - Interior-point methods might help with stabilization vs extremal duals
- **Price-and-branch heuristic**
  - For block-angular case, at end of each node, solve with $\lambda \in \mathbb{Z}$
  - Used in *root node* by Barahona and Jensen ('98), we extend to tree
- **Compression of master LP and object pools**: Reduce size of master LP, improve efficiency of subproblem processing.
- **Nested pricing**: Can solve more constrained versions of subproblem heuristically to get high quality columns.
- **Interfaces for Pricing Algorithms** (for IBM Project)
  - User can provide an initial dual vector
  - User can manipulate duals used at each pass (and specify per block)
  - User can select which block to process next (alternative to *all* or *round-robin*)

Algorithmic Details (cont.)

- **Choice of master LP solver**
  - Dual simplex after adding rows or adjusting bounds (warm-start dual feasible)
  - Primal simplex after adding columns (warm-start primal feasible)
  - Interior-point methods might help with stabilization vs extremal duals
- **Price-and-branch heuristic**
  - For block-angular case, at end of each node, solve with $\lambda \in \mathbb{Z}$
  - Used in *root node* by Barahona and Jensen ('98), we extend to tree
- **Compression of master LP and object pools**: Reduce size of master LP, improve efficiency of subproblem processing.
- **Nested pricing**: Can solve more constrained versions of subproblem heuristically to get high quality columns.
- **Interfaces for Pricing Algorithms** (for IBM Project)
  - User can provide an initial dual vector
  - User can manipulate duals used at each pass (and specify per block)
  - User can select which block to process next (alternative to *all* or *round-robin*)

Algorithmic Details (cont.)

- **Choice of master LP solver**
  - Dual simplex after adding rows or adjusting bounds (warm-start dual feasible)
  - Primal simplex after adding columns (warm-start primal feasible)
  - Interior-point methods might help with stabilization vs extremal duals
- **Price-and-branch heuristic**
  - For block-angular case, at end of each node, solve with $\lambda \in \mathbb{Z}$
  - Used in *root node* by Barahona and Jensen ('98), we extend to tree
- **Compression of master LP and object pools**: Reduce size of master LP, improve efficiency of subproblem processing.
- **Nested pricing**: Can solve more constrained versions of subproblem heuristically to get high quality columns.
- **Interfaces for Pricing Algorithms** (for IBM Project)
  - User can provide an initial dual vector
  - User can manipulate duals used at each pass (and specify per block)
  - User can select which block to process next (alternative to *all* or *round-robin*)

Algorithmic Details (cont.)

- **Choice of master LP solver**
  - Dual simplex after adding rows or adjusting bounds (warm-start dual feasible)
  - Primal simplex after adding columns (warm-start primal feasible)
  - Interior-point methods might help with stabilization vs extremal duals
- **Price-and-branch heuristic**
  - For block-angular case, at end of each node, solve with $\lambda \in \mathbb{Z}$
  - Used in *root node* by Barahona and Jensen ('98), we extend to tree
- **Compression of master LP and object pools**: Reduce size of master LP, improve efficiency of subproblem processing.
- **Nested pricing**: Can solve more constrained versions of subproblem heuristically to get high quality columns.
- **Interfaces for Pricing Algorithms** (for IBM Project)
  - User can provide an initial dual vector
  - User can manipulate duals used at each pass (and specify per block)
  - User can select which block to process next (alternative to *all* or *round-robin*)

Algorithmic Details (cont.)

- **Choice of master LP solver**
  - Dual simplex after adding rows or adjusting bounds (warm-start dual feasible)
  - Primal simplex after adding columns (warm-start primal feasible)
  - Interior-point methods might help with stabilization vs extremal duals
- **Price-and-branch heuristic**
  - For block-angular case, at end of each node, solve with $\lambda \in \mathbb{Z}$
  - Used in *root node* by Barahona and Jensen ('98), we extend to tree
- **Compression of master LP and object pools**: Reduce size of master LP, improve efficiency of subproblem processing.
- **Nested pricing**: Can solve more constrained versions of subproblem heuristically to get high quality columns.
- **Interfaces for Pricing Algorithms** (for IBM Project)
  - User can provide an initial dual vector
  - User can manipulate duals used at each pass (and specify per block)
  - User can select which block to process next (alternative to *all* or *round-robin*)

DIP Framework: Example Applications

Application	Description	$\mathcal{P}'$	$\mathbf{OPT}(c)$	$\mathbf{SEP}(x)$	Input
AP3	3-index assignment	AP	Jonker	user	user
ATM	cash management (SAS COE)	MILP(s)	CBC	CGL	user
GAP	generalized assignment	KP(s)	Pisinger	CGL	user
MAD	matrix decomposition	MaxClique	Cliquer	CGL	user
MILP	random partition into A', A''	MILP	CBC	CGL	mps
MILPBlock	user-defined blocks for A'	MILP(s)	CBC	CGL	mps, block
MMKP	multi-dim/choice knapsack	MCKP	Pisinger	CGL	user
		MDKP	CBC	CGL	user
SILP	intro example, tiny IP	MILP	CBC	CGL	user
TSP	traveling salesman problem	1-Tree	Boost	Concorde	user
		2-Match	CBC	Concorde	user
VRP	vehicle routing problem	k-TSP	Concorde	CVRPSEP	user
		b-Match	CBC	CVRPSEP	user

Outline

DIPPY

- DIPPY provides an interface to DIP through the modeling language PuLP.
- PuLP is a modeling language that provides functionality similar to other modeling languages.
- It is built on top of Python so you get the full power of that language for free.
- PuLP and DIPPY are being developed by Stuart Mitchell and Mike O'Sullivan in Auckland and are part of COIN.
- Through DIPPY, a user can
  - Specify the model and the relaxation, including the block structure.
  - Implement methods (coded in Python) for solving the relaxation, generating cuts, custom branching.
- With Dippy, it is possible to code a customized column-generation method from scratch in a few hours.
- This would have taken months with previously available tools.

Example: Facility Location Problem

- We are given n facility locations and m customers to be serviced from those locations.
- There is a fixed cost c_j and a capacity W_j associated with facility j.
- There is a cost d_{ij} and demand w_{ij} associated with serving customer i from facility j.
- We have two sets of binary variables.
  - y_j is 1 if facility j is opened, 0 otherwise.
  - x_{ij} is 1 if customer i is served by facility j, 0 otherwise.

Capacitated Facility Location Problem

$$
\begin{array}{lll}
\min & \sum_{j=1}^{n} c_j y_j + \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij} x_{ij} & \\
\text{s.t.} & \sum_{j=1}^{n} x_{ij} = 1 & \forall i \\
& \sum_{i=1}^{m} w_{ij} x_{ij} \leq W_j & \forall j \\
& x_{ij} \leq y_j & \forall i, j \\
& x_{ij}, y_j \in \{0, 1\} & \forall i, j
\end{array}
$$

DIPPY Code for Facility Location

DIPPY

```
from facility_data import REQUIREMENT, PRODUCTS, LOCATIONS, CAPACITY

prob = dippy.DipProblem("Facility_Location")

assign = LpVariable.dicts("Assignment", [(i, j) for i in LOCATIONS for
                          j in PRODUCTS], 0, 1, LpBinary)
open   = LpVariable.dicts("FixedCharge", LOCATIONS, 0, 1, LpBinary)

# objective: minimise waste
prob += lpSum(excess[i] for i in LOCATIONS), "min"

# assignment constraints
for j in PRODUCTS:
    prob += lpSum(assign[(i, j)] for i in LOCATIONS) == 1

# Aggregate capacity constraints
for i in LOCATIONS:
    prob.relaxation[i] += lpSum(assign[(i, j)]*REQUIREMENT[j] for j in
                                PRODUCTS) + excess[i] == CAPACITY * open[i]

# Disaggregated capacity constraints
for i in LOCATIONS:
    for j in PRODUCTS:
        prob.relaxation[i] += assign[(i, j)] <= open[i]

# Ordering constraints
for index, location in enumerate(LOCATIONS):
    if index > 0:
        prob += use[LOCATIONS[index-1]] >= open[location]
```

DIPPY Auxiliary Methods for Facility Location

DIPPY

```
def solve_subproblem(prob, index, redCosts, convexDual):
   ...
   z, solution = knapsack01(obj, weights, CAPACITY)
   ...
   return []
prob.relaxed_solver = solve_subproblem
def knapsack01(obj, weights, capacity):
    ...
    return c[n-1][capacity], solution
def first_fit(prob):
    ...
    return bvs
def one_each(prob):
   ...
   return bvs
prob.init_vars = first_fit
def choose_antisymmetry_branch(prob, sol):
   ...
   return ([], down_branch_ub, up_branch_lb, [])
prob.branch_method = choose_antisymmetry_branch
def generate_weight_cuts(prob, sol):
    ...
    return new_cuts
prob.generate_cuts = generate_weight_cuts
def heuristics(prob, xhat, cost):
    ...
    return sols
prob.heuristics = heuristics
dippy.Solve(prob, {
    'doPriceCut': '1',
})
```

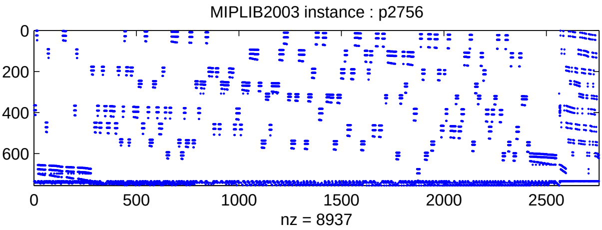
MILPBlock: Decomposition-based MILP Solver

- Many difficult MILPs have a block structure, but this structure is not part of the input (MPS) or is not exploitable by the solver.
- In practice, it is common to have models composed of independent subsystems coupled by global constraints.
- The result may be models that are highly symmetric and difficult to solve using traditional methods, but would be easy to solve if the structure were known.

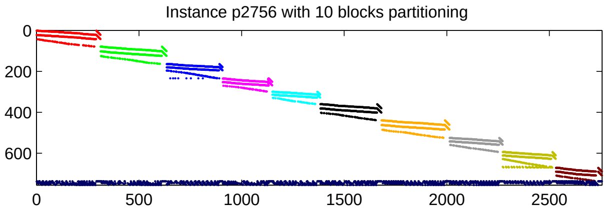
$$
\begin{pmatrix}
A_1'' & A_2'' & \cdots & A_\kappa'' \\
\hline
A_1' & & & \\
 & A_2' & & \\
 & & \ddots & \\
 & & & A_\kappa'
\end{pmatrix}
$$

- MILPBlock provides a black-box solver for applying integrated methods to generic MILP
- Input is an MPS/LP and a *block file* specifying structure.
- Optionally, the block file can be automatically generated using the hypergraph partitioning algorithm of HMetis.
- This is the engine underlying DIPPY.

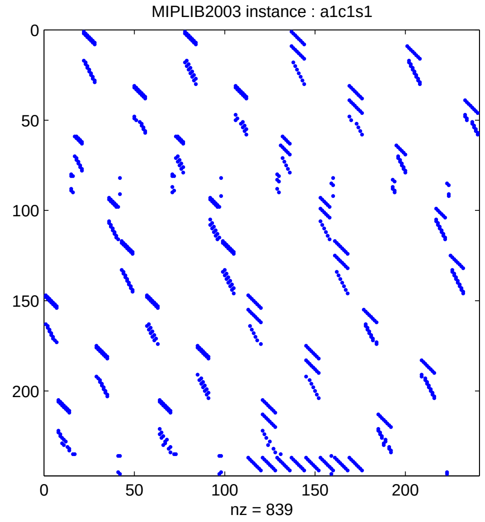
Hidden Block Structure



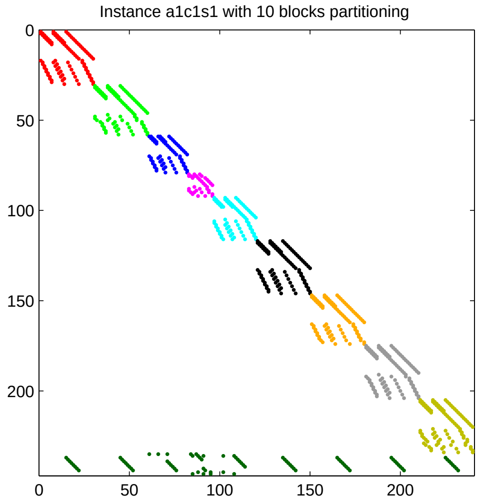
Hidden Block Structure



Hidden Block Structure



Hidden Block Structure



Bound Improvement

insta	cols	rows	opt	k	DWR bound	CBC root
10teams	2025	230	924	3	918.1	917
noswot	128	182	563.8	3	-41.2	-43
p2756	2756	755	3124	3	3115.5	2688.7
timtab1	397	171	764772	3	350885	28694
timtab2	675	294	1096560	3	431963	83592
vpm2	378	234	13.7	3	12.2	9.8
pg5_34	2600	125	-14339.4	3	-15179.2	-16646.5
pg	2700	125	-8674.34	3	-15179.2	-16646.5
k16x240	480	256	10674	3	3303.6	2769.8

Application - Block-Angular MILP (applied to Retail Optimization)

SAS Retail Optimization Solution

- *Multi-tiered supply chain distribution problem* where each block represents a store
- Prototype model developed in SAS/OR's OPTMODEL (algebraic modeling language)

	CPX11			DIP-PC		
Instance	**Time**	**Gap**	**Nodes**	**Time**	**Gap**	**Nodes**
retail27	T	2.30%	2674921	3.18	OPT	1
retail31	T	0.49%	1434931	767.36	OPT	41
retail3	529.77	OPT	2632157	0.54	OPT	1
retail4	T	1.61%	1606911	116.55	OPT	1
retail6	1.12	OPT	803	264.59	OPT	303

Outline

Related Projects Currently using DIP

- **OSDip** – Optimization Services (**OS**) wraps DIP
  - University of Chicago – Kipp Martin
- **Dippy** – Python interface for **DIP** through **PuLP**
  - University of Auckland – Michael O'Sullivan
- **SAS** – DIP-like solver for `PROC OPTMODEL`
  - SAS Institute – Matthew Galati
- **National Workforce Management, Cross-Training and Scheduling Project**
  - IBM Business Process Re-engineering – Alper Uygur
- **Transmission Switching Problem for Electricity Networks**
  - University of Denmark – Jonas Villumsem
  - University of Auckland – Andy Philipott

DIP@SAS in PROC OPTMODEL

- Prototype **PC** algorithm embedded in PROC OPTMODEL (based on MILPBlock)
- Minor API change - one new suffix on rows *or* cols (`.block`)

Preliminary Results (Recent Clients):

Client Problem	IP-GAP DIP@SAS	IP-GAP CPX12.1	Real-Time DIP@SAS	Real-Time CPX12.1
ATM Cash Management and Predictive Model (India)	OPT	∞	103	2000 (T)
ATM Cash Management (Singapore)	OPT	OPT	86	831
	OPT	OPT	90	783
Retail Inventory Optimization (UK)	1.6%	9%	1200	1200 (T)
	4.7%	19%	1200	1200 (T)
	2.6%	∞	1200	1200 (T)

Current Research

- **Block structure** (Important!)
  - Identical subproblems for eliminating symmetry
  - Better automatic detection
- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts
- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC
- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)
- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - **Gomory cuts** in Price-and-Cut

Current Research

- **Block structure** (Important!)
  - Identical subproblems for eliminating symmetry
  - Better automatic detection
- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts
- **Branch-and-Relax-and-Cut:** Computational focus thus far has been on CPM/DC/PC
- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)
- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - **Gomory cuts** in Price-and-Cut

Current Research

- **Block structure** (Important!)
  - Identical subproblems for eliminating symmetry
  - Better automatic detection
- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts
- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC
- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)
- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - **Gomory cuts** in Price-and-Cut

Current Research

- **Block structure** (Important!)
  - Identical subproblems for eliminating symmetry
  - Better automatic detection
- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts
- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC
- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)
- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - Gomory cuts in Price-and-Cut

Current Research

- **Block structure** (Important!)
  - Identical subproblems for eliminating symmetry
  - Better automatic detection
- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts
- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC
- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)
- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - **Gomory cuts** in Price-and-Cut