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

\[
\begin{align*}
  z_{\text{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
  z_{\text{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}
\end{align*}
\]
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:

- Remove a subproblem and solve its relaxation.
- The relaxation is infeasible $\Rightarrow$ subproblem is infeasible and can be pruned.
- Solution is feasible for the MILP $\Rightarrow$ subproblem solved (update upper bound).
- 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**

\[
\begin{align*}
    x(\delta(\{u\})) &= 2 \quad \forall u \in V \\
    x(E(S)) &\leq |S| - 1 \quad \forall S \subset V, \ 3 \leq |S| \leq |V| - 1 \\
    x_e &\in \{0, 1\} \quad \forall e \in E
\end{align*}
\]
Example: Exposing Combinatorial Structure

Traveling Salesman Problem Formulation

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

Two relaxations

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

\[
\begin{align*}
  x(\delta(\{0\})) &= 2 \\
  x(E(V)) &= |V| \\
  x(E(S)) &\leq |S| - 1 & \forall S \subset V \setminus \{0\}, \ 3 \leq |S| \leq |V| - 1 \\
  x_e &\in \{0, 1\} & \forall e \in E
\end{align*}
\]
Example: Exposing Combinatorial Structure

**Traveling Salesman Problem Formulation**

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

**Two relaxations**

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

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

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

\[
\begin{align*}
x(\delta(\{u\})) &= 2 \quad \forall u \in V \\
x_e &\in \{0, 1\} \quad \forall e \in E
\end{align*}
\]
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*. 

\[
\begin{pmatrix}
A''_1 & A''_2 & \cdots & A''_\kappa \\
A'_1 & A'_2 & \cdots & A'_\kappa \\
\vdots & \vdots & \ddots & \vdots \\
A'_1 & A'_2 & \cdots & A'_\kappa \\
\end{pmatrix}
\]
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.

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

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

- 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)

\[
\begin{align*}
\text{min} & \quad \sum_{k \in M} \sum_{(i,j) \in A} c_{ij} x_{ijk} \\
& \quad \sum_{k \in M} \sum_{j \in N} x_{ijk} = 1 \quad \forall i \in V \\
& \quad \sum_{i \in V} \sum_{j \in N} d_{ij} x_{ijk} \leq C \quad \forall k \in M \\
& \quad \sum_{j \in N} x_{0jk} = 1 \quad \forall k \in M \\
& \quad \sum_{i \in N} x_{ihk} - \sum_{j \in N} x_{hjk} = 0 \quad \forall h \in V, k \in M \\
& \quad \sum_{i \in N} x_{i,n+1,k} = 1 \quad \forall k \in M \\
& \quad x_{ijk} \in \{0, 1\} \quad \forall (i, j) \in A, k \in M
\end{align*}
\]
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.
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 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.

\[
\begin{align*}
\mathbf{z}_{IP} & = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
\mathbf{z}_{LP} & = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
\mathbf{z}_D & = \min_{x \in \mathcal{P}} \left\{ c^\top x \mid A''x \geq b'' \right\} \\
\mathbf{z}_{IP} & \geq \mathbf{z}_D \geq \mathbf{z}_{LP}
\end{align*}
\]

**Assumptions:**
- \( \text{OPT}(\mathcal{P}, c) \) and \( \text{SEP}(\mathcal{P}, x) \) are “hard”
- \( \text{OPT}(\mathcal{P}', c) \) and \( \text{SEP}(\mathcal{P}', x) \) are “easy”
- \( Q'' \) 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.

\[
\begin{align*}
z_{\text{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{\text{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{\text{D}} &= \min_{x \in \mathcal{P'}} \left\{ c^\top x \mid A''x \geq b'' \right\}
\end{align*}
\]

\[z_{\text{IP}} \geq z_{\text{D}} \geq z_{\text{LP}}\]

Assumptions:
- OPT(\mathcal{P}, c) and SEP(\mathcal{P}, x) are “hard”
- OPT(\mathcal{P}', c) and SEP(\mathcal{P}', x) are “easy”
- \(Q''\) can be represented explicitly (description has polynomial size)
- \(\mathcal{P}'\) must be represented implicitly (description has exponential size)

\[Q' = \{x \in \mathbb{R}^n \mid A'x \geq b'\}\]
\[Q'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}\]
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.

\[
\begin{align*}
 z_{\text{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A' x \geq b', A'' x \geq b'' \right\} \\
 z_{\text{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A' x \geq b', A'' x \geq b'' \right\} \\
 z_{\text{D}} &= \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A'' x \geq b'' \right\} \\
 z_{\text{IP}} &\geq z_{\text{D}} \geq z_{\text{LP}}
\end{align*}
\]

**Assumptions:**
- \( \text{OPT}(\mathcal{P}, c) \) and \( \text{SEP}(\mathcal{P}, x) \) are “hard”
- \( \text{OPT}(\mathcal{P}', c) \) and \( \text{SEP}(\mathcal{P}', x) \) are “easy”
- \( Q'' \) 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.

\[
\begin{align*}
z_{IP} &= \min_{x \in \mathbb{Z}^n} \left\{ c^T x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{LP} &= \min_{x \in \mathbb{R}^n} \left\{ c^T x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_D &= \min_{x \in \mathcal{P}'} \left\{ c^T x \mid A''x \geq b'' \right\}
\end{align*}
\]

\[z_{IP} \geq z_D \geq z_{LP}\]

**Assumptions:**
- \( \text{OPT}(\mathcal{P}, c) \) and \( \text{SEP}(\mathcal{P}, x) \) are "hard"
- \( \text{OPT}(\mathcal{P}', c) \) and \( \text{SEP}(\mathcal{P}', x) \) are "easy"
- \( Q'' \) 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.

\[
\begin{align*}
z_{\text{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{\text{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\} \\
z_{\text{D}} &= \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}
\end{align*}
\]

\[z_{\text{IP}} \geq z_{\text{D}} \geq z_{\text{LP}}\]

**Assumptions:**
- \(\text{OPT}(\mathcal{P}, c)\) and \(\text{SEP}(\mathcal{P}, x)\) are “hard”
- \(\text{OPT}(\mathcal{P}', c)\) and \(\text{SEP}(\mathcal{P}', x)\) are “easy”
- \(Q''\) 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:** $z_{\text{CP}} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid Dx \geq d, A''x \geq b'' \}$
- **Subproblem:** $\text{SEP}(\mathcal{P}', x_{\text{CP}})$

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

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

- **Master:** $z_{\text{CP}} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid Dx \geq d, A''x \geq b'' \}$
- **Subproblem:** $\text{SEP}(\mathcal{P}', x_{\text{CP}})$

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

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

- **Master:** $z_{CP} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid Dx \geq d, A''x \geq b'' \}$
- **Subproblem:** $\text{SEP}(\mathcal{P}', x_{CP})$

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

*Exponential number of constraints*
**Cutting Plane Method (CPM)**

**CPM** combines an *outer* approximation of \( P' \) with an explicit description of \( Q'' \)

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

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

*Exponential number of constraints*

---

\( P_0 = Q' \cap Q'' \)

\( x_{CP}^0 = (2.25, 2.75) \)

\( P_0^1 = P_0 \cap \{ x \in \mathbb{R}^n \mid 3x_1 - x_2 \geq 5 \} \)

\( x_{CP}^1 = (2.42, 2.25) \)
**Dantzig-Wolfe Method (DW)**

**DW** combines an *inner* approximation of \( P' \) with an explicit description of \( Q'' \)

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

\[ P' = \left\{ x \in \mathbb{R}^n \mid x = \sum_{s \in \mathcal{E}} s \lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \ \forall s \in \mathcal{E} \right\} \]

*Exponential number of variables*
The Dantzig-Wolfe Method (DW) combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$.

**Master:**

\[
\begin{align*}
 z_{\text{DW}} &= \min_{\lambda \in \mathbb{R}^E} \left\{ c^\top \left( \sum_{s \in \mathcal{E}} s \lambda_s \right) \mid A'' \left( \sum_{s \in \mathcal{E}} s \lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\} \\
\end{align*}
\]

**Subproblem:**

\[
\begin{align*}
 \text{OPT} \left( \mathcal{P}', c^\top - u_{\text{DW}}^\top A'' \right)
\end{align*}
\]

\[
\mathcal{P}' = \left\{ x \in \mathbb{R}^n \mid x = \sum_{s \in \mathcal{E}} s \lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \forall s \in \mathcal{E} \right\}
\]

*Exponential number of variables*
Dantzig-Wolfe Method (DW)

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

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

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

*Exponential number of variables*
Dantzig-Wolfe Method (DW)

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

- **Master:** $z_{\text{DW}} = \min_{\lambda \in \mathbb{R}_+^E} \left\{ c^T \left( \sum_{s \in \mathcal{E}} s \lambda_s \right) \mid A'' \left( \sum_{s \in \mathcal{E}} s \lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$

- **Subproblem:** $\text{OPT} \left( \mathcal{P}', c^T - u_{\text{DW}}^T A'' \right)$

\[
\mathcal{P}' = \left\{ x \in \mathbb{R}^n \mid x = \sum_{s \in \mathcal{E}} s \lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \ \forall s \in \mathcal{E} \right\}
\]

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

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

- **Subproblem:** $\text{OPT} (\mathcal{P}', c^\top - u_{LD}^\top A'')$

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

- **Subproblem:**
  $$\text{OPT} (\mathcal{P}', c^\top - u_{LD}^\top A'')$$

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

- **Subproblem:**  
  \[ \text{OPT} \left( \mathcal{P}', c^\top - u_{LD}^\top A'' \right) \]

\[ z_{LD} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}_+^{m''}} \left\{ \alpha + b''^\top u \mid (c^\top - u^\top A'') s - \alpha \geq 0 \ \forall s \in E \right\} = z_{DW} \]
Common Threads

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

\[
    z_{LP} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in Q' \cap Q''\}
\]

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

\[
    z_{CP} = z_{DW} = z_{LD} = z_D = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in P' \cap Q''\} \geq z_{LP}
\]

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

**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.

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

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

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

Traditional decomp-based bounding methods contain two primary steps

- Master Problem: Update the primal/dual solution information
- Subproblem: Update the approximation of $P'$: $\text{SEP}(P', x)$ or $\text{OPT}(P', c)$

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.

  \[ z_{\text{LP}} = \min_{x \in \mathbb{R}^n} \{ c^\top x \mid x \in Q' \cap Q'' \} \]

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

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

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

- 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.

\[
    z_{LP} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in Q' \cap Q'' \}
\]

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

\[
    z_{CP} = z_{DW} = z_{LD} = z_D = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in P' \cap Q'' \} \geq z_{LP}
\]

Traditional decomp-based bounding methods contain two primary steps

- **Master Problem**: Update the primal/dual solution information
- **Subproblem**: Update the approximation of \( P' \): SEP(\( P' \), \( x \)) or OPT(\( P' \), \( c \))

**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 $\mathcal{P}'$

$$\min_{\lambda \in \mathbb{R}_+^E, (x^+, x^-) \in \mathbb{R}_{++}^n} \left\{ x^+ + x^- \left| \sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}_{CP}, \sum_{s \in \mathcal{E}} \lambda_s = 1 \right. \right\}$$
Decompose-and-Cut (DC)

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

\[
\min_{\lambda \in \mathbb{R}_{+}^{E}, (x^+, x^-) \in \mathbb{R}_{+}^{n}} \left\{ \begin{array}{l}
x^+ + x^- \\
\sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}_{CP}, \\
\sum_{s \in \mathcal{E}} \lambda_s = 1
\end{array} \right. \]

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

**Decomposition Cuts**

\[
u^t_{DC} s + \alpha^t_{DC} \leq 0 \ \forall s \in \mathcal{P}' \text{ and } u^t_{DC} \hat{x}_{CP} + \alpha^t_{DC} > 0
\]
Decompose-and-Cut (DC)

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

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

- 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
  - $\hat{x}_i = 1$ 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** (Decomposition for Integer Programming) 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 \(\text{OPT}(\mathcal{P}', c)\) and/or \(\text{SEP}(\mathcal{P}', x)\)
- 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 \texttt{DecompApp} provides an interface for user to define the application-specific components of their algorithm.

- Define the model(s)
  - \texttt{setModelObjective(double * c)}: define \( c \)
  - \texttt{setModelCore(DecompConstraintSet * model)}: define \( Q'' \)
  - \texttt{setModelRelaxed(DecompConstraintSet * model, int block)}: define \( Q' \) [optional]

- \texttt{solveRelaxed()}: define a method for \( \text{OPT}(P', c) \) [optional, if \( Q' \), \texttt{CBC} is built-in]

- \texttt{generateCuts()}: define a method for \( \text{SEP}(P', x) \) [optional, \texttt{CGL} is built-in]

- \texttt{isUserFeasible()}: \( \hat{x} \in \mathcal{P} \)? [optional, if \( \mathcal{P} = \text{conv}(P' \cap Q'' \cap \mathbb{Z}) \)]

- All other methods have appropriate defaults but are \texttt{virtual} and may be overridden.
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.

```plaintext
DecompAlgoDC
DecompAlgoC
DecompAlgoPC
DecompAlgoRC
DecompAlgoDC
```
DIP Framework: Feature Overview

- One interface to all algorithms: CP/DC, DW, LD, PC, RC. Change approach 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 $P'$ 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.

Ralphs, Galati, Wang
Decomposition Methods for Integer Linear Programming
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 \( \hat{x} = \sum_{s \in E} s\hat{\lambda}_s \).
- Variable branching in the compact space is constraint branching in the extended space.
- This idea makes it possible to define generic branching procedures.
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 to 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 to define generic branching procedures.

By default, we branch on variables in the compact space. By default, we branch on variables in the compact space.
Branching for RC

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

\[ P_I = \left\{ x \in \mathbb{R}^n \mid x = \sum_{s \in B} s\lambda_s, \sum_{s \in B} \lambda_s = 1, \lambda_s \geq 0 \forall s \in B \right\} \]

\[ \min_{\lambda \in \mathbb{R}^B_+} \left\{ c^T \left( \sum_{s \in B} s\lambda_s \right) \mid A'' \left( \sum_{s \in B} s\lambda_s \right) \geq b'', \sum_{s \in B} \lambda_s = 1 \right\} \]

- Closely related to volume algorithm and bundle methods
Branching for RC

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

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

$$\min_{\lambda \in \mathbb{R}_+^B} \left\{ c^T \left( \sum_{s \in B} s \lambda_s \right) \mid A'' \left( \sum_{s \in B} s \lambda_s \right) \geq b'', \sum_{s \in B} \lambda_s = 1 \right\}$$

- Closely related to volume algorithm and bundle methods
In general, Lagrangian methods do not provide a primal solution $\lambda$.

Let $B$ define the extreme points found in solving subproblems for $z_{LD}$.

Build an inner approximation using this set, then proceed as in PC.

\[
P_I = \left\{ x \in \mathbb{R}^n \right\} \quad \text{s.t.} \quad x = \sum_{s \in B} s\lambda_s, \sum_{s \in B} \lambda_s = 1, \lambda_s \geq 0 \quad \forall s \in B
\]

\[
\min_{\lambda \in \mathbb{R}_+^B} \left\{ c^\top \left( \sum_{s \in B} s\lambda_s \right) \right\} \quad \text{s.t.} \quad A'' \left( \sum_{s \in B} s\lambda_s \right) \geq b'', \sum_{s \in B} \lambda_s = 1
\]

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 \( \hat{x} = \sum_{s \in E} s\hat{\lambda}_s \) we can import any generic MILP technique 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 \( \text{OPT}(\mathcal{P}', c + r) \) for random perturbations
  - Solve \( \text{OPT}(\mathcal{P}_N) \) 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 $\hat{x} = \sum_{s \in E} s \hat{\lambda}_s$ we can import any generic MILP technique 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 $\text{OPT}(\mathcal{P}^r, c + r)$ for random perturbations
  - Solve $\text{OPT}(\mathcal{P}_N)$ 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 \( \hat{x} = \sum_{s \in E} s \hat{\lambda}_s \) we can import any generic MILP technique 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 \( \text{OPT}(P', c + r) \) for random perturbations
  - Solve \( \text{OPT}(P_N) \) 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 $\hat{x} = \sum_{s \in E} s\hat{\lambda}_s$ we can import any generic MILP technique 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 $\text{OPT}(P', c + r)$ for random perturbations
  - Solve $\text{OPT}(P_N)$ 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

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

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

```python
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',
})
```

Ralphs, Galati, Wang
Decomposition Methods for Integer Linear Programming 36/48
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.

\[
\begin{pmatrix}
A_1'' & A_2'' & \cdots & A_\kappa'' \\
A_1' & A_2' & \cdots & A_\kappa'
\end{pmatrix}
\]

- 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

MIPLIB2003 instance: p2756

nz = 8937

Detected block structure for p2756 instance
Hidden Block Structure
Hidden Block Structure

MIPLIB2003 instance: a1c1s1

nz = 839
Hidden Block Structure

Instance a1c1s1 with 10 blocks partitioning
## Bound Improvement

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>CPX11</th>
<th>DIP-PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Gap</td>
</tr>
<tr>
<td>retail27</td>
<td>T 2.30%</td>
<td>2674921</td>
</tr>
<tr>
<td>retail31</td>
<td>T 0.49%</td>
<td>1434931</td>
</tr>
<tr>
<td>retail3</td>
<td>529.77 OPT</td>
<td>2632157</td>
</tr>
<tr>
<td>retail4</td>
<td>T 1.61%</td>
<td>1606911</td>
</tr>
<tr>
<td>retail6</td>
<td>1.12 OPT</td>
<td>803</td>
</tr>
</tbody>
</table>
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
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):

<table>
<thead>
<tr>
<th>Client Problem</th>
<th>IP-GAP</th>
<th>CPX12.1</th>
<th>Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIP@SAS</td>
<td>CPX12.1</td>
<td>DIP@SAS</td>
</tr>
<tr>
<td>ATM Cash Management and Predictive Model (India)</td>
<td>OPT</td>
<td>∞</td>
<td>103</td>
</tr>
<tr>
<td>ATM Cash Management (Singapore)</td>
<td>OPT</td>
<td>OPT</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>OPT</td>
<td>90</td>
</tr>
<tr>
<td>Retail Inventory Optimization (UK)</td>
<td>1.6%</td>
<td>9%</td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td>4.7%</td>
<td>19%</td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td>2.6%</td>
<td>∞</td>
<td>1200</td>
</tr>
</tbody>
</table>
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