# A Framework for Decomposition in Integer Programming

Matthew Galati[1] Ted Ralphs[2]

[1] SAS Institute, Advanced Analytics, Operations Research R & D

[2] COR@L Lab, Department of Industrial and Systems Engineering, Lehigh University

International Symposium on Mathematical Programming 2009
Chicago, IL

# Outline

# The Decomposition Principle in Integer Programming

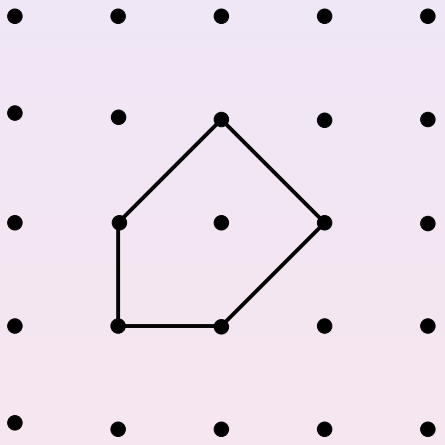
**Basic Idea:** By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

$$z_D = \min_{x \in \mathcal{P}'} \{c^\top x \mid A''x \geq b''\}$$

$$z_{IP} \geq z_D \geq z_{LP}$$



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

**Assumptions:**

- $OPT(c, \mathcal{P})$ and $SEP(x, \mathcal{P})$ are *"hard"*.
- $OPT(c, \mathcal{P}')$ and $SEP(x, \mathcal{P}')$ 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

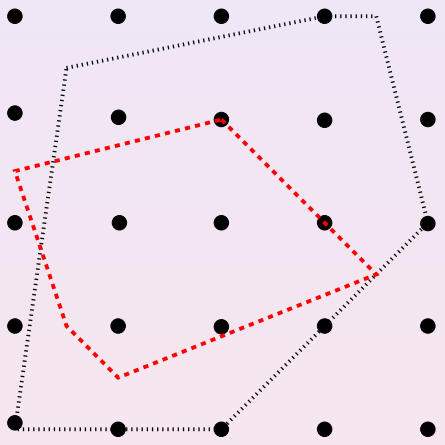
**Basic Idea:** By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

$$z_D = \min_{x \in \mathcal{P}'} \{c^\top x \mid A''x \geq b''\}$$

$$z_{IP} \geq z_D \geq z_{LP}$$



**Assumptions:**

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

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

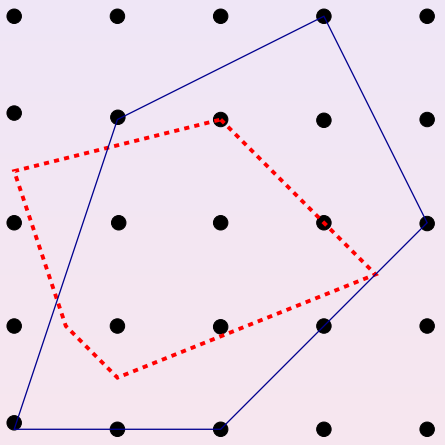
$\mathcal{Q}'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}$

# 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.

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

$$z_{IP} \geq z_D \geq z_{LP}$$



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

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

**Assumptions:**

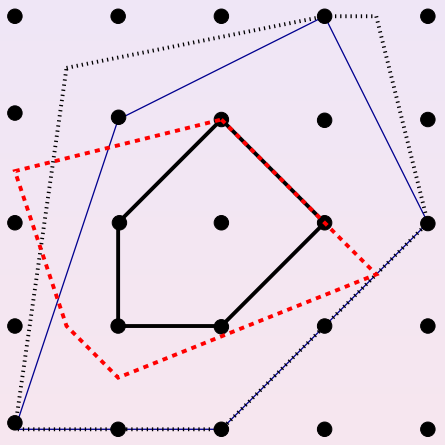
- $OPT(c, \mathcal{P})$ and $SEP(x, \mathcal{P})$ are *"hard"*.
- $OPT(c, \mathcal{P}')$ and $SEP(x, \mathcal{P}')$ are *"easy"*.
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size).
- $\mathcal{P}'$ must be represented implicitly (description has exponential size).

# 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.

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

$$z_{IP} \geq z_D \geq z_{LP}$$



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

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

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

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

**Assumptions:**

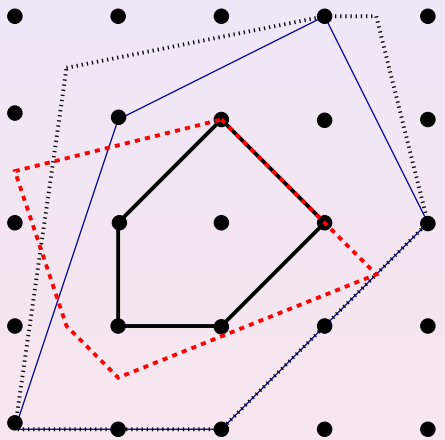
- $OPT(c, \mathcal{P})$ and $SEP(x, \mathcal{P})$ are *"hard"*.
- $OPT(c, \mathcal{P}')$ and $SEP(x, \mathcal{P}')$ are *"easy"*.
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size).
- $\mathcal{P}'$ must be represented implicitly (description has exponential size).

# 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.

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

$$z_{IP} \geq z_D \geq z_{LP}$$



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

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

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

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

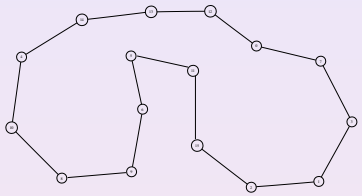
**Assumptions:**

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

# Example - Traveling Salesman Problem

**Classical Formulation**

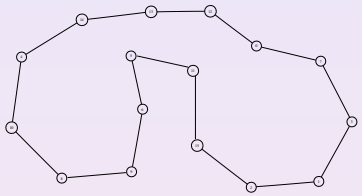
$$
\begin{array}{lcll}
x(\delta(\{u\})) & = & 2 & \forall u \in V \\
x(E(S)) & \leq & |S|-1 & \forall S \subset V,\ 3 \leq |S| \leq |V|-1 \\
x_e \in \{0,1\} & & & \forall e \in E
\end{array}
$$



# Example - Traveling Salesman Problem

**Classical Formulation**

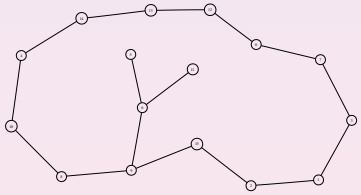
$$\begin{array}{lcll} x(\delta(\{u\})) & = & 2 & \forall u \in V \\ x(E(S)) & \leq & |S|-1 & \forall S \subset V,\ 3 \leq |S| \leq |V|-1 \\ x_e \in \{0,1\} & & & \forall e \in E \end{array}$$



**Two Relaxations**

1-Tree

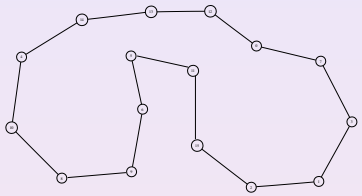
$$\begin{array}{lcll} x(\delta(\{0\})) & = & 2 & \\ x(E(V \setminus \{0\})) & = & |V|-2 & \\ x(E(S)) & \leq & |S|-1 & \forall S \subset V \setminus \{0\}, 3 \leq |S| \leq |V|-1 \\ x_e \in \{0,1\} & & & \forall e \in E \end{array}$$



# Example - Traveling Salesman Problem

**Classical Formulation**

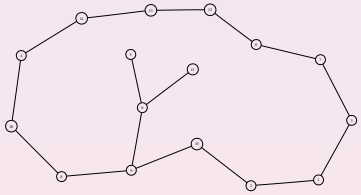
$$\begin{array}{lcll} x(\delta(\{u\})) & = & 2 & \forall u \in V \\ x(E(S)) & \leq & |S|-1 & \forall S \subset V,\ 3 \leq |S| \leq |V|-1 \\ x_e \in \{0,1\} & & & \forall e \in E \end{array}$$



**Two Relaxations**

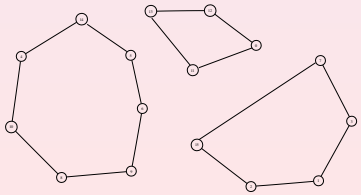
1-Tree

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



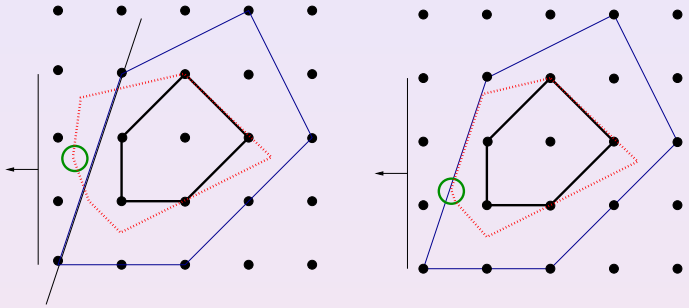
2-Matching

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



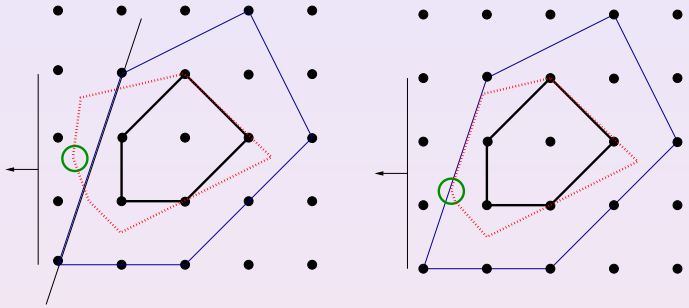
# Traditional Decomposition Methods

The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a cutting plane generation subproblem.

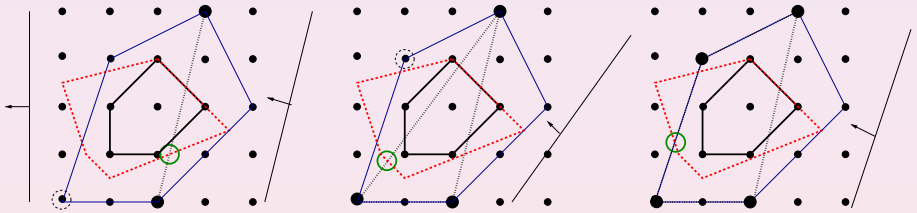


# Traditional Decomposition Methods

The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a cutting plane generation subproblem.

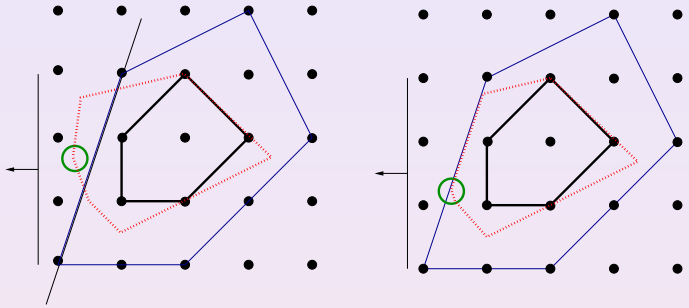


The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$ by solving a column generation subproblem.

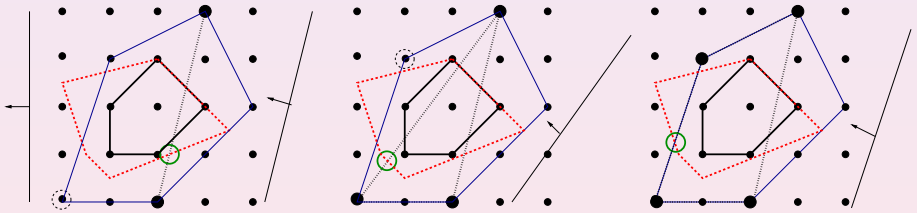


# Traditional Decomposition Methods

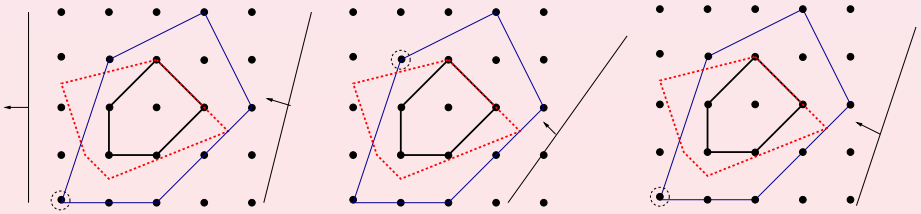
The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a cutting plane generation subproblem.



The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$ by solving a column generation subproblem.



The **Lagrangian Method (LD)** iteratively solves a Lagrangian relaxation subproblem.



# Common Threads

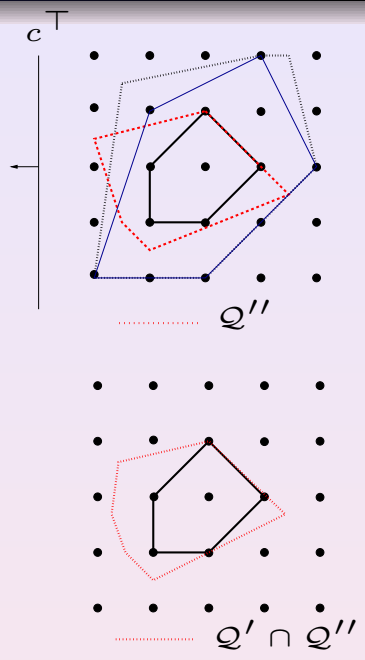
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

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

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

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

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

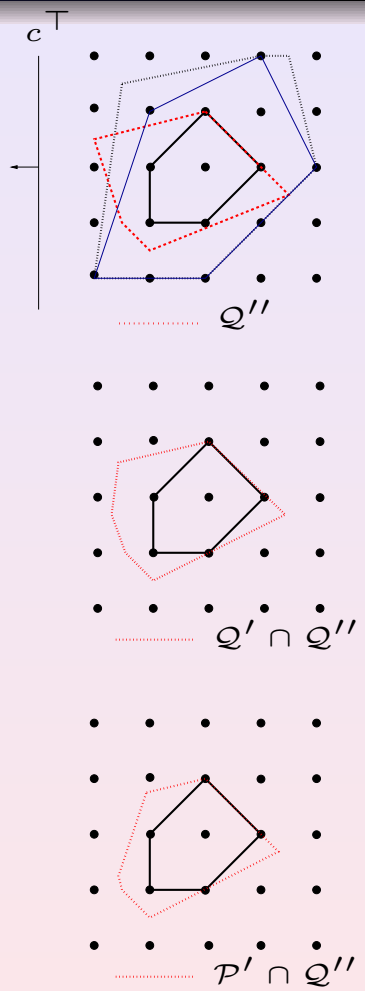
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

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

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

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

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

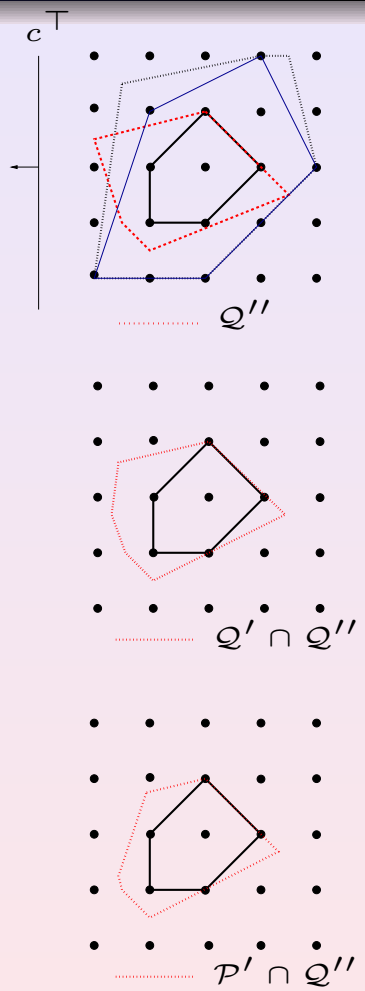
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

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

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

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

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

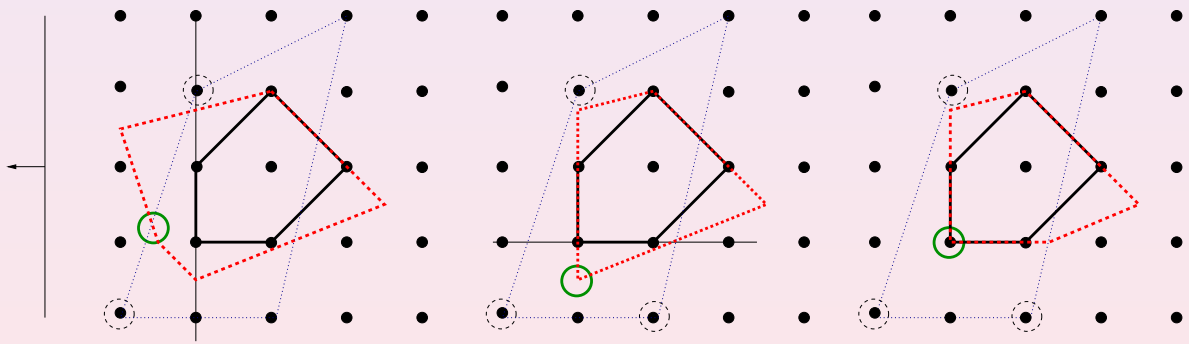


# Price and Cut

Price and Cut: Use **DW** as the bounding method. If we let $\mathcal{F}' = \mathcal{P}' \cap \mathbb{Z}^n$, then

$$z_{DW} = \min_{\lambda \in \mathbb{R}_+^{\mathcal{F}'}} \{c^\top (\sum_{s \in \mathcal{F}'} s\lambda_s) : A''(\sum_{s \in \mathcal{F}'} s\lambda_s) \geq b'', \sum_{s \in \mathcal{F}'} \lambda_s = 1\}$$

- As in the cutting plane method, separate $\hat{x} = \sum_{s \in \mathcal{F}'} s\hat{\lambda}_s$ from $\mathcal{P}$ and add cuts to $[A'', b'']$.
- **Advantage**: Cut generation takes place in the space of the compact formulation (the original space), maintaining the structure of the column generation subproblem.

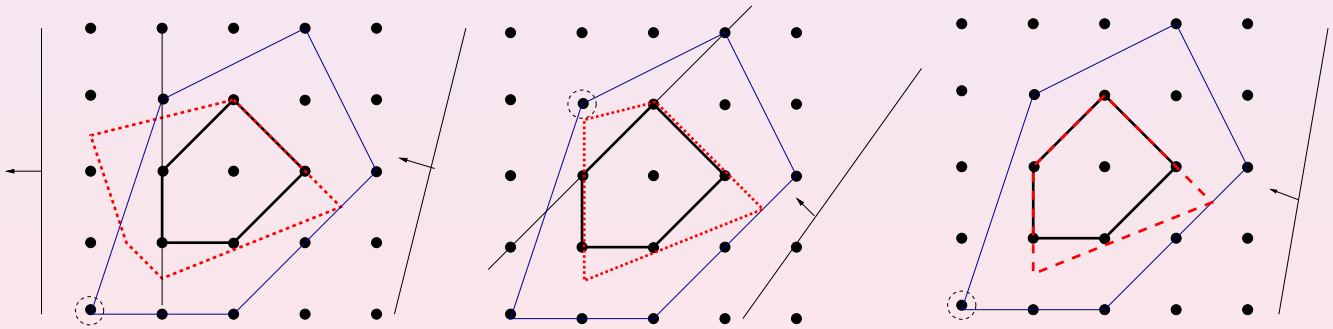


# Relax and Cut

Relax and Cut: Use **LD** as the bounding method.

$$z_{LD} = \max_{u \in \mathbb{R}^n_+} \min_{s \in \mathcal{F}'} \{(c^\top - u^\top A'')s + u^\top b''\}$$

- In each iteration, separate $\hat{s} \in \operatorname{argmin}_{s \in \mathcal{F}'} \{(c^\top - u^\top A'')s + u^\top b''\}$, a solution to the Lagrangian relaxation.
- **Advantage**: It is often much easier to separate a member of $\mathcal{F}'$ from $\mathcal{P}$ than an arbitrary real vector, such as $\hat{x}$.

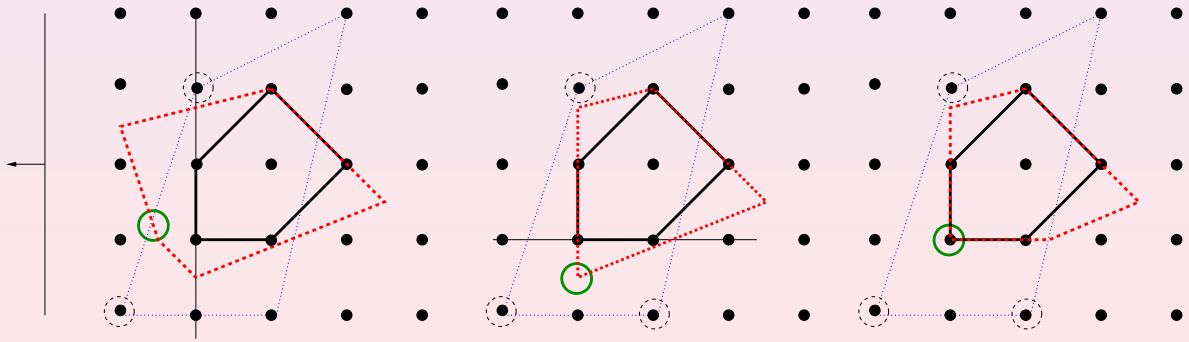


# Decompose and Cut (in CPM)

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

$$\min\{0\lambda : \sum_{s \in \mathcal{F}'} s\lambda_s = \hat{x}, \sum_{s \in \mathcal{F}'} \lambda_s = 1\}$$

- Original idea proposed by Ralphs, later used in TSP Concorde by ABCC (*Local Cuts*).
- If $\hat{x}$ lies outside $\mathcal{P}'$ the decomposition will fail.
  - Its dual ray (a *Farkas Cut*) provides a valid and violated inequality.
- The machinery for solving this already exists (=column generation).
- Often gets *lucky* and produces incumbent solutions to original IP.

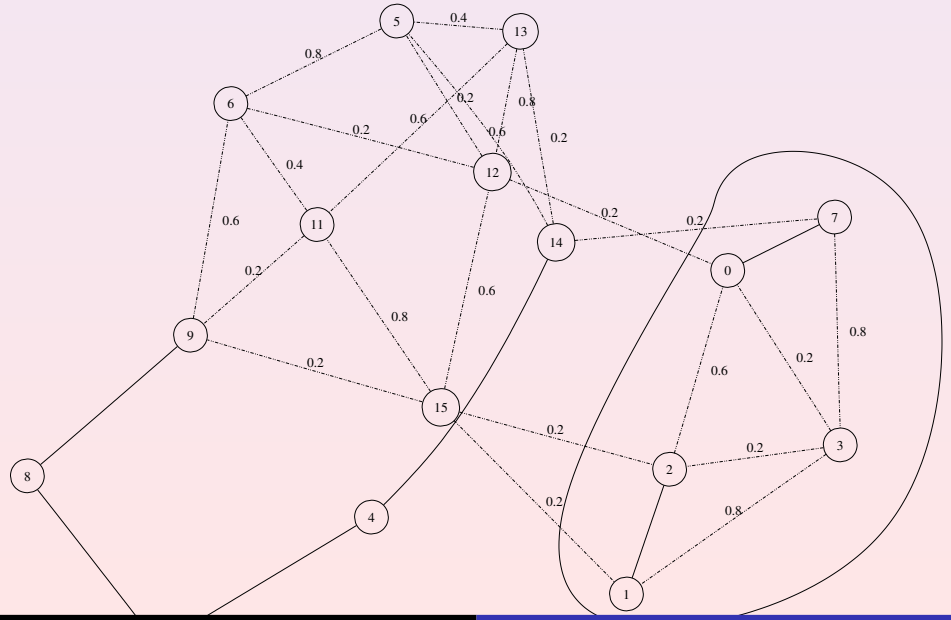


# Example - TSP

- Separation of Subtour Inequalities:

$$x(E(S)) \leq |S| - 1$$

- $SEP(x, Subtour)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^4)$ (Min-Cut)
- $SEP(s, Subtour)$, for $s$ a 2-matching, can be solved in $O(|V|)$
  - Simply determine the connected components $C_i$, and set $S = C_i$ for each component (each gives a violation of 1).

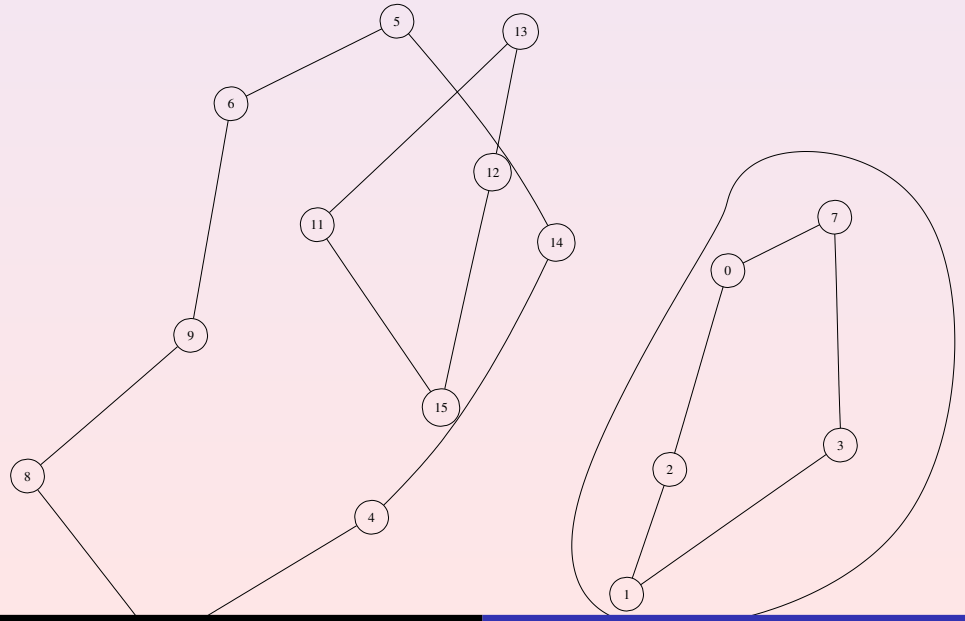


# Example - TSP

- Separation of Subtour Inequalities:

$$x(E(S)) \leq |S| - 1$$

- $SEP(x, Subtour)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^4)$ (Min-Cut)
- $SEP(s, Subtour)$, for $s$ a 2-matching, can be solved in $O(|V|)$
  - Simply determine the connected components $C_i$, and set $S = C_i$ for each component (each gives a violation of 1).

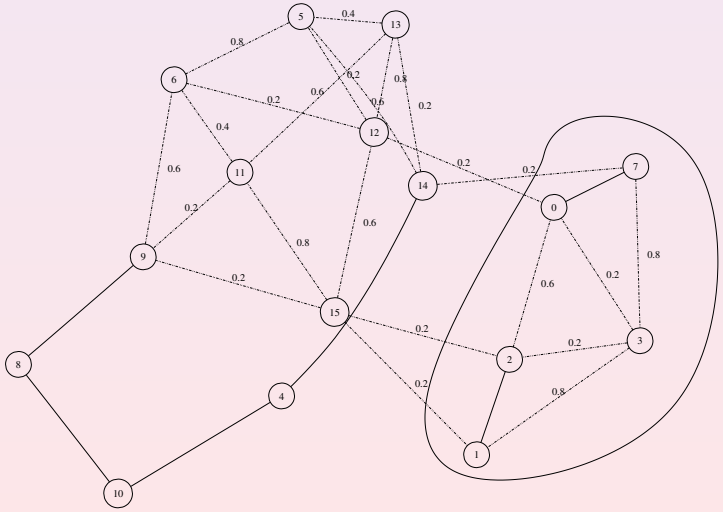


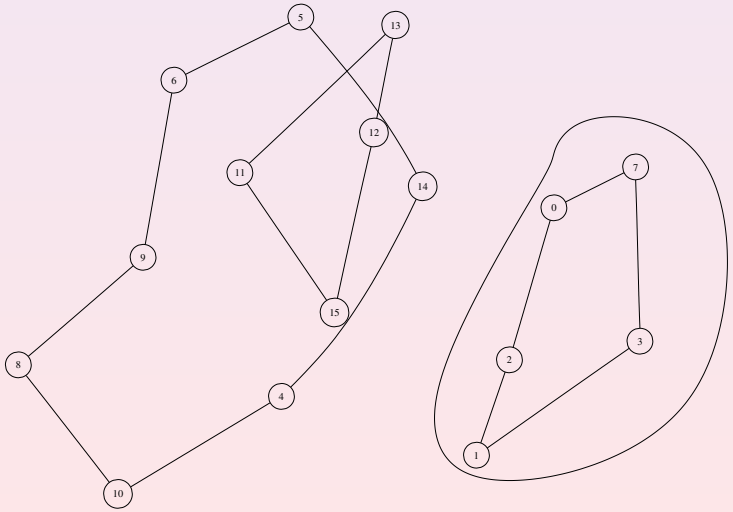
# Example - TSP

- Separation of Subtour Inequalities:

$$x(E(S)) \leq |S| - 1$$

- $SEP(x, Subtour)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^4)$ (Min-Cut)
- $SEP(s, Subtour)$, for $s$ a 2-matching, can be solved in $O(|V|)$
  - Simply determine the connected components $C_i$, and set $S = C_i$ for each component (each gives a violation of 1).



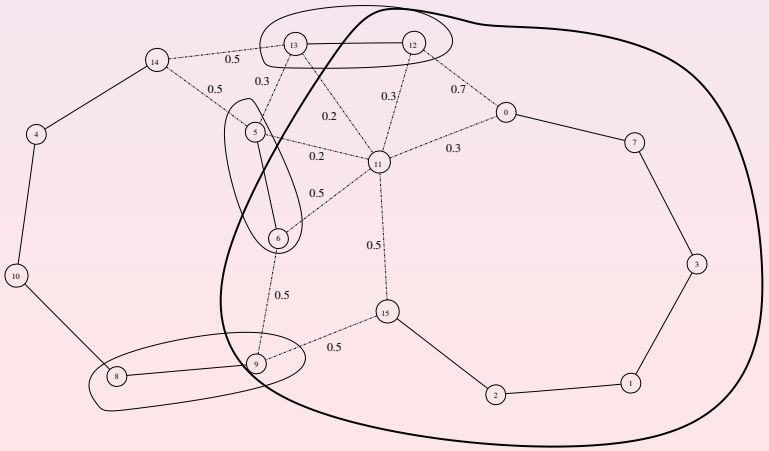


# Example - TSP

- Separation of Comb Inequalities:

$$x(E(H)) + \sum_{i=1}^{k} x(E(T_i)) \leq |H| + \sum_{i=1}^{k} (|T_i| - 1) - \lceil k/2 \rceil$$

- $SEP(x, Blossoms)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^5)$ (Padberg-Rao)
- $SEP(s, Blossoms)$, for $s$ a 1-Tree, can be solved in $O(|V|^2)$
  - Construct candidate handles $H$ from BFS tree traversal and an odd ($>= 3$) set of edges with one endpoint in $H$ and one in $V \setminus H$ as candidate teeth (each gives a violation of $\lceil k/2 \rceil - 1$).
  - This can also be used as a quick heuristic to separate 1-Trees for more general comb structures, for which there is no known polynomial algorithm for separation of arbitrary vectors.

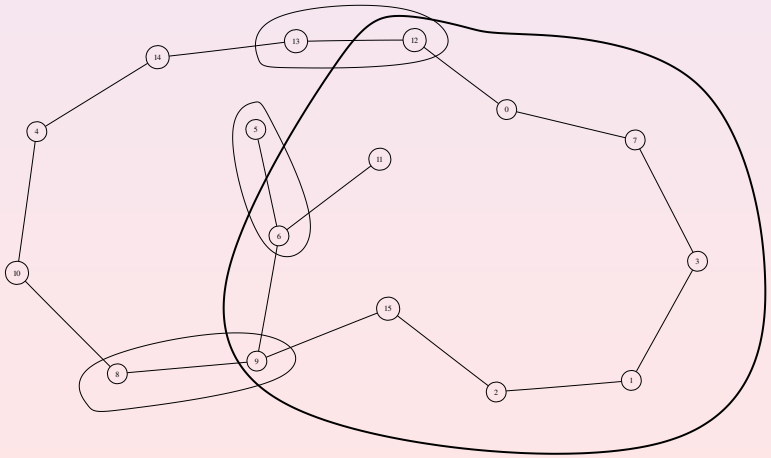


# Example - TSP

- Separation of Comb Inequalities:

$$x(E(H)) + \sum_{i=1}^{k} x(E(T_i)) \leq |H| + \sum_{i=1}^{k} (|T_i| - 1) - \lceil k/2 \rceil$$

- $SEP(x, Blossoms)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^5)$ (Padberg-Rao)
- $SEP(s, Blossoms)$, for $s$ a 1-Tree, can be solved in $O(|V|^2)$
  - Construct candidate handles $H$ from BFS tree traversal and an odd ($>= 3$) set of edges with one endpoint in $H$ and one in $V \setminus H$ as candidate teeth (each gives a violation of $\lceil k/2 \rceil - 1$).
  - This can also be used as a quick heuristic to separate 1-Trees for more general comb structures, for which there is no known polynomial algorithm for separation of arbitrary vectors.

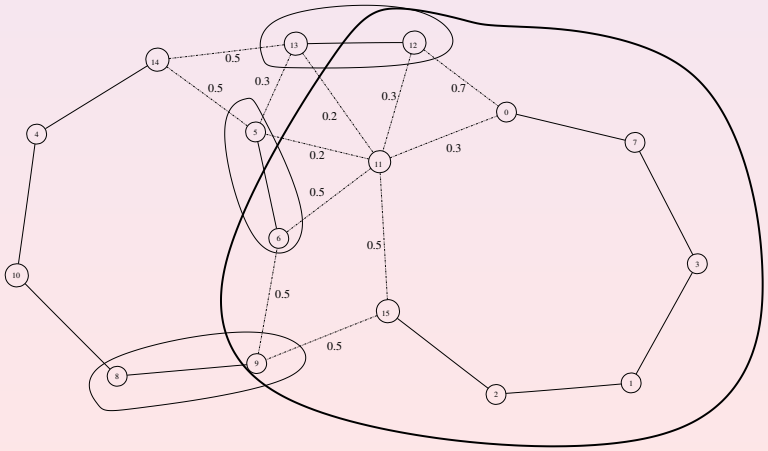


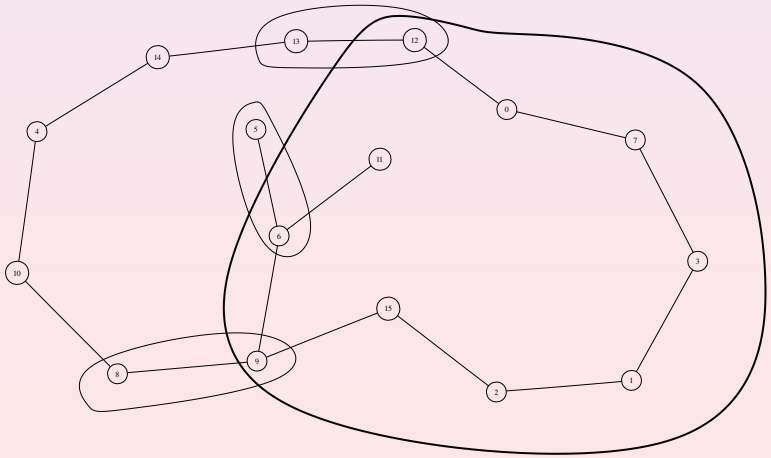
# Example - TSP

- Separation of Comb Inequalities:

$$x(E(H)) + \sum_{i=1}^{k} x(E(T_i)) \leq |H| + \sum_{i=1}^{k} (|T_i| - 1) - \lceil k/2 \rceil$$

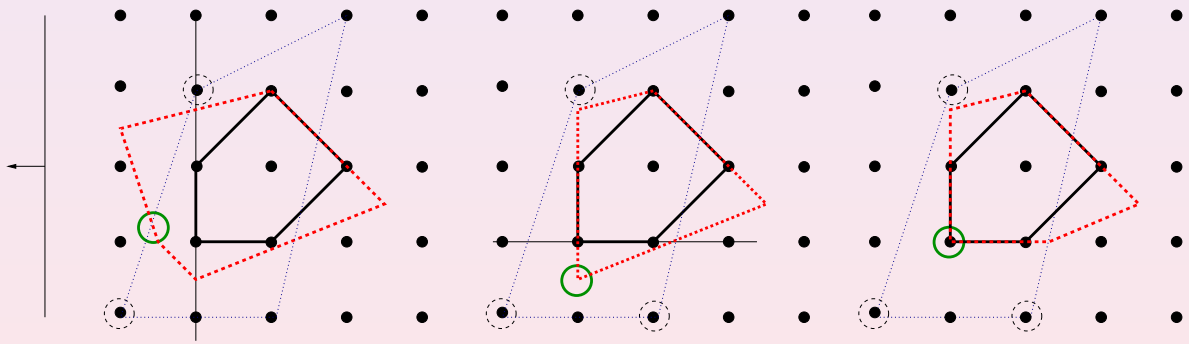
- $SEP(x, Blossoms)$, for $x \in \mathbb{R}^n$ can be solved in $O(|V|^5)$ (Padberg-Rao)
- $SEP(s, Blossoms)$, for $s$ a 1-Tree, can be solved in $O(|V|^2)$
  - Construct candidate handles $H$ from BFS tree traversal and an odd ($>= 3$) set of edges with one endpoint in $H$ and one in $V \setminus H$ as candidate teeth (each gives a violation of $\lceil k/2 \rceil - 1$).
  - This can also be used as a quick heuristic to separate 1-Trees for more general comb structures, for which there is no known polynomial algorithm for separation of arbitrary vectors.





# Decompose and Cut (in PC)

- Run CPM+DC for a few iterations using Farkas cuts to push point into $\mathcal{P}'$. Upon succesful decomposition, use this as initial seed columns.
  - Jump starts master bound $z^0_{DW} = z_{CP}$.
  - Often gets *lucky* and produces incumbent solutions to original IP.
- Rather than (or in addition to) separating $\hat{x}$, separate each member of $\{s \in \mathcal{F}' \mid \hat{\lambda}_s > 0\}$.
- As with **RC**, much easier to separate members of $\mathcal{F}'$ from $\mathcal{P}$ than $\hat{x}$.
- **RC** only gives us one member of $\mathcal{F}'$ to separate, while **PC** gives us a set, one of which must be violated by any inequality violated by $\hat{x}$.



# Branching in Price and Cut

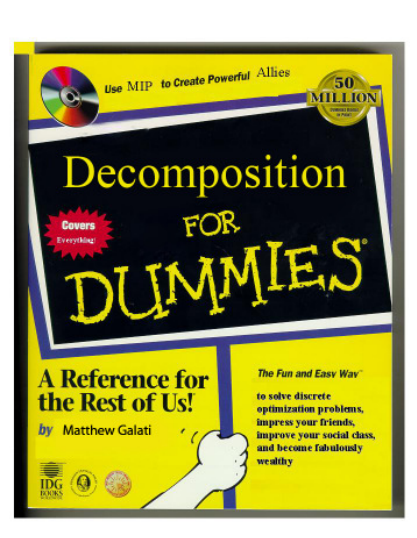
- Many complex approaches possible, but using $\hat{x} = \sum_{s \in \mathcal{F}'} s\hat{\lambda}_s$ we can simply branch on variables in the original space. $x$.
- This is equivalent to branching on cuts in the reformulated space. Simply add the original column bounds into $[A'', b'']$.
- This simple idea takes care of (most) of the design issues related to branching including dichotomty and dual updates in pricing.
- Current Limitation: Identical subproblems are currently treated like non-identical (bad for symmetry).
  - ThA08: *Review and Classification of Branching Schemes for Branch-and-price* by Francois Vanderbeck

# DECOMP Framework: Motivation

## DECOMP Framework

**DECOMP** is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It's very difficult to compare the variants discussed here in a controlled way.
- The method for separation/optimization over $\mathcal{P}'$ is the primary application-dependent component of any of these algorithms.
- DECOMP abstracts the common, generic elements of these methods.
  - Key: The user defines application-specific components in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants described here.

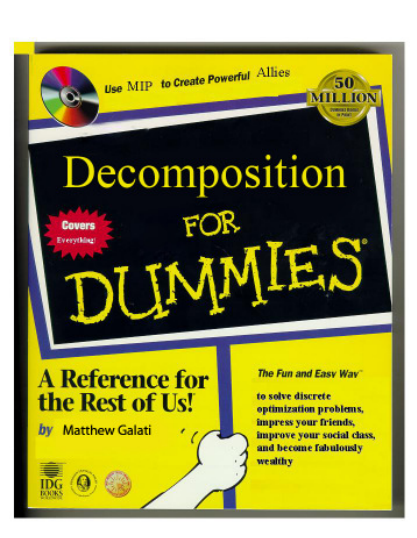


# DECOMP Framework: Motivation

## DECOMP Framework

**DECOMP** is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It's very difficult to compare the variants discussed here in a controlled way.
- The method for separation/optimization over $\mathcal{P}'$ is the primary application-dependent component of any of these algorithms.
- DECOMP abstracts the common, generic elements of these methods.
  - Key: The user defines application-specific components in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants described here.

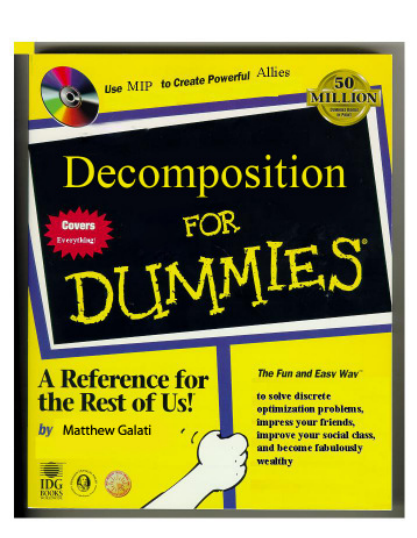


# DECOMP Framework: Motivation

## DECOMP Framework

**DECOMP** is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It's very difficult to compare the variants discussed here in a controlled way.
- The method for separation/optimization over $\mathcal{P}'$ is the primary application-dependent component of any of these algorithms.
- DECOMP abstracts the common, generic elements of these methods.
  - **Key:** The user defines application-specific components in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants described here.



# DECOMP Framework: Implementation

**COmputational INfrastructure for Operations Research**

- **DECOMP** was built around data structures and interfaces provided by COIN-OR.
- The **DECOMP** framework, written in C++, is accessed through two user interfaces:
  - Applications Interface: `DecompApp`
  - Algorithms Interface: `DecompAlgo`
- **DECOMP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel 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`

# DECOMP Framework: Implementation

**COmputational INfrastructure for Operations Research**

- **DECOMP** was built around data structures and interfaces provided by COIN-OR.
- The **DECOMP** framework, written in C++, is accessed through two user interfaces:
  - Applications Interface: `DecompApp`
  - Algorithms Interface: `DecompAlgo`
- **DECOMP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel 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`

# DECOMP Framework: Implementation

**COmputational INfrastructure for Operations Research**

- **DECOMP** was built around data structures and interfaces provided by COIN-OR.
- The **DECOMP** framework, written in C++, is accessed through two user interfaces:
  - Applications Interface: `DecompApp`
  - Algorithms Interface: `DecompAlgo`
- **DECOMP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel 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`

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP Features

- One interface to all default algorithms: **CP/DC, DW, LD, PC, RC**.
- Automatic reformulation allows users to deal with vars and cons in the original space.
- Built on top of the **OSI** interface, so easy to swap solvers (simplex to interior point).
- Can utilize **CGL** cuts in all algorithms (separate from original space).
  - Design question: What about LP-based cuts (Gomory, L&P)?
  - General design of COIN/CGL needs to be reconsidered? Should not depend on a solver.
- Column generation based on *multiple algorithms* can be easily defined and employed.
- Can derive bounds based on *multiple model/algorithm* combinations.
- Provides default (naive) branching rules in the original space.
- Active LP compression, variable and cut pool management.
- Flexible parameter interface: command line, param file, direct call overrides.
- Threaded oracle for block angular case.
  - Design issue with current COIN/OsiCpx.

# DECOMP - Applications

- The base class `DecompApp` provides an interface for the user to define the application-specific components of their algorithm.
- In order to develop an application, the user must derive the following methods/objects.

- `DecompApp::APPcreateModel()`. Define $[A'', b'']$ and $[A', b']$ (optional).
  - TSP 1-Tree: $[A'', b'']$ define the 2-matching constraints.
  - TSP 2-Match: $[A'', b'']$ define trivial subtour constraints.
- `DecompApp::isUserFeasible()`. Does $x^*$ define a feasible solution?
  - TSP: do we have a feasible tour?
- `DecompApp::APPsolveRelaxed()`. Provide a subroutine for $OPT(c, \mathcal{P}')$.
  - This is optional as well, if $[A', b']$ is defined (it will call the built in IP solver, currently CBC).
  - TSP 1-Tree: provide a solver for 1-tree.
  - TSP 2-Match: provide a solver for 2-matching.

- All other methods have appropriate defaults but are `virtual` and may be overridden.
  - `DecompApp::APPheuristics()`
  - `DecompApp::generateInitVars()`
  - `DecompApp::generateCuts()`
  - ...

# DECOMP - Applications

- The base class `DecompApp` provides an interface for the user to define the application-specific components of their algorithm.
- In order to develop an application, the user must derive the following methods/objects.

- `DecompApp::APPcreateModel()`. Define $[A'', b'']$ and $[A', b']$ (optional).
  - TSP 1-Tree: $[A'', b'']$ define the 2-matching constraints.
  - TSP 2-Match: $[A'', b'']$ define trivial subtour constraints.
- `DecompApp::isUserFeasible()`. Does $x^*$ define a feasible solution?
  - TSP: do we have a feasible tour?
- `DecompApp::APPsolveRelaxed()`. Provide a subroutine for $OPT(c, \mathcal{P}')$.
  - This is optional as well, if $[A', b']$ is defined (it will call the built in IP solver, currently CBC).
  - TSP 1-Tree: provide a solver for 1-tree.
  - TSP 2-Match: provide a solver for 2-matching.

- All other methods have appropriate defaults but are `virtual` and may be overridden.
  - `DecompApp::APPheuristics()`
  - `DecompApp::generateInitVars()`
  - `DecompApp::generateCuts()`
  - ...

# DECOMP - Applications

- The base class `DecompApp` provides an interface for the user to define the application-specific components of their algorithm.
- In order to develop an application, the user must derive the following methods/objects.

- `DecompApp::APPcreateModel()`. Define $[A'', b'']$ and $[A', b']$ (optional).
  - TSP 1-Tree: $[A'', b'']$ define the 2-matching constraints.
  - TSP 2-Match: $[A'', b'']$ define trivial subtour constraints.
- `DecompApp::isUserFeasible()`. Does $x^*$ define a feasible solution?
  - TSP: do we have a feasible tour?
- `DecompApp::APPsolveRelaxed()`. Provide a subroutine for $OPT(c, \mathcal{P}')$.
  - This is optional as well, if $[A', b']$ is defined (it will call the built in IP solver, currently CBC).
  - TSP 1-Tree: provide a solver for 1-tree.
  - TSP 2-Match: provide a solver for 2-matching.

- All other methods have appropriate defaults but are `virtual` and may be overridden.
  - `DecompApp::APPheuristics()`
  - `DecompApp::generateInitVars()`
  - `DecompApp::generateCuts()`
  - ...

# DECOMP - Applications

- The base class `DecompApp` provides an interface for the user to define the application-specific components of their algorithm.
- In order to develop an application, the user must derive the following methods/objects.

- `DecompApp::APPcreateModel()`. Define $[A'', b'']$ and $[A', b']$ (optional).
  - TSP 1-Tree: $[A'', b'']$ define the 2-matching constraints.
  - TSP 2-Match: $[A'', b'']$ define trivial subtour constraints.
- `DecompApp::isUserFeasible()`. Does $x^*$ define a feasible solution?
  - TSP: do we have a feasible tour?
- `DecompApp::APPsolveRelaxed()`. Provide a subroutine for $OPT(c, \mathcal{P}')$.
  - This is optional as well, if $[A', b']$ is defined (it will call the built in IP solver, currently CBC).
  - TSP 1-Tree: provide a solver for 1-tree.
  - TSP 2-Match: provide a solver for 2-matching.

- All other methods have appropriate defaults but are `virtual` and may be overridden.
  - `DecompApp::APPheuristics()`
  - `DecompApp::generateInitVars()`
  - `DecompApp::generateCuts()`
  - ...

# DECOMP - Applications

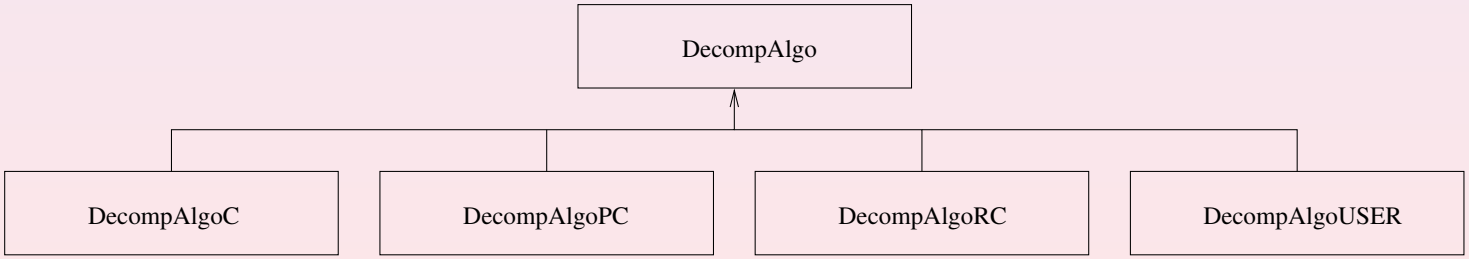
- The base class `DecompApp` provides an interface for the user to define the application-specific components of their algorithm.
- In order to develop an application, the user must derive the following methods/objects.

- `DecompApp::APPcreateModel()`. Define $[A'', b'']$ and $[A', b']$ (optional).
  - TSP 1-Tree: $[A'', b'']$ define the 2-matching constraints.
  - TSP 2-Match: $[A'', b'']$ define trivial subtour constraints.
- `DecompApp::isUserFeasible()`. Does $x^*$ define a feasible solution?
  - TSP: do we have a feasible tour?
- `DecompApp::APPsolveRelaxed()`. Provide a subroutine for $OPT(c, \mathcal{P}')$.
  - This is optional as well, if $[A', b']$ is defined (it will call the built in IP solver, currently CBC).
  - TSP 1-Tree: provide a solver for 1-tree.
  - TSP 2-Match: provide a solver for 2-matching.

- All other methods have appropriate defaults but are `virtual` and may be overridden.
  - `DecompApp::APPheuristics()`
  - `DecompApp::generateInitVars()`
  - `DecompApp::generateCuts()`
  - ...

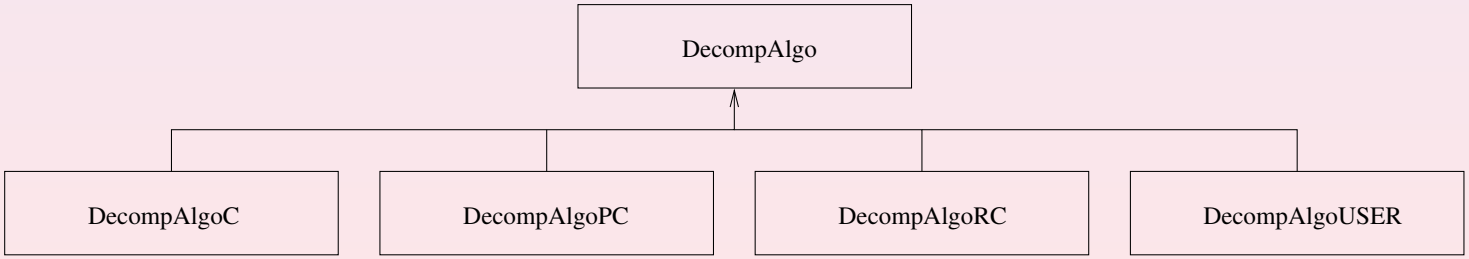
# DECOMP - Algorithms

- The base class `DecompAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described have 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, as in **bundle methods**.
  - For LD, replace subgradient with **Volume**, providing an approximate primal solution.
  - Hybrid methods like using LD to initialize the columns of the DW master.
  - During PC, adding cuts to either master and subproblem.
  - ...



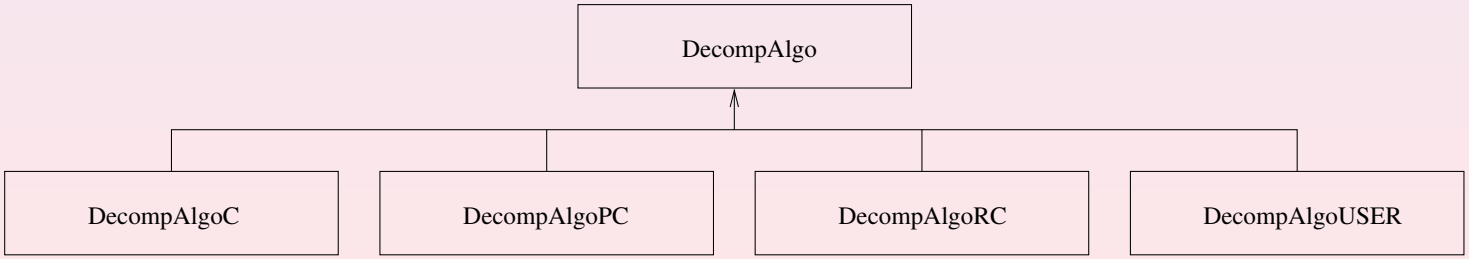
# DECOMP - Algorithms

- The base class `DecompAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described have 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, as in **bundle methods**.
  - For LD, replace subgradient with **Volume**, providing an approximate primal solution.
  - Hybrid methods like using LD to initialize the columns of the DW master.
  - During PC, adding cuts to either master and subproblem.
  - ...



# DECOMP - Algorithms

- The base class `DecompAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described have 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, as in **bundle methods**.
  - For LD, replace subgradient with **Volume**, providing an approximate primal solution.
  - Hybrid methods like using LD to initialize the columns of the DW master.
  - During PC, adding cuts to either master and subproblem.
  - ...



# DECOMP - TSP Example

## TSP_Main

```
int main(int argc, char ** argv){
   //create the utility class for parsing parameters
   UtilParameters utilParam(argc, argv);

   //create the user application (a DecompApp)
   TSP_DecompApp tsp(utilParam);
   tsp.createModel();

   //create the algorithm(s) (a DecompAlgo)
   DecompAlgoC  * cut        = new DecompAlgoC(&tsp, &utilParam);
   DecompAlgoPC * pcOneTree  = new DecompAlgoPC(&tsp, &utilParam,
                                                TSP_DecompApp::MODEL_ONETREE);
   DecompAlgoPC * pcTwoMatch = new DecompAlgoPC(&tsp, &utilParam,
                                                TSP_DecompApp::MODEL_TWOMATCH);
   DecompAlgoRC * rcOneTree  = new DecompAlgoRC(&tsp, &utilParam,
                                                TSP_DecompApp::MODEL_ONETREE);
   DecompAlgoRC * rcTwoMatch = new DecompAlgoRC(&tsp, &utilParam,
                                                TSP_DecompApp::MODEL_TWOMATCH);

   //create the driver AlpsDecomp model
   AlpsDecompModel alpsModel(utilParam);

   //install the algorithms
   //alpsModel.addDecompAlgo(cut);
   alpsModel.addDecompAlgo(pcOneTree);

   //solve
   alpsModel.solve();
}
```

# Application - ATM Cash Management Problem - MINLP Formulation

- Simple *looking* nonconvex quadratic integer NLP
  - *"it is not interesting for MINLP - it is too easy"*
- Linearize the absolute value, add binaries for count constraints.
- So far, no MINLP solvers seem to be able to solve this (several die with numerical failures).

$$
\begin{array}{llll}
\min & \sum_{a \in A, d \in D} |f_{ad}| & & \\
\text{s.t.} & c^x_{ad} x_a + c^y_{ad} y_a + c^{xy}_{ad} x_a y_a + c^u_{ad} u_a + c_{ad} & = f_{ad} & \forall a \in A, d \in D \\
& \sum_{a \in A} f_{ad} & \leq B_d & \forall d \in D \\
& |\{d \in D \mid f_{ad} < 0\}| & \leq K_a & \forall a \in A \\
& x_a, y_a & \in [0,1] & \forall a \in A \\
& u_a & \geq 0 & \forall a \in A \\
& f_{ad} & \in [0, w_{ad}] & \forall a \in A, d \in D
\end{array}
$$

# Application - ATM Cash Management Problem - MILP Approx Formulation

- Discretization of $x$ domain $\{0, 0.1, 0.2, ..., 1.0\}$.
- Linearization of product of binary and continuous, and absolute value.
- Binaries for counting constraints.

$$
\begin{aligned}
\min \quad & \sum_{a \in A, d \in D} f^+_{ad} + f^-_{ad} \\
\text{s.t.} \quad & c^x_{ad} \sum_{t \in T} c_t x_{at} + c^y_{ad} y_a + c^{xy}_{ad} \sum_{t \in T} c_t z_{at} + c^u_{ad} u_a + c_{ad} && = f^+_{ad} - f^-_{ad} && \forall a \in A, d \in D \\
& \sum_{t \in T} x_{at} && \leq 1 && \forall a \in A \\
& z_{at} && \leq x_{at} && \forall a \in A, t \in T \\
& z_{at} && \geq y_a && \forall a \in A, t \in T \\
& z_{at} && \geq x_{at} + y_a - 1 && \forall a \in A, t \in T \\
& f^-_{ad} && \leq w_{ad} v_{ad} && \forall a \in A, d \in D \\
& \sum_{a \in A} f^+_{ad} - f^-_{ad} && \leq B_d && \forall d \in D \\
& \sum_{d \in D} v_{ad} && \leq K_a && \forall a \in A
\end{aligned}
$$

# Application - ATM Cash Management Problem - MILP Approx Formulation

| | | |
|---|---|---|
| $x_{at}$ | $\in \{0,1\}$ | $\forall a \in A, t \in T$ |
| $z_{at}$ | $\in [0,1]$ | $\forall a \in A, t \in T$ |
| $v_{ad}$ | $\in \{0,1\}$ | $\forall a \in A, d \in D$ |
| $y_a$ | $\in [0,1]$ | $\forall a \in A$ |
| $u_a$ | $\geq 0$ | $\forall a \in A$ |
| $f^+_{ad}, f^-_{ad}$ | $\in [0, w_{ad}]$ | $\forall a \in A, d \in D$ |

- The MILP formulation has a natural block angular structure.
  - Master constraints are just the budget constraint.
  - Subproblem constraints (*the rest*) - one block for each ATM.

# Application - ATM Cash Management Problem - in DECOMP

- Extremely easy to define this problem in **DECOMP**.
- `DecompApp::APPcreateModel`. Just define master constraints and blocks.
  - Master constraints (budget constraints).
  - Subproblem constraints (*the rest*) - one for each ATM.
- Data setup: 648 lines of code.

```
> wc -l ATM_Instance.*
  491 ATM_Instance.cpp
  157 ATM_Instance.h
  648 total
```

- Model setup: 1221 lines of code (407 lines are comments).

```
> wc -l ATM_Decomp*.*
  951 ATM_DecompApp.cpp
  197 ATM_DecompApp.h
   73 ATM_DecompParam.h
 1221 total
> grep "//" ATM_Decomp*.* | wc -l
407
```

- Nothing else is neccessary to solve this model in **DECOMP**!

# Computational Results - ATM Cash Management Problem

Lehigh University (altair1): 8 quad-core, Intel Xeon E5410 2.33Ghz, 64-bit, 128GB, 6MB Cache

Table: One hour time limit

| $\|A\|$ | $\|D\|$ | Time | | | Gap | | |
|---|---|---|---|---|---|---|---|
| | | CPX11 | D-1 | D-8 | CPX11 | D-1 | D-8 |
| 5 | 25 | 1 | 127 | 45 | OPT | OPT | OPT |
| 5 | 50 | 3600 | 3600 | 3356 | 5.9% | 0.57% | OPT |
| 10 | 50 | 3600 | 3600 | 3600 | 0.8% | 0.07% | 0.01% |
| 10 | 100 | 3600 | 3600 | 3253 | $\infty$ | 0.67% | OPT |
| 20 | 100 | 3600 | 3423 | 1554 | $\infty$ | OPT | OPT |

Table: Time and Number of Nodes to 10% Gap

| $\|A\|$ | $\|D\|$ | TimeTo10 | | | NodeTo10 | |
|---|---|---|---|---|---|---|
| | | CPX11 | D-1 | D-8 | CPX11 | D-1 |
| 5 | 25 | 1 | 22 | 7 | 120 | 1 |
| 5 | 50 | 2,040 | 127 | 56 | 361,305 | 21 |
| 10 | 50 | 130 | 53 | 15 | 31,100 | 1 |
| 10 | 100 | no-sol | 615 | 213 | no-sol | 1 |
| 20 | 100 | no-sol | 1,156 | 416 | no-sol | 1 |

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

# Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similiar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableaux cuts.
  - Will the design of OSI and CGL work like this?
- Decomp and Cut is expensive but has many potential benefits. What is the tradeoff?
  - Generation of initial columns to start Price and Cut. Gives $z^0_{DW} = z_{CP}$.
  - If the initial $\hat{x}$ is not in $\mathcal{P}'$, Farkas cuts can move the point to the interior.
  - Along the way, we might generate incumbents for $z_{IP}$.
- Nested pricing.
  - Choose an oracle with $\mathcal{P}'$ and a restriction $\hat{\mathcal{P}}' \subset \mathcal{P}'$.
  - Price exactly (for bounds) on $\mathcal{P}'$, but generate columns heuristically on $\hat{\mathcal{P}}'$.
- *Feasibility pump* for Price and Cut.
  - Given $s \in \mathcal{F}'$, solve an auxillary MILP feasible to $\mathcal{P}'$ minimizing the $L1$ norm between $s$ and $A''$.
- For block angular case, solve the master (small model) as an IP at end of each B&B node.
  - Cheap, often produces incumbents.
- Built on top of ALPS - so parallelization of the B&B should be easy to try.

## Summary

- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DECOMP** provides an easy-to-use framework for comparing and developing various decomposition-based bounding methods.
  - The user only needs to define the components based on the compact formulation (irrespective of algorithm).
- The interface to **ALPS** allows us to investigate large-scale problems on distributed networks.
- The code is open-source, currently released under CPL and will soon (Sep or Oct '09) be available through the COIN-OR project repository **www.coin-or.org**.
- Related publications:
  - T. Ralphs and M.G., *Decomposition and Dynamic Cut Generation in Integer Programming*, Mathematical Programming 106 (2006), 261
  - T. Ralphs and M.G., *Decomposition in Integer Programming*, in Integer Programming: Theory and Practice, John Karlof, ed. (2005), 57

## Summary

- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DECOMP** provides an easy-to-use framework for comparing and developing various decomposition-based bounding methods.
  - The user only needs to define the components based on the compact formulation (irrespective of algorithm).
- The interface to **ALPS** allows us to investigate large-scale problems on distributed networks.
- The code is open-source, currently released under CPL and will soon (Sep or Oct '09) be available through the COIN-OR project repository **www.coin-or.org**.
- Related publications:
  - T. Ralphs and M.G., *Decomposition and Dynamic Cut Generation in Integer Programming*, Mathematical Programming 106 (2006), 261
  - T. Ralphs and M.G., *Decomposition in Integer Programming*, in Integer Programming: Theory and Practice, John Karlof, ed. (2005), 57

## Summary

- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DECOMP** provides an easy-to-use framework for comparing and developing various decomposition-based bounding methods.
  - The user only needs to define the components based on the compact formulation (irrespective of algorithm).
- The interface to **ALPS** allows us to investigate large-scale problems on distributed networks.
- The code is open-source, currently released under CPL and will soon (Sep or Oct '09) be available through the COIN-OR project repository **www.coin-or.org**.
- Related publications:
  - T. Ralphs and M.G., *Decomposition and Dynamic Cut Generation in Integer Programming*, Mathematical Programming 106 (2006), 261
  - T. Ralphs and M.G., *Decomposition in Integer Programming*, in Integer Programming: Theory and Practice, John Karlof, ed. (2005), 57

## Summary

- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DECOMP** provides an easy-to-use framework for comparing and developing various decomposition-based bounding methods.
  - The user only needs to define the components based on the compact formulation (irrespective of algorithm).
- The interface to **ALPS** allows us to investigate large-scale problems on distributed networks.
- The code is open-source, currently released under CPL and will soon (Sep or Oct '09) be available through the COIN-OR project repository **www.coin-or.org**.
- Related publications:
  - T. Ralphs and M.G., *Decomposition and Dynamic Cut Generation in Integer Programming*, Mathematical Programming 106 (2006), 261
  - T. Ralphs and M.G., *Decomposition in Integer Programming*, in Integer Programming: Theory and Practice, John Karlof, ed. (2005), 57

## Summary

- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DECOMP** provides an easy-to-use framework for comparing and developing various decomposition-based bounding methods.
  - The user only needs to define the components based on the compact formulation (irrespective of algorithm).
- The interface to **ALPS** allows us to investigate large-scale problems on distributed networks.
- The code is open-source, currently released under CPL and will soon (Sep or Oct '09) be available through the COIN-OR project repository **www.coin-or.org**.
- Related publications:
  - T. Ralphs and M.G., *Decomposition and Dynamic Cut Generation in Integer Programming*, Mathematical Programming 106 (2006), 261
  - T. Ralphs and M.G., *Decomposition in Integer Programming*, in Integer Programming: Theory and Practice, John Karlof, ed. (2005), 57