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

INFORMS Annual Meeting 2009
San Diego, CA

Outline

The Decomposition Principle in Integer Programming

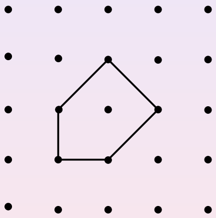
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

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

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



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

Assumptions:

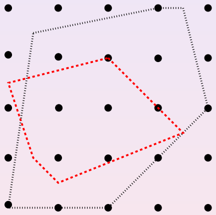
- $OPT(c, \mathcal{P})$ and $SEP(x, \mathcal{P})$ are *"hard"*
- $OPT(c, \mathcal{P}')$ and $SEP(x, \mathcal{P}')$ are *"easy"*
- $\mathcal{Q}''$ can be represented explicitly (description has polynomial size)
- $\mathcal{P}'$ must be represented implicitly (description has exponential size)

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.

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

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



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

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

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

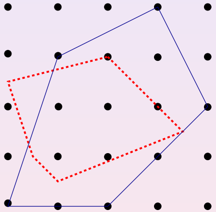
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

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

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



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

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

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

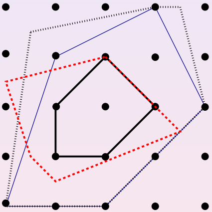
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

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

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



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

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

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

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

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

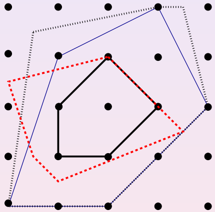
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

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

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



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

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

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

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

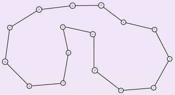
Assumptions:

- $OPT(c, \mathcal{P})$ 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

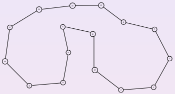
$$
\begin{array}{lcll}
x(\delta(\{u\})) & = & 2 & \forall u \in V \\
x(E(S)) & \leq & |S| - 1 & \forall S \subset V,\ 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0,1\} & & & \forall e \in E
\end{array}
$$



Example - Traveling Salesman Problem

Classical Formulation

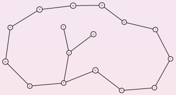
$$
\begin{array}{lcll}
x(\delta(\{u\})) & = & 2 & \forall u \in V \\
x(E(S)) & \leq & |S| - 1 & \forall S \subset V,\ 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0,1\} & & & \forall e \in E
\end{array}
$$



Two Relaxations

1-Tree

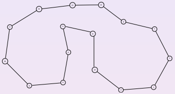
$$
\begin{array}{lcll}
x(\delta(\{0\})) & = & 2 & \\
x(E(V \setminus \{0\})) & = & |V| - 2 & \\
x(E(S)) & \leq & |S| - 1 & \forall S \subset V \setminus \{0\}, 3 \leq |S| \leq |V| - 1 \\
x_e \in \{0,1\} & & & \forall e \in E
\end{array}
$$



Example - Traveling Salesman Problem

Classical Formulation

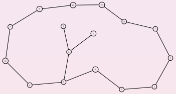
$$\begin{array}{lcll} x(\delta(\{u\})) & = & 2 & \forall u \in V \\ x(E(S)) & \leq & |S| - 1 & \forall S \subset V,\ 3 \leq |S| \leq |V| - 1 \\ x_e \in \{0,1\} & & & \forall e \in E \end{array}$$



Two Relaxations

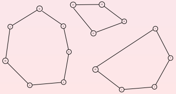
1-Tree

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



2-Matching

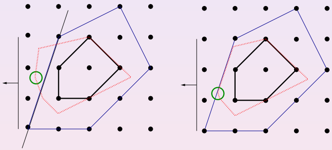
$$\begin{array}{lcll} x(\delta(u)) & = & 2 & \forall u \in V \\ x_e \in \{0,1\} & & & \forall e \in E \end{array}$$



Traditional Decomposition Methods

The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$.

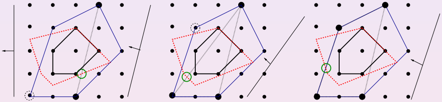
$$\min_{x \in \mathbb{R}^n} \{c^\top x \mid A'x \geq b', A''x \geq b''\}$$



Traditional Decomposition Methods

The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$.

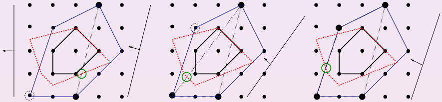
$$\min_{\lambda \in \mathbb{R}_+^{\mathcal{F}'}} \{c^\top(\textstyle\sum_{s\in\mathcal{F}'} s\lambda_s) : A''(\sum_{s\in\mathcal{F}'} s\lambda_s) \geq b'', \sum_{s\in\mathcal{F}'} \lambda_s = 1\}$$



Traditional Decomposition Methods

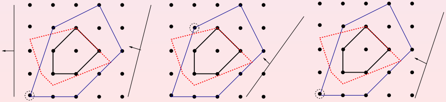
The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$.

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



The **Lagrangian Method (LD)** iteratively *traces* an *inner* approximation of $\mathcal{P}'$

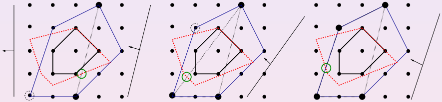
$$\max_{u \in \mathbb{R}_+^n} \min_{s \in \mathcal{F}'} \{(c^\top - u^\top A'')s + u^\top b''\}$$



Traditional Decomposition Methods

The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$.

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



Common Threads

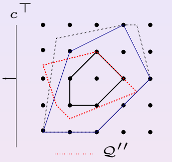
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

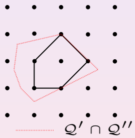
$$z_{LP} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

- 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 decomp-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

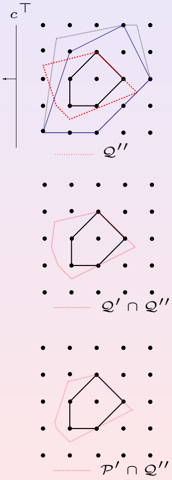
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

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

- 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 decomp-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

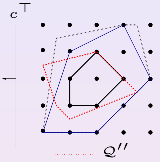
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

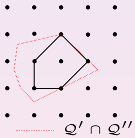
$$z_{LP} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

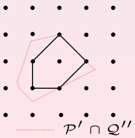
- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

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

- Traditional decomp-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)





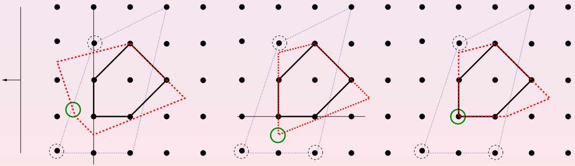


Price and Cut

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.

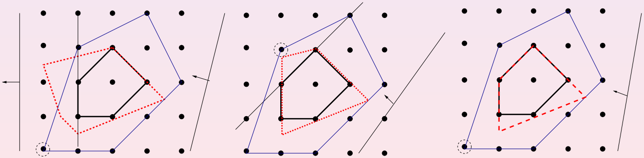


Relax and Cut

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 $\hat{x}$.

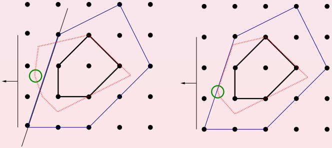


Decompose and Cut (in CPM)

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

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

- If $\hat{x}$ lies outside $\mathcal{P}'$ the decomposition will fail
  - Its dual ray (a *Farkas Cut*) provides a valid and violated inequality
  - This tells us that our cuts are *missing something* related to $\mathcal{P}'$
- Original idea proposed by Ralphs for VRP
  - Later used in TSP Concorde by ABCC (*Local Cuts*)
  - Now being used for generic MILP by Gurobi
- The machinery for solving this already exists (=column generation)
- Often gets *lucky* and produces incumbent solutions to original IP

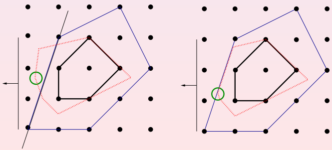


Decompose and Cut (in CPM)

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

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

- If $\hat{x}$ lies outside $\mathcal{P}'$ the decomposition will fail
  - Its dual ray (a *Farkas Cut*) provides a valid and violated inequality
  - This tells us that our cuts are *missing something* related to $\mathcal{P}'$
- Original idea proposed by Ralphs for VRP
  - Later used in TSP Concorde by ABCC (*Local Cuts*)
  - Now being used for generic MILP by Gurobi
- The machinery for solving this already exists (=column generation)
- Often gets *lucky* and produces incumbent solutions to original IP

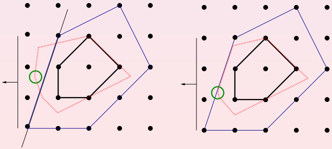


Decompose and Cut (in CPM)

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

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

- If $\hat{x}$ lies outside $\mathcal{P}'$ the decomposition will fail
  - Its dual ray (a *Farkas Cut*) provides a valid and violated inequality
  - This tells us that our cuts are *missing something* related to $\mathcal{P}'$
- Original idea proposed by Ralphs for VRP
  - Later used in TSP Concorde by ABCC (*Local Cuts*)
  - Now being used for generic MILP by **Gurobi**
- The machinery for solving this already exists (=column generation)
- Often gets *lucky* and produces incumbent solutions to original IP

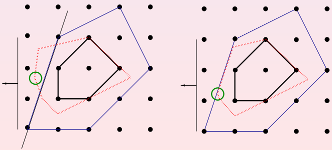


Decompose and Cut (in CPM)

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

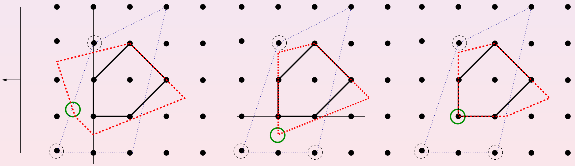
$$\min\{0\lambda : \sum_{s \in \mathcal{F}'} s\lambda_s = \hat{x}, \sum_{s \in \mathcal{F}'} \lambda_s = 1\}$$

- If $\hat{x}$ lies outside $\mathcal{P}'$ the decomposition will fail
  - Its dual ray (a *Farkas Cut*) provides a valid and violated inequality
  - This tells us that our cuts are *missing something* related to $\mathcal{P}'$
- Original idea proposed by Ralphs for VRP
  - Later used in TSP Concorde by ABCC (*Local Cuts*)
  - Now being used for generic MILP by **Gurobi**
- The machinery for solving this already exists (=column generation)
- Often gets *lucky* and produces incumbent solutions to original IP



Decompose and Cut (in PC)

- Run CPM+DC for a few iterations using Farkas cuts to push point into $\mathcal{P}'$. Upon successful 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 we can simply branch on variables in the original compact space using:

$$\hat{x} = \sum_{s \in \mathcal{F}'} s \hat{\lambda}_s$$

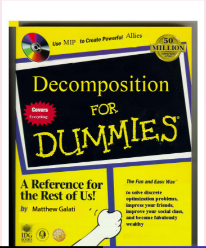
- This is equivalent to branching on cuts in the reformulated space. Simply add the original column bounds into $[A'', b'']$.
- This simple idea takes care of (most) of the design issues related to branching including dichotomy and dual updates in pricing.
- Current Limitation: Identical subproblems are currently treated like non-identical (bad for symmetry).
  - *Review and Classification of Branching Schemes for Branch-and-price* by Francois Vanderbeck
  - In some cases, we can still get around this using this framework

DIP Framework: Motivation

DIP Framework

DIP (Decomposition for Integer Programming) 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.
- DIP 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.

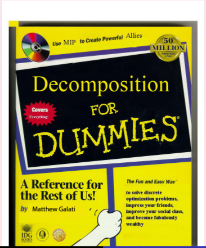


DIP Framework: Motivation

DIP Framework

DIP (Decomposition for Integer Programming) 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.
- DIP 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.

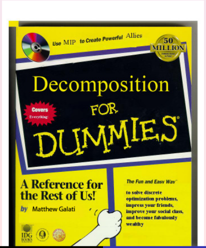


DIP Framework: Motivation

DIP Framework

DIP (Decomposition for Integer Programming) 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.
- DIP 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.



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: `DipApp`
  - Algorithms Interface: `DipAlgo`
- **DIP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel tree search.
  - `AlpsDipModel : public AlpsModel`
    - a wrapper class that calls (data access) methods from `DipApp`
  - `AlpsDipTreeNode : public AlpsTreeNode`
    - a wrapper class that calls (algorithmic) methods from `DipAlgo`

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: `DipApp`
  - Algorithms Interface: `DipAlgo`
- **DIP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel tree search.
  - `AlpsDipModel : public AlpsModel`
    - a wrapper class that calls (data access) methods from `DipApp`
  - `AlpsDipTreeNode : public AlpsTreeNode`
    - a wrapper class that calls (algorithmic) methods from `DipAlgo`

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: `DipApp`
  - Algorithms Interface: `DipAlgo`
- **DIP** provides the bounding method for branch and bound.
- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel tree search.
  - `AlpsDipModel : public AlpsModel`
    - a wrapper class that calls (data access) methods from `DipApp`
  - `AlpsDipTreeNode : public AlpsTreeNode`
    - a wrapper class that calls (algorithmic) methods from `DipAlgo`

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP 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.

DIP - Applications

- The base class `DipApp` 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.

- `DipApp::createModels()`. 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.
- `DipApp::isUserFeasible()`. Does x^* define a feasible solution?
  - TSP: do we have a feasible tour?
- `DipApp::solveRelaxed()`. 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.
  - `DipApp::heuristics()`
  - `DipApp::generateInitVars()`
  - `DipApp::generateCuts()`
  - ...

DIP - Applications

- The base class DipApp 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.

- `DipApp::createModels()`. 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.
- `DipApp::isUserFeasible()`. Does x^* define a feasible solution?
  - TSP: do we have a feasible tour?
- `DipApp::solveRelaxed()`. 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.
  - `DipApp::heuristics()`
  - `DipApp::generateInitVars()`
  - `DipApp::generateCuts()`
  - ...

DIP - Applications

- The base class DipApp 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.

- `DipApp::createModels()`. 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.
- `DipApp::isUserFeasible()`. Does x^* define a feasible solution?
  - TSP: do we have a feasible tour?
- `DipApp::solveRelaxed()`. 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.
  - `DipApp::heuristics()`
  - `DipApp::generateInitVars()`
  - `DipApp::generateCuts()`
  - ...

DIP - Applications

- The base class `DipApp` 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.

- `DipApp::createModels()`. 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.
- `DipApp::isUserFeasible()`. Does x^* define a feasible solution?
  - TSP: do we have a feasible tour?
- `DipApp::solveRelaxed()`. 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.
  - `DipApp::heuristics()`
  - `DipApp::generateInitVars()`
  - `DipApp::generateCuts()`
  - ...

DIP - Applications

- The base class `DipApp` 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.

- `DipApp::createModels()`. 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.
- `DipApp::isUserFeasible()`. Does x^* define a feasible solution?
  - TSP: do we have a feasible tour?
- `DipApp::solveRelaxed()`. 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.
  - `DipApp::heuristics()`
  - `DipApp::generateInitVars()`
  - `DipApp::generateCuts()`
  - ...

DIP Framework: Compare and Contrast to BCP

- Limitations:
  - **BCP**: The user must derive methods for almost all of the algorithmic components: (master reformulation, expansion of rows and columns, branching in reformulated space, calculation of pricing mechanisms like reduced cost, etc).
  - **DIP**: There exists a compact formulation which forms the basis of the model attributes.
- Design:
  - **BCP**: The user defines the model attributes and algorithmic components based on one pre-defined solution *method* (i.e., PC or CPM).
  - **DIP**: The user defines the model attributes and algorithmic components based on one pre-defined compact *formulation*. The different algorithmic details are managed by the framework.
- Parallelization:
  - **BCP**: Designed for shared or distributed memory for branch-and-bound search.
  - **DIP**: Threaded for block angular shared memory processing.
  - **DIP**: Built on top of Alps so potential for fully distributed branch-and-bound search (*in the future*).

DIP Framework: Compare and Contrast to BCP

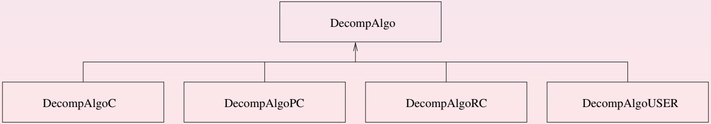
- Limitations:
  - **BCP**: The user must derive methods for almost all of the algorithmic components: (master reformulation, expansion of rows and columns, branching in reformulated space, calculation of pricing mechanisms like reduced cost, etc).
  - **DIP**: There exists a compact formulation which forms the basis of the model attributes.
- Design:
  - **BCP**: The user defines the model attributes and algorithmic components based on one pre-defined solution *method* (i.e., PC or CPM).
  - **DIP**: The user defines the model attributes and algorithmic components based on one pre-defined compact *formulation*. The different algorithmic details are managed by the framework.
- Parallelization:
  - **BCP**: Designed for shared or distributed memory for branch-and-bound search.
  - **DIP**: Threaded for block angular shared memory processing.
  - **DIP**: Built on top of Alps so potential for fully distributed branch-and-bound search (*in the future*).

DIP Framework: Compare and Contrast to BCP

- Limitations:
  - **BCP**: The user must derive methods for almost all of the algorithmic components: (master reformulation, expansion of rows and columns, branching in reformulated space, calculation of pricing mechanisms like reduced cost, etc).
  - **DIP**: There exists a compact formulation which forms the basis of the model attributes.
- Design:
  - **BCP**: The user defines the model attributes and algorithmic components based on one pre-defined solution *method* (i.e., PC or CPM).
  - **DIP**: The user defines the model attributes and algorithmic components based on one pre-defined compact *formulation*. The different algorithmic details are managed by the framework.
- Parallelization:
  - **BCP**: Designed for shared or distributed memory for branch-and-bound search.
  - **DIP**: Threaded for block angular shared memory processing.
  - **DIP**: Built on top of Alps so potential for fully distributed branch-and-bound search (*in the future*).

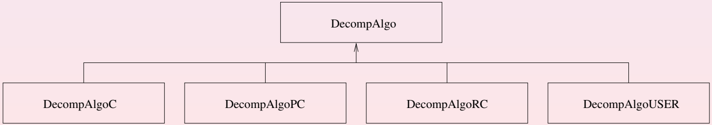
DIP - Algorithms

- The base class `DipAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described has derived default implementations `DipAlgoX : public DipAlgo` 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.
  - ...



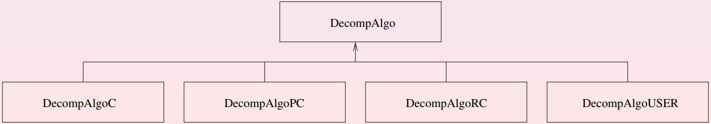
DIP - Algorithms

- The base class `DipAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described has derived default implementations `DipAlgoX : public DipAlgo` 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.
  - ...



DIP - Algorithms

- The base class `DipAlgo` provides the shell (init / master / subproblem / update).
- Each of the methods described has derived default implementations `DipAlgoX : public DipAlgo` 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.
  - ...



DIP - Applications

Table: COIN/DIP Applications

Application	Description	$\mathcal{P}'$	Cuts	Input
SmallIP	intro example, tiny IP	MILP	CGL	user
MCF	intro BCP to DIP example	NetFlow	CGL	user
MILP	random partition into A', A''	MILP	CGL	mps/lp
MILPBlock	user-defined blocks for A'	MILP(s)	CGL	mps/lp, block
AP3	3-index assignment	AP	user	user
GAP	generalized assignment	KP(s)	CGL	user
MAD	matrix decomposition	MaxClique	CGL	user
MMKP	multi-dim/choice knapsack	MCKP	CGL	user
		MDKP	CGL	user
TSP	traveling salesman problem	1Tree	Concorde	user
		2Match	Concorde	user
VRP	vehicle routing problem	mTSP	CVRPSEP	user
		kTree	CVRPSEP	user
		q-Route(s)	CVRPSEP	user
ATM	cash management (SAS COE)	MILP(s)	CGL	user

Application - ATM Cash Management Problem - Business Problem

SAS Center of Excellence in Operations Research Applications (OR COE)

- Determine schedule for allocation of cash inventory at branch banks to service ATMs
- Given historical training data per day/ATM first define polynomial fit for predicted cash flow need
  - Determine the multipliers for fit to minimize mismatch based on predicted withdrawals
- Constraints
  - Amount of cash allocated each day
  - For each ATM, limit on number of days cash flow can be less than predicted withdrawal

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).

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

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.

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

Application - ATM Cash Management Problem - MILP Approx Formulation

$$
\begin{array}{lll}
x_{at} & \in \{0,1\} & \forall a \in A, t \in T \\
z_{at} & \in [0,1] & \forall a \in A, t \in T \\
v_{ad} & \in \{0,1\} & \forall a \in A, d \in D \\
y_{a} & \in [0,1] & \forall a \in A \\
u_{a} & \geq 0 & \forall a \in A \\
f^{+}_{ad}, f^{-}_{ad} & \in [0, w_{ad}] & \forall a \in A, d \in D
\end{array}
$$

- 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 DIP

- Extremely easy to define this problem in **DIP**.
- `DipApp::createModels`. 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_Dip*.*
  951 ATM_DipApp.cpp
  197 ATM_DipApp.h
   73 ATM_DipParam.h
 1221 total
> grep "//" ATM_Dip*.* | wc -l
407
```

- Nothing else is necessary to solve this model in **DIP**!

Computational Results - ATM Cash Management Problem (5 min)

			DIP1.0			CPX11		
$\|A\|$	$\|D\|$	s	Time	Gap	Nodes	Time	Gap	Nodes
5	25	1	1.76	OPT	7	0.76	OPT	467
5	25	2	3.18	OPT	21	1.41	OPT	804
5	25	3	4.52	OPT	24	0.43	OPT	147
5	25	4	2.89	OPT	26	1.51	OPT	714
5	25	5	5.12	OPT	8	0.15	OPT	32
5	50	1	T	3.88%	331	T	∞	64081
5	50	2	T	0.20%	458	88.46	OPT	38341
5	50	3	29.40	OPT	46	8.10	OPT	3576
5	50	4	2.49	OPT	3	4.16	OPT	1317
5	50	5	T	1.08%	448	57.50	OPT	32443
10	50	1	T	0.22%	487	T	3.79%	76376
10	50	2	109.47	OPT	99	T	∞	58130
10	50	3	T	0.11%	403	T	∞	41236
10	50	4	6.03	OPT	1	T	1.92%	93891
10	50	5	7.02	OPT	3	T	0.17%	158470
10	100	1	T	1.80%	38	T	∞	13581
10	100	2	T	1.90%	101	T	∞	9486
10	100	3	T	1.57%	112	T	∞	9080
10	100	4	T	3.44%	19	T	∞	10766
10	100	5	T	1.15%	35	T	∞	11807
20	100	1	T	0.02%	7	T	∞	8786
20	100	2	T	1.12%	26	T	∞	3773
20	100	3	T	0.22%	164	T	∞	5878
20	100	4	T	0.64%	306	T	∞	7613
20	100	5	T	0.11%	538	T	∞	4775

Computational Results - ATM Cash Management Problem (1 hr)

			DIP1.0			CPX11		
$\lvert A\rvert$	$\lvert D\rvert$	s	Time	Gap	Nodes	Time	Gap	Nodes
5	25	1	1.83	OPT	7	0.76	OPT	467
5	25	2	4.49	OPT	21	1.41	OPT	804
5	25	3	6.15	OPT	24	0.42	OPT	147
5	25	4	4.25	OPT	26	1.49	OPT	714
5	25	5	5.12	OPT	8	0.16	OPT	32
5	50	1	639.69	OPT	291	T	0.10%	1264574
5	50	2	2244.28	OPT	791	87.96	OPT	38341
5	50	3	30.27	OPT	46	8.09	OPT	3576
5	50	4	2.36	OPT	3	4.13	OPT	1317
5	50	5	T	0.76%	439	57.55	OPT	32443
10	50	1	1543.85	OPT	709	T	0.76%	998624
10	50	2	107.89	OPT	99	1507.84	OPT	351879
10	50	3	T	0.11%	1496	T	0.81%	667371
10	50	4	5.82	OPT	1	1319.00	OPT	433155
10	50	5	6.61	OPT	3	365.51	OPT	181013
10	100	1	T	1.11%	87	T	∞	128155
10	100	2	T	1.43%	6017	T	∞	116522
10	100	3	T	0.95%	334	T	∞	118617
10	100	4	T	2.50%	126	T	∞	108899
10	100	5	T	1.11%	179	T	∞	167617
20	100	1	358.38	OPT	8	T	∞	93519
20	100	2	T	0.61%	126	T	∞	68863
20	100	3	T	0.18%	365	T	∞	95981
20	100	4	T	0.11%	224	T	∞	81836
20	100	5	T	0.11%	220	T	∞	101917

Application - Block Angular MILP (as a Generic Framework)

- DIP provides a black-box framework for applying Branch-Cut-And-Price to generic MILP.
  - This is the *first* framework to do this (to my knowledge).
  - Similar efforts are being talked about by F. Vanderbeck **BaPCod**.
- Currently, the only input needed is MPS/LP and a *block file*.
- Future work will attempt to embed automatic recognition of the block angular structure using packages from linear algebra like: MONET, hMETIS, Mondriaan.

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

Application - Block Angular MILP (applied to Retail Optimization)

SAS Retail Optimization Solution

- The following problem comes from SAS Retail Optimization.
- It is related to a *multi-tiered supply chain distribution problem* where each block represents a store.

Table: One hour time limit

	DIP-PC			DIP-Hyb			CPX11		
Instance	**Time**	**Gap**	**Nodes**	**Time**	**Gap**	**Nodes**	**Time**	**Gap**	**Nodes**
retail3	0.39	OPT	1	10.51	OPT	1	T	2.30%	2674921
retail27	2.88	OPT	1	12.36	OPT	1	T	0.49%	1434931
retail4	87.81	OPT	1	100.66	OPT	1	T	19.57%	991976
retail6	528.91	OPT	1866	176.35	OPT	984	T	0.01%	2632157
retail31	554.63	OPT	54	1159.46	OPT	495	T	1.61%	1606911
retail33	T	99.49%	5318	T	29.32%	329	T	0.01%	288257

Note: *retail33* LowerBound: CPX11 = 492, DIP-PC = 562

Current Computational Research for Price and Cut

- Can we implement Gomory cuts in Price and Cut?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - Generation of initial columns to start Price and Cut. Gives $z_{DW}^0 = z_{CP}$.
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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?
  - Similar to Interior Point crossover to Simplex, we can crossover from $\hat{x}$ to a feasible basis, load that into the solver and generate tableau cuts.
  - Will the design of OSI and CGL work like this? **YES.** J Forrest has added a `crossover` to OsiClp.
- Decomp and Cut is expensive but has many potential benefits. What is the trade-off?
  - 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 auxiliary MILP feasible to $\mathcal{P}'$ minimizing the L_1 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.
- DIP 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 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.
- **DIP** 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 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.
- **DIP** 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 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.
- **DIP** 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 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

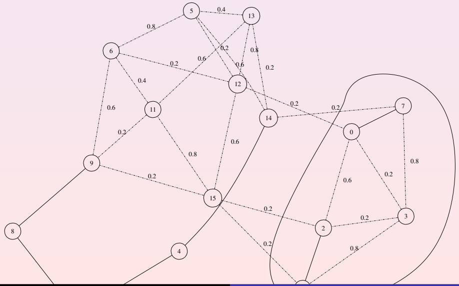
- Traditional Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}' \cap \mathcal{Q}''$.
  - $\mathcal{P}' \supset \mathcal{P}$ may have a *large* description.
- Integrated Decomposition Methods approximate $\mathcal{P}$ as $\mathcal{P}_I \cap \mathcal{P}_O$.
  - Both $\mathcal{P}_I \subset \mathcal{P}'$ and $\mathcal{P}_O \supset \mathcal{P}$ may have a *large* description.
- **DIP** 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 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

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).

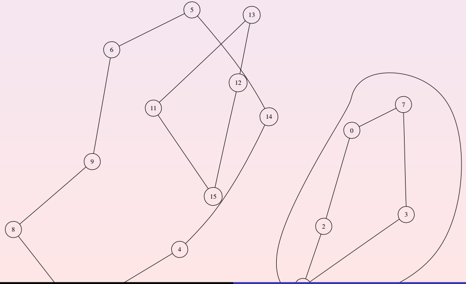


Example - TSP

- 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).

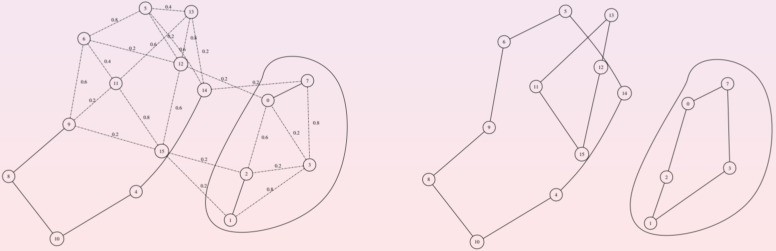


Example - TSP

- 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).

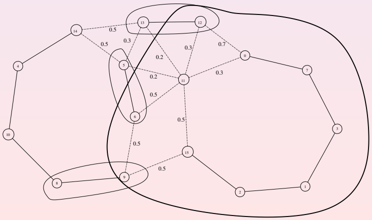


Example - TSP

- 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.

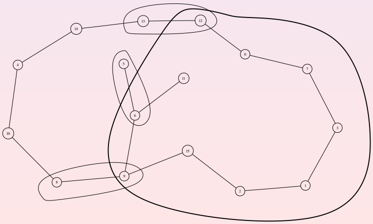


Example - TSP

- 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.