Bilevel Programming, Interdiction, and Branching for Binary Integer Programs

Andrea Lodi[1], Ted Ralphs[2], Fabrizio Rossi[3], Stefano Smriglio[3]

[1]DEIS, Universitá di Bologna
[2]COR@L Lab, Department of Industrial and Systems Engineering, Lehigh University
[3]Dipartimento di Informatica, Universitá di L'Aquila

Outline

Good Things Come in Threes

This talk concerns the relationship of three seemingly unrelated topics:

- Branching methods
- Bilevel programming
- Interdiction problems

It came together in three different cities:

- Bologna, Italy
- L'Aquila, Italy
- Bethlehem, PA, USA

And the work involved three of my Italian colleagues, who graciously hosted me during my sabbatical.[1]

[1] It should be noted that this work was fueled by the unlimited supply of excellent Italian espresso provided by my hosts.

First Theme: Branching Methods

Definition 1 *Branching is a method of partitioning of the feasible region of a mathematical program by means of a logical disjunction.*

Definition 2 *A (linear) disjunction is a logical operator consisting of a finite set of systems of inequalities that evaluates TRUE with respect to a given $\tilde{x} \in \mathbb{R}^n$ if and only if at least one of the systems is satisfied by $\tilde{x}$.*

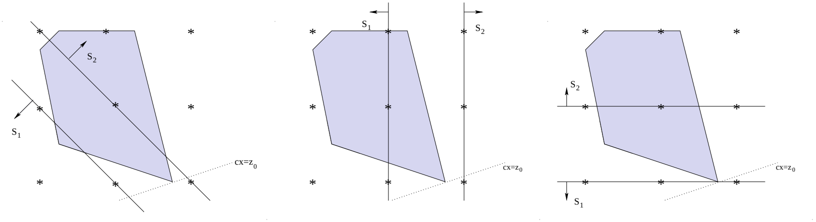
- Specifically, a disjunction is a logical operator of the form

$$\bigvee_{h \in \mathcal{Q}} A^h x \geq b^h, \; x \in \mathcal{S} \tag{1}$$

where $A^h \in \mathbb{Q}^{m_h \times n}$, $b^h \in \mathbb{Q}^{m_h}$, $n \in \mathbb{N}$, $m_h \in \mathbb{N}$, $h \in \mathcal{Q}$.

- The disjunction evaluates TRUE for $\tilde{x}$ if and only if there exists $h \in \mathcal{Q}$ such that $A^h \tilde{x} \geq b^h$.

The Branching Decision



- In *branch and bound*, branching creates one new subproblem for each term in the branching disjunction.
- Each resulting subproblem is solved recursively.
- Key Question: How should we select a disjunction?
  - Typically, the set of disjunctions to be considered is limited a priori in some fashion.
  - From this limited set, one must choose the "best" disjunction by a given measure.

What is the Criteria for Choosing?

- The overall goal of any branching scheme is to reduce running time.
- As a proxy, most branching schemes try to maximize the (estimated) bound increase resulting from imposing the disjunction.
- The problem of selecting the disjunction whose imposition results in the largest bound improvement has a natural *bilevel structure*.
- This comes from the fact that the bound is computed by solving another optimization problem.
- The disjunction selection problem can sometimes be formulated as a *bilevel program*.

Second Theme: Bilevel Linear Programming

Formally, a *bilevel linear program* is described as follows.

- $x \in X \subseteq \mathbb{R}^{n_1}$ are the *upper-level variables*
- $y \in Y \subseteq \mathbb{R}^{n_2}$ are the *lower-level variables*

Bilevel Linear Program

$$\max \left\{ c^1 x + d^1 y \mid x \in \mathcal{P}_U \cap X, y \in \operatorname{argmin}\{d^2 y \mid y \in \mathcal{P}_L(x) \cap Y\} \right\}$$

The *upper-* and *lower-level feasible regions* are:

$$\mathcal{P}_U = \left\{ x \in \mathbb{R}_+ \mid A^1 x \leq b^1 \right\} \text{ and}$$
$$\mathcal{P}_L(x) = \left\{ y \in \mathbb{R}_+ \mid G^2 y \geq b^2 - A^2 x \right\}.$$

What is the Connection?

- The upper-level variables can be used to model the choice of disjunction (we'll see an example shortly).
- The lower-level problem models the bound computation after the disjunction has been imposed.
- In strong branching, we are solving this problem essentially by enumeration.
- The bilevel branching paradigm is to select the branching disjunction directly by solving a bilevel program.

Multi-variable Branching

- For certain combinatorial problems, branching on single variables can result in very unbalanced trees.
- Consider the knapsack or set-partitioning problems, for instance.
    - Fixing a variable to 1 is typically very strong.
    - Fixing a variable to zero can have little or not effect for difficult instances.
- Often, this phenomena is caused by symmetry or near-symmetry of the variables.
- Fixing a single variable to zero has no effect because there will be another (symmetric) variable to take its place.
- However, fixing a whole set of variables to zero may have an impact.
- A number of authors have proposed methods specific to certain combinatorial problems, see, e.g., Ryan and Foster (1981); Balas and Yu (1986).
- There have also been attempts to derive general methods of multi-variable branching, e.g., SOS branching.

Branching Sets

- Consider a binary integer program $\min\{cx \mid x \in \mathcal{P} \cap \mathbb{B}^n\}$, where $c \in \mathbb{Q}^n$ is the objective function and $\mathcal{P}$ is a polyhedron.
- For any set $S = \{i_1, \ldots, i_{|S|}\} \subseteq N = \{1, \ldots, n\}$, the following disjunction is valid.

$$x_{i_1} = 1 \vee x_{i_2} = 1 \vee \ldots \vee x_{i_{|S|}} = 1 \vee \sum_{i \in S)} x_i = 0 \tag{2}$$

- Let α be the target. An index set $S \subseteq N$ is a *branching set* if and only if:

$$\max_{x \in \{0,1\}^n} \{c^T x \,| x \in \mathcal{F}, x_i = 0 \text{ for all } i \in S\} \leq \alpha, \tag{3}$$

  where $\mathcal{F} \supseteq \mathcal{P} \cap \mathbb{B}^n$.
- Our goal is to select a set with the property that simultaneously fixing all of them to zero will move the bound above a given target.
- If we set the target to the current lower bound, then we can ignore the last term and strengthen the above to:

$$x_{i_1} = 1 \vee (x_{i_2} = 1 \wedge x_{i_1} = 0) \vee \ldots \vee (x_{i_{|S|}} = 1 \wedge x_{i_1} = 0 \wedge \ldots \wedge x_{i_{|S|-1}} = 0) \tag{4}$$

Example: Knapsack Problem

Let us consider the knapsack problem:

item	1	2	3	4	5	6	7
w_j	3	3	3	4	4	5	6
a_j	1	2	2	3	3	4	5

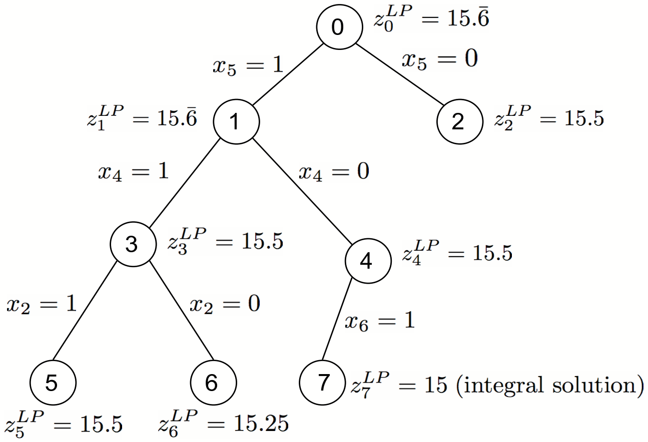
where the knapsack has size $b = 10$ and the associated IP:

$$\max \quad 3x_1 + 3x_2 + 3x_3 + 4x_4 + 4x_5 + 5x_6 + 6x_7$$

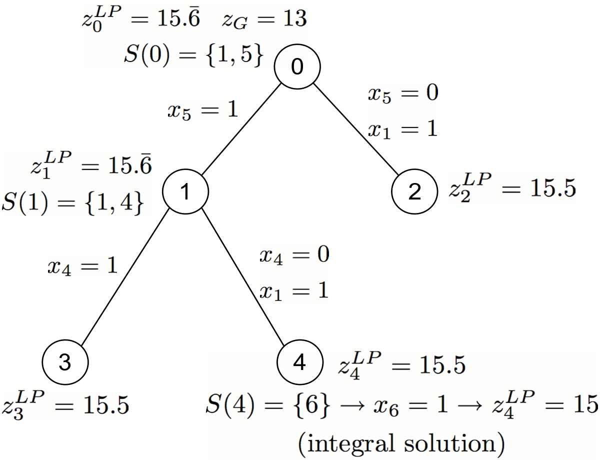
$$x_1 + 2x_2 + 2x_3 + 3x_4 + 3x_5 + 4x_6 + 5x_7 \leq 10$$

$$x_j \in \{0, 1\}, \qquad i = 1, 2, 3, 4, 5, 6, 7.$$

Example: Variable Branching



Example: Multi-variable Branching



Choosing a Branching Set

The following is a bilevel programming formulation for the problem of finding the smallest branching set.

$$
\begin{array}{ll}
\text{(BBP)} & \min \sum_{i \in N} y_i \\
\text{s.t.} & \\
& c^\top x \leq \bar{z} \\
& y \in \mathbb{B}^n \\
& x \in \arg\max_x c^\top x \\
& \text{s.t.} \\
& \qquad x_i + y_i \leq 1, \quad i \in N^a \\
& \qquad x \in \mathcal{F}
\end{array}
$$

where $\mathcal{F}$ is the feasible region of a given relaxation of the original problem used for computing the bound.

Third Theme: Interdiction Problems

- The *mixed integer interdiction problem* (MIPINT) is a bilevel program in which there is a binary upper-level *interdiction* variable for each lower-level variable.
- The interdiction variable represents the choice of which variable to remove (fix to zero) in the lower-level problem.
- The objective is to determine the set of variables whose removal has the greatest effect with respect to the upper-level objective subject to constraints.
- Often, the upper-level objective is just the negative of the lower-level objective.

Mixed Integer Interdiction

$$\max_{x \in \mathcal{P}_U^I} \min_{y \in \mathcal{P}_L^I(x)} dy \tag{MIPINT}$$

where

$$\mathcal{P}_U^I = \left\{x \in \mathbb{B}^n \mid A^1 x \leq b^1\right\}$$
$$\mathcal{P}_L^I(x) = \left\{y \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \mid G^2 y \geq b^2, y \leq u(e-x)\right\}.$$

Interdiction Branching

- Notice that the bilevel branching problem is nothing more than an interdiction problem with a slight twist.
- The twist is that we require that the lower-level objective be above the target.
- This requires allowing lower-level variables in the upper-level constraints.
- Ordinarily, this would cause problems, but because of the special form of the constraint, we can handle it.
- We can now easily state the *interdiction branching problem*.

Interdiction Branching Problem

Find the smallest interdiction set that results in an increase in the objective function value of an MILP above a certain target amount.

Solving the Interdiction Branching Problem

- The interdiction problem is a bilevel program with very special structure.
- We can solve it exactly using methods we are developing.
- Note that the exact form of the branching problem depends on the bounding subproblem (lower-level problem).
- In practice, this bound would ordinarily be an LP relaxation.
- In this case, the branching problem is a bilevel linear program with continuous variables at the lower level.
- Details of the methods for solving these problems are beyond the scope of this talk, however.

A Simple Heuristic

- Consider solving a 0-1 knapsack problem with pure branch and bound.
- In this case, we have only one fractional variable on which to branch.
- Our branching set will thus be composed of variables that are already at value one in the solution to the current relaxation.
- Idea: Build up the branching set by iteratively adding the variable with the largest reduced cost.
- Easy to implement efficiently for the knapsack problem.
- Notes
  - The current solution does not actually violate the disjunction.
  - Adding the fractional variable to the branching set ensures the disjunction will be violated.
  - When the branching set has size one and the target is the current lower bound, this means the variable can be fixed.

Variations on the Theme

- If we make the target equal to the value of the current incumbent, then we don't need to include the "all zero branch".
- Any branching set will do—we don't need the smallest one.
- We can use any upper bound on the problem to judge the effectiveness of the branching set.
- We can also use the procedure in the opposite way to fix variables to zero or even intermix variables to be fixed to zero and one.
- We can take the bounds improvement of more than one branch into account in choosing the branching set.
- Note that the bilevel branching method can apply to a much richer set of branching rules than just interdiction branching.

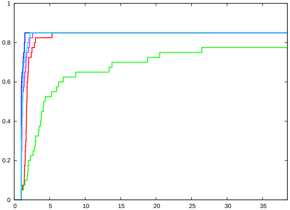
Computational Experiments: Implementation

- We coded a simple branch and bound solver for the knapsack problem using the CHiPPS tree search framework.
- Bounding is done using the Dantzig bound.
- Search order is best first.
- Note that the branching is the most computationally intensive procedure.
- Therefore, we put the node back in the queue after bounding and only branch it when it is chosen again.
- This is only possible due to the generality of CHiPPS.

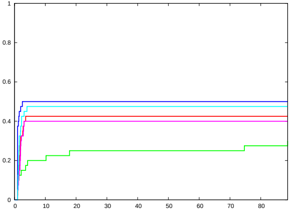
Computational Experiments: Setup

- Generated 120 difficult knapsack instances using the generator of Domenico Salvagnin.
- 20 instance each of size were 50, 60, 70, 80, 90, 100.
- Run on Linux box with Intel Xeon 2.4GHz processor and 4G memory.
- Time limit of 1800 seconds.
- Settings Tested
  - Variable branching
  - LP interdiction branching
  - LP interdiction with fractional variable added
  - IP interdiction branching with target set to 50% of gap
  - IP interdiction branching with target set to 95% of gap

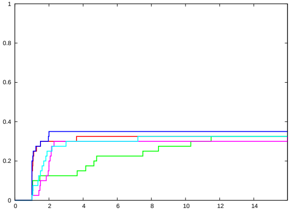
Comparing CPU Time for Fractional and Bilevel Branching



(a) Instances of size 50 and 60



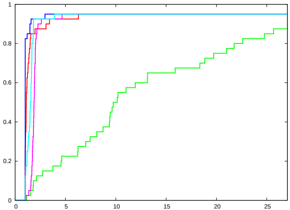
(b) Instances of size 70 and 80



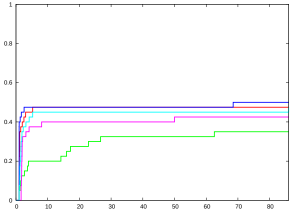
(c) Instances of size 90 and 100

Figure: Performance profile for number CPU time for knapsack instances.

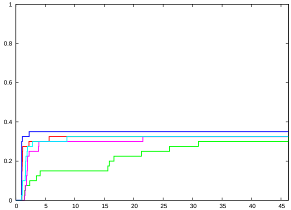
Comparing Tree Size for Fractional and Bilevel Branching



(a) Instances of size 50 and 60



(b) Instances of size 70 and 80



(c) Instances of size 90 and 100

Figure: Performance profile for tree size for knapsack instances.

Application to Other Combinatorial Problems

- In principle, the method applies to other combinatorial problems.
- However, it is not exactly clear how to generalize the methods for choosing the branching set.
- It is possible to naively apply the same method in other settings.
- Preliminary results with the TSP and VRP indicate that this does not work well.
- Our assumption that branches in which variables are fixed to one will necessarily be strong does not seem to hold.
- In most branches the bound does not seem to move.
- It seems likely we will need to take fractional variables into account in more general settings.
- We conjecture the method will work much better for problems like set-partitioning or packing problems..

Current Work: Implementation

- Interdiction branching is now an option in the MILP solver BLIS, which is a parallel solver built with in the CHiPPS framework.

COIN-OR Components Used

- The COIN High Performance Parallel Search (CHiPPS) framework to perform the branch and bound.
- The COIN LP Solver (CLP) framework for solving the LPs arising in the branch and cut.
- The Cut Generation Library (CGL) for generating cutting planes within CBC.
- The Open Solver Interface (OSI) for interfacing with CBC and CLP.

- Currently, the branching set is chosen using the simple heuristic described earlier, but this does not seem to work well.
- We are working generalizations and a more efficient implementation.

Conclusions and Future Work

- We presented a simple branching rule that works well in the case of pure branch and bound for the knapsack problem.
- It is unclear whether these performance gains can be realized in state-of-the-art solvers.
- There are connections to the *orbital branching* method of Ostrowski that need to be explored.
- If you want to play with it, you can download the solver at

  `www.coin-or.org`

References I

Balas, E. and C. Yu 1986. Finding maximum clique in an arbitrary graph. *SIAM Journal on Computing* **15**, 1054–1068.

Ryan, D. M. and B. A. Foster 1981. *Computer Scheduling of Public Transport*, chapter An integer programming approach to scheduling. North-Holland Publishing Company.