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

Setting and Notation

We consider the solution of

Binary Integer Program

$$\max_{x \in \{0,1\}^n} \{c^\top x \mid Ax \leq b\}. \qquad \text{(BIP)}$$

by branch and bound.

- A node $a = (F_1^a, F_0^a)$ of the search tree is characterized by sets
  - $F_1^a \Rightarrow$ variables fixed to one and
  - $F_0^a \Rightarrow$ variables fixed to zero.
  - $N^a = I^n \setminus (F_0^a \cup F_1^a) \Rightarrow$ free variables.
- For a node a.
  - $\mathcal{F}(a) \Rightarrow$ set of feasible solutions,
  - $\bar{z}(a) \Rightarrow$ optimal value, and
  - $z_{LP}(a) \Rightarrow$ the optimal value of the LP relaxation.
  - $\mathcal{F}(a, \bar{z}) \Rightarrow$ the set of improving solutions with respect to bound $\bar{z}$.

I apologize in advance for switching betwen min and max througout the talk!

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* $\tilde{x}$.

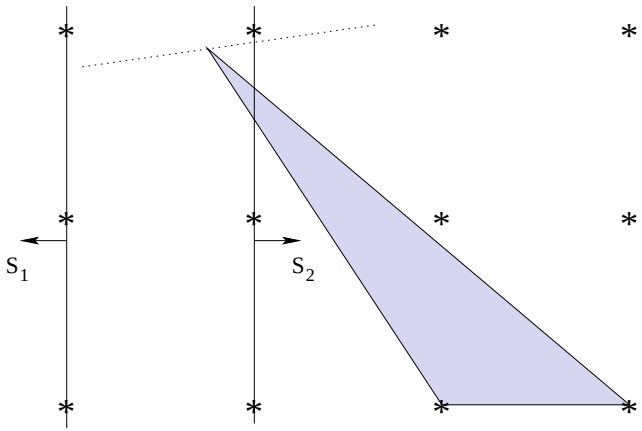
- Specifically, a disjunction is a logical operator of the form

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

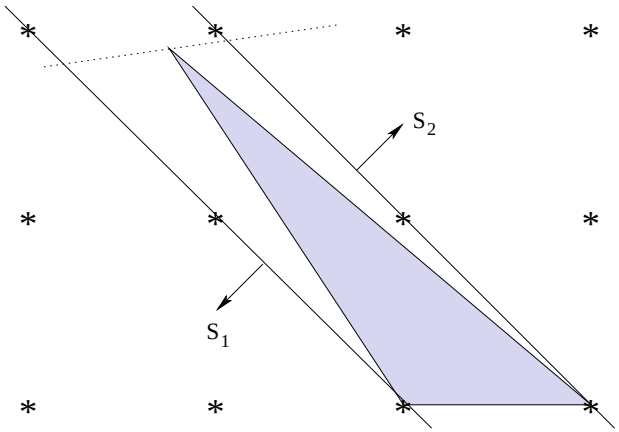
  where $A^h \in \mathbb{Q}^{m_h \times n}$, $b^h \in \mathbb{Q}^{m_h}$, $n \in \mathbb{N}$, $m_h \in \mathbb{N}$, $h \in \mathcal{Q}$.
- The disjunction evaluates TRUE for $\tilde{x}$ if and only if there exists $h \in \mathcal{Q}$ such that $A^h \tilde{x} \geq b^h$.

Branching Disjunctions

- The most common form of disjunction used for branching is $x_i \leq \pi_0 \vee x_i \geq \pi_0 + 1$ for an $i \in \{1, \ldots, d\}$.
- For example, $S_1 = \{x \in \mathbb{R}^n \mid x_1 \leq 0\}, S_2 = \{x \in \mathbb{R}^n \mid x_1 \geq 1\}$.
- This is called a Variable Disjunction.
- More general disjunctions are also possible, however.
- Most branching methods use disjunctions with only two terms.
- Here, we consider disjunctions with a potentially much larger number of terms.



A variable disjunction



A more general disjunction

Branching Sets

- 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.
- A number of authors have proposed methods for overcoming this for certain combinatorial problems, see, e.g., Ryan and Foster (1981); Balas and Yu (1986).
- We propose to branch by choosing a set $S = \{i_1, \ldots, i_{|S|}\} \subseteq N = \{1, \ldots, n\}$ using the following associated disjunction.

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

Example of Unbalanced Branching

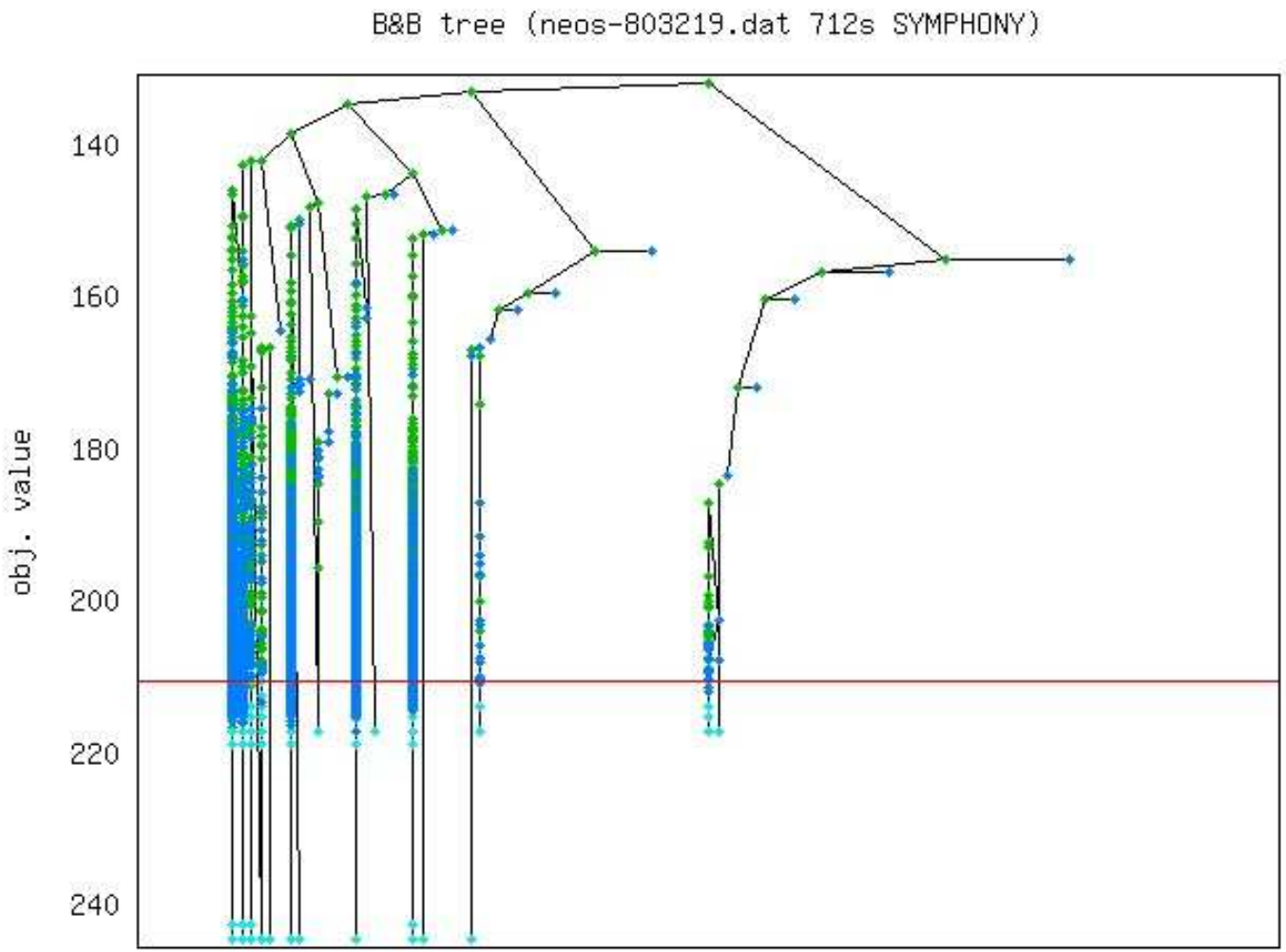


Figure: Tree for instance neos-803219

Example of Using a Branching Set

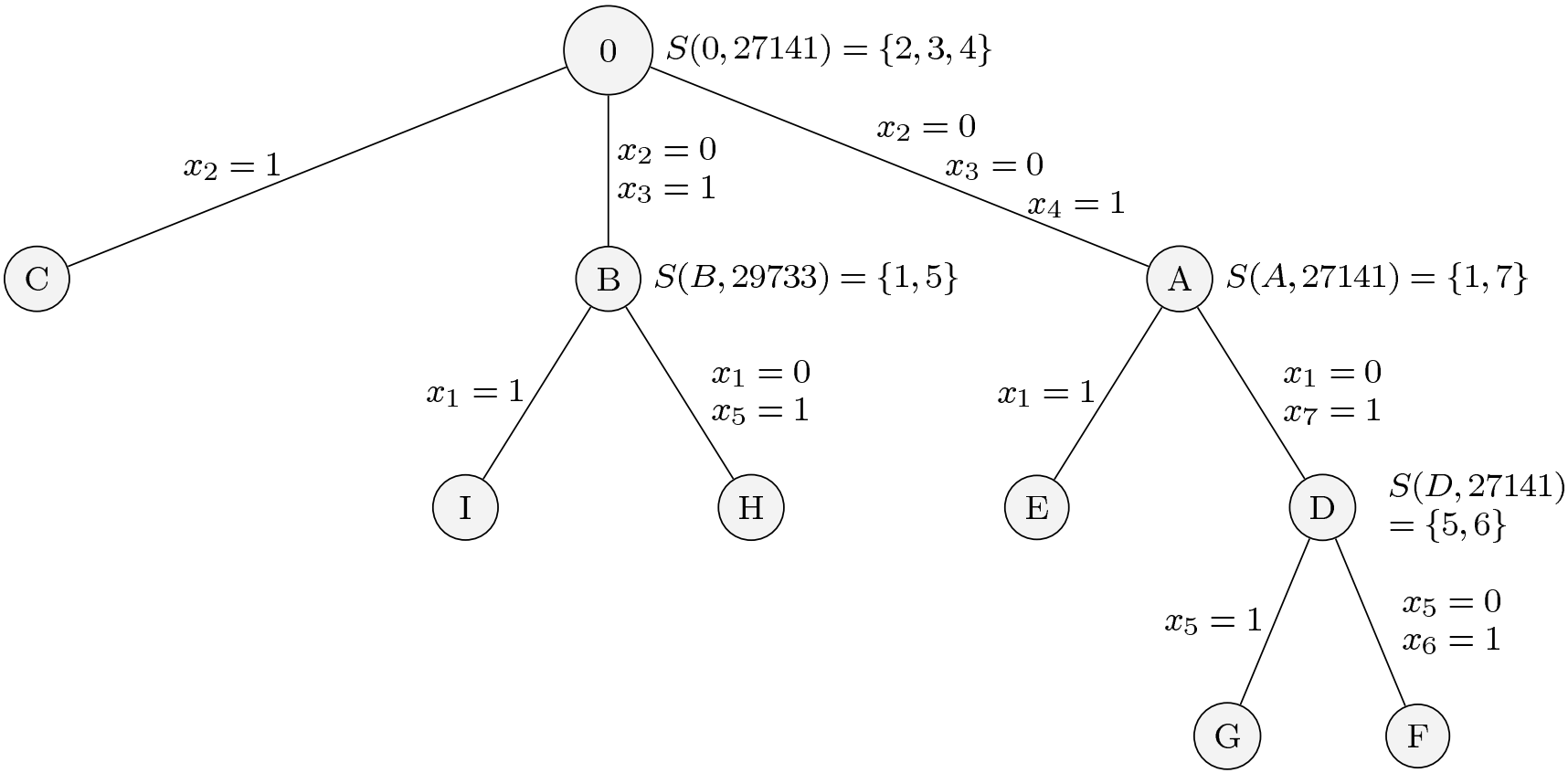


Figure: Example of Multi-variable Branching

Improving Solution Covers

- We say that a variable index $i \in I^n$ *covers* a feasible solution $\hat{x}$ if $\hat{x}_i = 1$.
- An index set S covers a solution set X if every solution in X is covered by at least one index in S.

Improving Solution Cover

An *improving solution cover* for a with respect to $\bar{z}$ is an index set $S(a, \bar{z}) = \{i_1, \ldots, i_{|S(a,\bar{z})|}\} \subseteq N^a$ covering $\mathcal{F}(a, \bar{z})$.

Theorem 1 *If $S(a, \bar{z})$ is a minimal improving solution cover, then each term of the disjunction (2) is satisfied by at least one solution in $\mathcal{F}(a, \bar{z})$, except for the right-most one, which cannot contain an improving solution.*

Corollary 1 *The number of subproblems generated when branching on minimal improving solution covers is at most $2|\mathcal{F}(a_0, \bar{z})| - 1$.*

Characterizing Improving Solution Covers

Theorem 2 *A nonempty index set $S(a, \bar{z}) \subseteq N^a$ is an improving solution cover for a with respect to $\bar{z}$ if and only if*

$$\max_{x \in \{0,1\}^n} \{c^\top x \mid x \in \mathcal{F}(a), x_i = 0 \; \forall i \in S(a, \bar{z})\} \leq \bar{z}. \tag{3}$$

- 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.
- Note that to find an improving solution cover, we need only concern ourselves with the bound in the right-most node.
- But how do we find *minimal* solution covers?

Second Theme: Bilevel Linear Programming

Formally, a *bilevel linear program* (BLP) 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:

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

Finding a Minimal Improving Solution Cover

- The following is a BLP formulation of the problem of finding the smallest branching set.

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

  where $\mathcal{F}$ is the feasible region of the original problem.

- The problem of finding an optimal disjunction for branching is often a BLP.

Third Theme: Interdiction Problems

- The *mixed integer interdiction problem* (MIPINT) is a BLP 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

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

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 target α.

Finding Branching Sets in Practice

- The interdiction branching problem is a BLP with *special structure*.
- Nevertheless, we cannot hope to find minimal improving solution covers in general (this property is *very strong*).
- We can relax the requirement $x \in \mathcal{F}$ in (BBP) by substituting a valid bound to get a more tractable subproblem.
- In standard branching, the bound ordinarily used is an LP relaxation.
- In this case, the branching problem is a BLP with continuous variables at the lower level.
- Although we can solve such a problem with standard methods, it is still difficult in practice.

A Simple Heuristic (Knapsack Problems)

- 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.

Other Ideas

In the general case, there are a number of ways in which we can obtain solution covers heuristically.

- In general, any branching set will do—we don't need the smallest one.
- In general, we can use a target $\alpha \geq \bar{z}$ (we need to explore the right-most branch in this case).
- We can use *any* upper bounding problem in the lower level of (BBP).
- Further extensions
  - We can also use the procedure w.r.t. complemented variables if that makes sense.
  - We can take the bounds improvement of more than one branch into account in choosing the branching set.
  - Note that this bilevel branching scheme can apply to a much richer set of branching rules than just interdiction branching.

Using the LP Relaxation

- The most obvious tractable relaxation of (BIP) that can be used in solving (BBP) is the LP relaxation.
- Using this relaxation in (BBP) and taking the dual of the inner optimization problem (which is an LP), we obtain this reformulation.

$$\min \sum_{i \in I^n} y_i$$

s.t.

$$b^\top u + \mathbf{1}^\top w \leq \bar{z}$$

$$c_i y_i + u^\top a_i + w_i \geq c_i \quad i \in I^n$$

$$u, w \geq 0$$

$$y \in \{0,1\}^n$$

- This is a standard MILP.

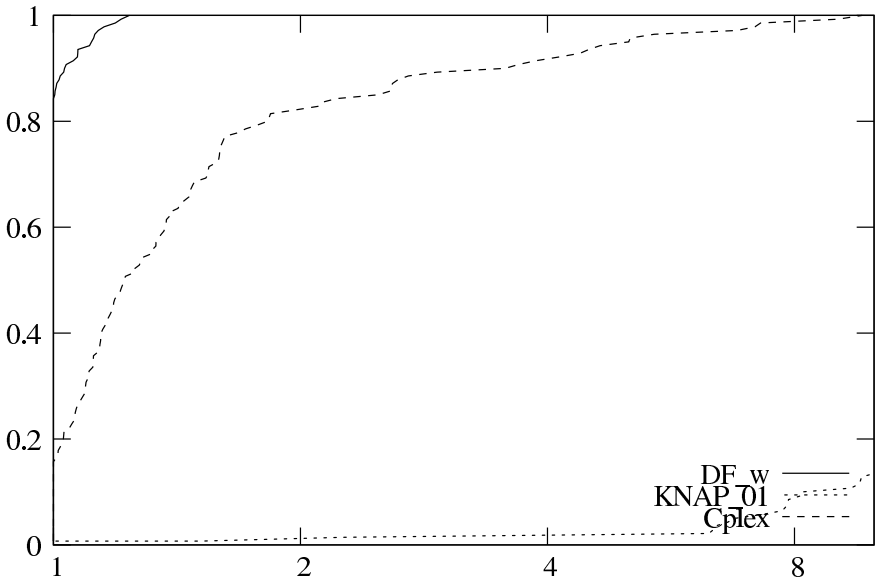
Computational Experiments: First Implementation

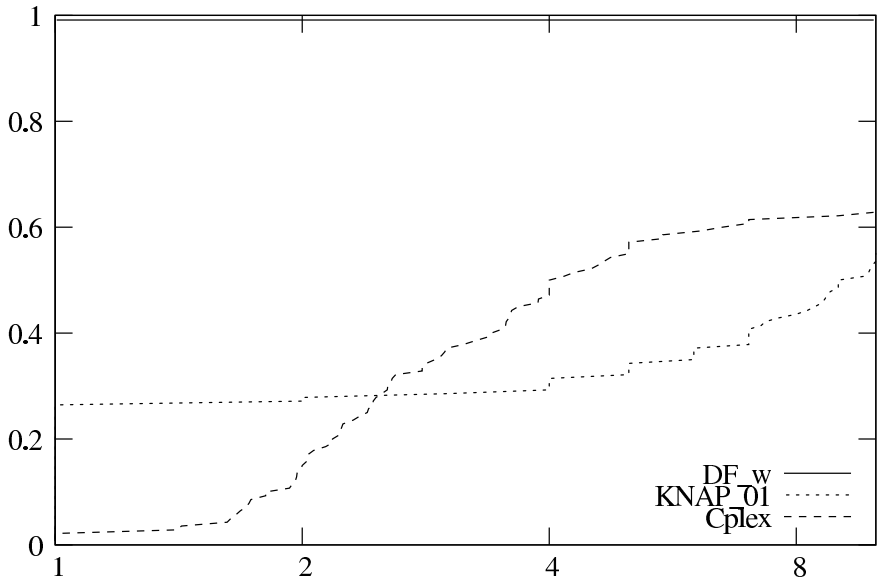
- We coded a simple branch-and-bound solver for the knapsack problem using the CHiPPS tree search framework.
  - Bounding is done using the Dantzig bound.
  - Solution cover is found using the simple heuristic.
  - Note that the branching is the most computationally intensive procedure.
  - Therefore, we put the node back in the queue after bounding and only branch on it when it is chosen again.
- Preliminary experiments showed that a depth-first search order preferring the right-most node was the most effective.

Computational Experiments: Setup

- Generated 120 difficult knapsack instances using the generator of Domenico Salvagnin.
  - SC instances: strongly correlated instances
  - SP instances: spanner set instances
  - Instance sizes were 50, 60, 70, 80, 90, 100, 110
- Run on a 2.8 GHz Intel Quad Core Linux box with 4 GB of RAM.
- A time limit of 1 hour is imposed to all experiments.
- Settings Tested
  - Interdiction branching with previously described strategy (DF_w)
  - Standard fraction branching using the CHiPPS implementation (KNAP_01)
  - CPLEX with all enhancing methodology turned off (preprcoessing, heuristics, etc.) (CPLEX)

Comparing Fractional and Interdiction Branching

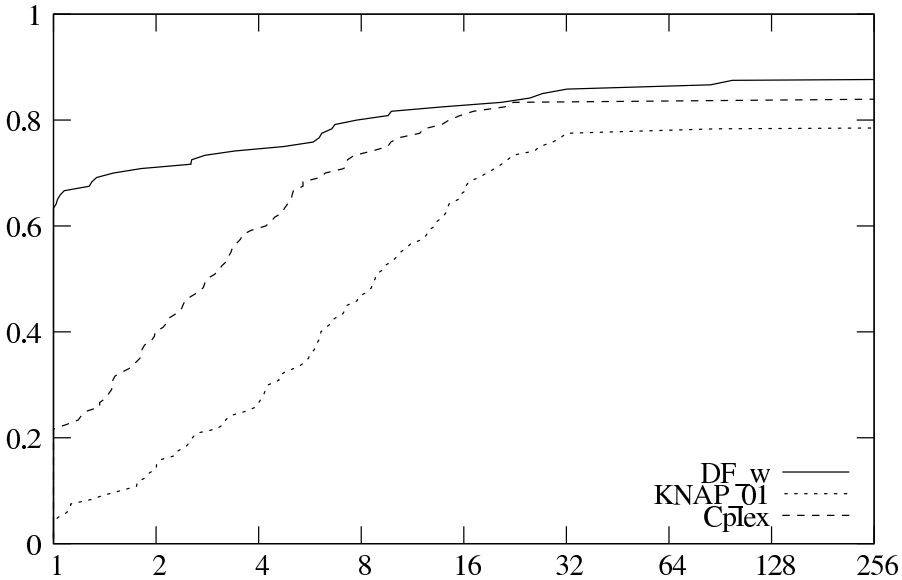


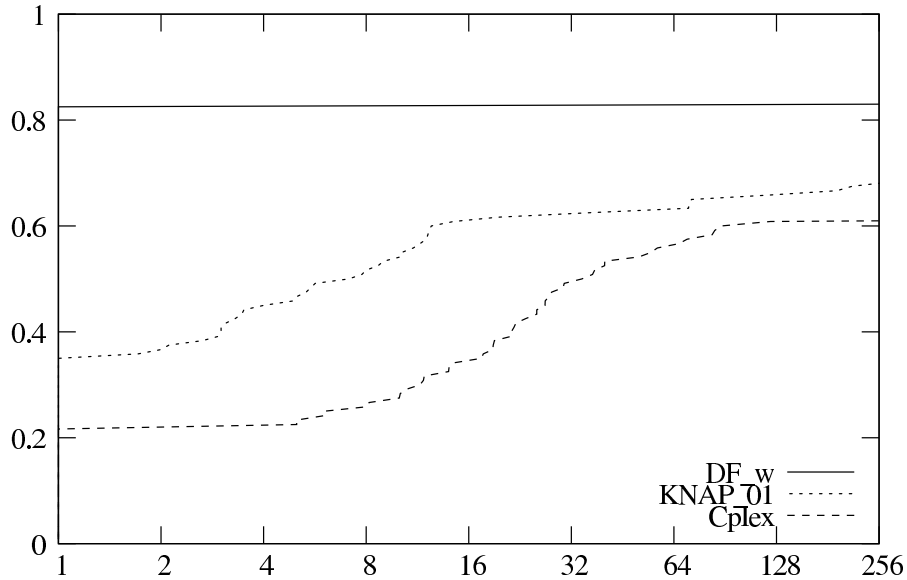


	DF_w		KNAP_01		CPLEX	
Size	# sub.	time	# sub.	time	# sub.	time
50	3640.50	0.022	100206.00	0.217	4411.30	0.121
60	14998.50	0.119	533578.45	4.039	17171.65	0.452
70	26411.10	0.248	1052241.55	29.336	27351.10	0.746
80	36449.30	0.436	1798031.40	94.355	37505.15	1.032
90	190963.25	2.626	—	—	208103.95	5.868
100	924878.25	15.251	—	—	1030983.70	30.245
110	2486483.70	47.723	—	—	2771524.75	104.482

Table: Performance for SC instances: comparison with 0-1 branching

Comparing Fractional and Interdiction Branching





	DF_w		KNAP_01		CPLEX	
Size	# sub.	time	# sub.	time	# sub.	time
50	3777.50	0.016	48608.10	0.129	21974.50	0.474
60	7218.65	0.041	92609.75	0.339	31176.05	0.731
70	444135.00	1.795	3976733.15	173.801	1770829.45	42.371
80	1821151.70	11.114	—	—	6405645.60	153.838
90	7333221.65	30.469	—	—	—	—
100	9154529.45	35.022	—	—	—	—

Table: Performance for SP instances: comparison with 0-1 branching

Computational Experiments: Second Implementation

- The multi-way branching ostensibly prevents this method from being implemented within a standard framework.
- However, we can convert a single multi-way branch into a series of one-way branches.
- This allows us to implement the whole method in the context of a standard commercial solver.
- We have done this with CPLEX 12, allowing us to do a comparison of the method with all the bells and whistles turned on.
- We had to restrict CPLEX to traditional search to use callbacks.

Comparing Tree Size for CPLEX Implementation

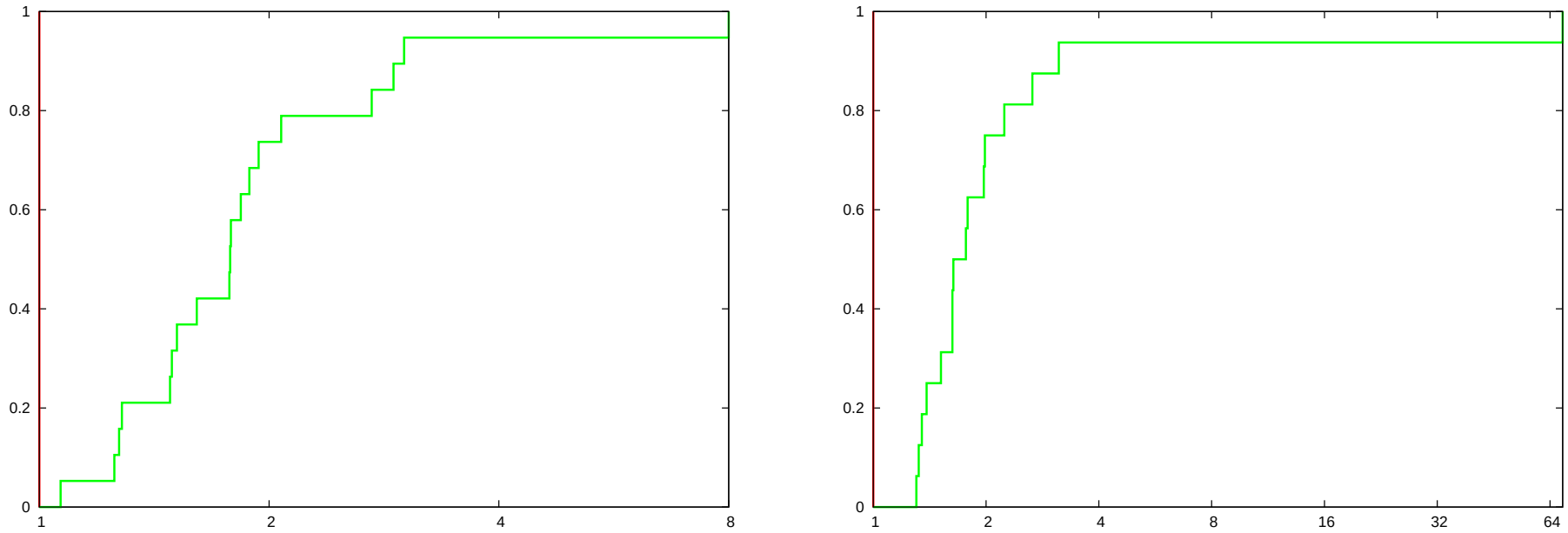


Figure: Comparing tree size for CPLEX implementation

- The performance profiles above show runs with instances of size 50 (left) and 60 (right).
- The tree is uniformly smaller with interdiction branching.
- The overhead involved in setting up the subproblem kills the running time, however.

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 on 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.