Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TOBIAS ACHTERBERG[1] STEFAN HEINZ THORSTEN KOCH

Counting solutions of integer programs using unrestricted subtree detection

[1] ILOG Deutschland, Ober-Eschbacher Str. 109, 61352 Bad Homburg, Germany

Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

ZIB-Report 08-09 (February 2008)

Counting solutions of integer programs using unrestricted subtree detection

Tobias Achterberg[1], Stefan Heinz[2]*, and Thorsten Koch[2]

[1] ILOG Deutschland, Ober-Eschbacher Str. 109, 61352 Bad Homburg, Germany
tachterberg@ilog.de
[2] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{heinz,koch}@zib.de

Abstract. In the recent years there has been tremendous progress in the development of algorithms to find optimal solutions for integer programs. In many applications it is, however, desirable (or even necessary) to generate *all* feasible solutions. Examples arise in the areas of hardware and software verification and discrete geometry.
In this paper, we investigate how to extend branch-and-cut integer programming frameworks to support the generation of all solutions. We propose a method to detect so-called *unrestricted subtrees*, which allows us to prune the integer program search tree and to collect several solutions simultaneously. We present computational results of this *branch-and-count* paradigm which show the potential of the unrestricted subtree detection.

1 Introduction

In the last decades much progress has been made in finding optimal solutions to integer linear programs (IP) [6]. Recently, more attention has been given to the task of finding all feasible solutions to a given IP, since it arises in applications, for instance, in the context of hardware and software verification and the analysis of polyhedra (see De Loera et al. [9] and references therein). Furthermore, for IP problems that evolve from industry applications, it is desirable to find multiple or even all optimal solutions as discussed in [8].

A common way to solve IP counting or enumeration problems is to transform them into an equivalent binary representation and use specialized solvers. For Boolean satisfiability instances an algorithm for counting solutions is introduced in [13]. A method based on binary decision diagrams is stated in [4]. This algorithm is capable of counting or enumerating all feasible solutions of binary linear programs (BP), which are IPs containing only binary variables. Alternative methods for these type of problems are given in [7] and [10]. Both approaches make use of a search tree. The first one additionally uses linear programming (LP) relaxations to detect infeasible subproblems.

* Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

There are only few algorithms that count or enumerate all feasible solutions of a general IP and work on the integer variable space. In [8] a branch-and-cut based algorithm is introduced which is able to generate multiple or even all (near) optimal solutions of a given IP (available in CPLEX). Setting the objective function to zero forces this algorithm to enumerate all feasible solutions. Another method which operates on the integer variable space is Barvinok's algorithm [3]. This algorithm counts all lattice points inside a convex polytope in polynomial time when the dimension is fixed.

In this paper, we introduce a *branch-and-count* method based on a branch-and-cut framework to generate all solutions of a given IP. This method works on the integer domain. Furthermore, we state a technique called *unrestricted subtree detection* which collects several solutions simultaneously.

2 Problem definition

We consider *integer programs* (IP) of the form

$$\min\{\boldsymbol{c}^T\boldsymbol{x} \mid A\,\boldsymbol{x} \leq \boldsymbol{b},\, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u},\, \boldsymbol{x} \in \mathbb{Z}^n\}$$

with $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$. Note that all variables are bounded and of integer type. We are addressing the task of computing the finite set $X_{\mathrm{IP}} = \{\boldsymbol{x} \mid A\,\boldsymbol{x} \leq \boldsymbol{b},\, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u},\, \boldsymbol{x} \in \mathbb{Z}^n\}$ of all feasible solutions of a given IP. We denote by $X^*_{\mathrm{IP}} \subseteq X_{\mathrm{IP}}$ the set of all optimal solutions of the integer program, that is, $X^*_{\mathrm{IP}} = \operatorname{argmin}\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in X_{\mathrm{IP}}\}$. If $\boldsymbol{c} = \boldsymbol{0}$, both sets are equal.

It is known that the above formulation is quite general. Maximization problems can be transformed to minimization problems by multiplying the objective function coefficients by -1. Similarly, "$\geq$" constraints can be multiplied by -1 to obtain "$\leq$" constraints. Equations can be replaced by two opposite inequalities.

In the next section, we discuss an approach to compute X_{IP}. With this method it is also possible to generate X^*_{IP}. There are two natural ways to do this: one is to first compute X_{IP} and subsequently X^*_{IP} by only keeping those elements of X_{IP} that minimize the objective function. The other possibility is to solve the underlying IP to optimality, add an additional constraint of the form $\boldsymbol{c}^T\boldsymbol{x} \leq c^*$ to the IP, with c^* being the optimal value of the IP, and finally, compute the set $X_{\mathrm{IP}'}$ for the resulting IP$'$. Obviously, $X_{\mathrm{IP}'}$ is equal to X^*_{IP}.

3 Branch-and-count approach

Currently, the most successful general technique to solve IPs (to optimality) is branch-and-cut using LP-relaxations. For a detailed description of the work-flow of branch-and-cut algorithms in general, we refer to Nemhauser and Wolsey [11].

Branch-and-cut algorithms can be adapted to enumerate all feasible solutions of a given integer program, by traversing the whole search tree and collecting all feasible solutions step-by-step. In this section we introduce a technique to speed up the enumeration process of a brach-and-cut based algorithm.

3.1 Pruning by detecting unrestricted subtrees

The basic idea of our approach is to find a way to deduce and construct *all* solution vectors contained in a subtree. If this is possible, the whole subtree can be pruned without explicitly enumerating all leaves. The two most simple structures are subtrees which have no solutions and subtrees where any variable assignment constitutes a feasible solution. We call these subtrees *infeasible subtrees* and *unrestricted subtrees*, respectively.

The infeasible subtree detection is also an issue for a standard branch-and-cut based algorithm focusing on optimal solutions. One way to improve infeasible subtree detection is *conflict analysis*, see [1, 12]. Unrestricted subtrees can be detected in the following way: at every node S in the search tree, it is checked whether each constraint is *locally redundant*, i.e., whether it is always satisfied in the local domains.

Definition. A constraint is called *locally redundant at subproblem S* if it is satisfied by all possible variable assignments of values in the local domains at subproblem S.

Lemma 1. *The subtree at a node S of the search tree is unrestricted if and only if all constraints are locally redundant at node S.*

Proof. Let x be an arbitrary vector in the local domains of subproblem S. If all constraints are locally redundant, each constraint is satisfied by x and thus, x is a feasible solution. Hence, the subtree below node S is unrestricted. On the other hand, if the subtree below S is unrestricted, x must be feasible. Therefore, it satisfies each individual constraint. It follows that each constraint is locally redundant at node S. □

A similar observation was made by Morgado et al. [10] with respect to BPs. Their search algorithm detects feasible solutions if all constraints are locally redundant (through previous variable fixings). Additionally, they have to add so-called *blocking clauses* to prevent the algorithm to count the same solutions several times. Branch-and-cut based algorithms find feasible solutions without checking each constraint for locally redundancy. Therefore, the redundancy check has to be performed explicitly in every search node to find unrestricted subtrees.

Example 1. Consider the following IP:

$$\begin{aligned} \min\{\mathbf{0}^T\boldsymbol{x} \mid\ & x_0 + x_1 + x_2 \leq 2, \\ & x_0 - x_1 + x_2 \leq 1, \\ & x_0 + x_1 - x_2 \leq 1, \\ & x_0 - x_1 - x_2 \leq 0, \\ & \boldsymbol{x} \in \{0,1\}^3\}. \end{aligned}$$

In Figure 1 we depict different branching possibilities for the root node. Only in the first case, where we branch on variable x_0, the resulting subproblems constitute an unrestricted and an infeasible subtree. More precisely, if variable x_0 is fixed to zero, all constraints are locally redundant; setting variable x_0 to one, leads to an infeasible subproblem.

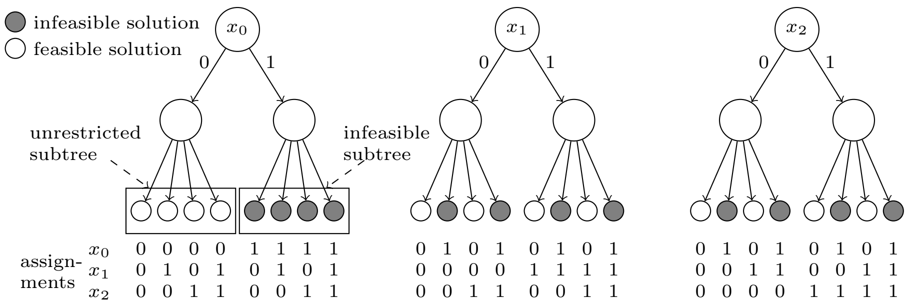

Fig. 1. Possible branching decisions in the root node for Example 1.

Table 1. Results for chip verification instances.

Instance				basic approach			unrestricted subtree detection			
Name	Cons	Vars	$\lvert X_{\mathrm{IP}}\rvert$	time	nodes	depth	time	nodes	depth	unrest.
veri1	1589	1251	809 424	12.9	1 618 847	26	0.3	17 639	19	8 448
veri2	854	691	655 360	16.8	1 310 762	30	9.6	491 567	29	245 766
veri3	219	138	573 440	18.2	1 146 948	29	15.0	860 227	29	143 360
veri4	748	623	2 097 152	33.0	4 194 319	23	4.9	327 687	20	163 840
veri5	1631	1294	260 096	4.5	520 207	22	0.7	41 011	19	20 487
veri6	1140	901	100 980	1.6	201 959	22	0.1	2 087	13	1 044
veri7	2123	1683	>68 749 M	>1800	>272 M	50	>1800	>111 M	42	>55 684 k
veri8	43	53	264 241 407	>1800	>237 M	34	77.1	4 316 909	31	2 122 366

3.2 Computational results

We integrated the unrestricted subtree detection into the branch-and-cut framework SCIP (Version 1.00.5) [2]. As an LP-solver we used SOPLEX 1.3.3 [14]. All computations presented in this section were run on computers with an Intel Core 2 Quad CPU with 2.66 GHz, 4 MB cache, and 4 GB of RAM. A time limit of 30 minutes was employed.

Due to the lack of space we first restrict our self to 8 real-world instances which contain several ten-thousand solutions each. These instances arise from chip verification problems and have been provided by OneSpin Solutions[1]. The results are given in Table 1. The first four columns contain the problem instance information, namely the name ("Name"), the number of constraints and variables ("Cons", "Vars"), and the number of feasible solutions ("$|X_{\mathrm{IP}}|$"). Columns labeled with "basic approach" and "unrestricted subtree detection" report the individual results for the branch-and-count framework without and with unrestricted subtree detection, respectively; the first subcolumns report the running time in seconds, the total number of search nodes, and the maximum search tree depth. For the unrestricted subtree detection we further state the number of detected (non-trivial) unrestricted subtrees ("unrest.").

The unrestricted subtree detection leads to a substantial decrease in the number of needed search nodes. This comes along with a reduction in the total running time and the maximum depth level of the search tree.

[1] `www.onespin-solutions.com`

We also applied our method to the MIPLIB [5] instances that do not have continuous variables to compute the sets X^*_{IP} of all optimal solutions. The generation of all optimal solutions can be performed in less than 5 minutes for each instance, except for *cracpb1* and *p2756*. For *p0548* the unrestricted subtree detection was necessary to solve the instance within the time limit.

We compared our approach (SCIP) to existing methods, in particular AZOVE [4], CPLEX [8], LattE [9], and zerOne [7]. Our approach clearly dominates the other solvers on the chip verification instances. For the MIPLIB instances the branch-and-cut based algorithms, i.e., CPLEX, zerOne, and SCIP, are similar in efficiency, while the other solvers are inferior.

References

1. T. Achterberg, *Conflict analysis in mixed integer programming*, Discrete Optim., 4 (2007), pp. 4–20.
2. ———, *Constraint Integer Programming*, PhD thesis, TU Berlin, 2007.
3. A. I. Barvinok, *A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed*, Math. Oper. Res., 19 (1994), pp. 769–779.
4. M. Behle and F. Eisenbrand, *0/1 vertex and facet enumeration with BDDs*, in Workshop on Algorithm Engineering and Experiments (ALENEX), 2007.
5. R. E. Bixby, E. A. Boyd, and R. R. Indovina, *MIPLIB: A test set of mixed integer programming problems*, SIAM News, 25 (1992), p. 16.
6. R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, *MIP: Theory and practice – closing the gap*, in Systems Modelling and Optimization: Methods, Theory, and Applications, M. Powell and S. Scholtes, eds., Kluwer, 2000, pp. 19–49.
7. M. R. Bussieck and M. E. Lübbecke, *The vertex set of a 0/1-polytope is strongly $\mathcal{P}$-enumerable*, Comput. Geom, 11 (1998), pp. 103–109.
8. E. Danna, M. Fenelon, Z. Gu, and R. Wunderling, *Generating multiple solutions for mixed integer programming problems*, in Integer Programming and Combinatorial Optimization, M. Fischetti and D. P. Williamson, eds., vol. 4513 of LNCS, 2007, pp. 280–294.
9. J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, J. Symb. Comput., 38 (2004), pp. 1273–1302.
10. A. Morgado, P. J. Matos, V. M. Manquinho, and J. P. M. Silva, *Counting models in integer domains*, in Theory and Applications of Satisfiability Testing – SAT 2006, vol. 4121 of LNCS, 2006, pp. 410–423.
11. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1988.
12. T. Sandholm and R. Shields, *Nogood learning for mixed integer programming*, Tech. Rep. CMU-CS-06-155, Carnegie Mellon University, Computer Science Department, 2006.
13. M. Thurley, *sharpSAT - counting models with advanced component caching and implicit BCP*, in Theory and Applications of Satisfiability Testing – SAT 2006, vol. 4121 of LNCS, 2006, pp. 424–429.
14. R. Wunderling, *Paralleler und objektorientierter Simplex-Algorithmus*, PhD thesis, TU Berlin, 1996.