Upper Bounds on the Number of Solutions of Binary Integer Programs

Siddhartha Jain, Serdar Kadioglu,* Meinolf Sellmann*

Brown University, Department of Computer Science
115 Waterman Street, P.O. Box 1910, Providence, RI 02912
sj10,serdark,sello@cs.brown.edu

Abstract. We present a new method to compute upper bounds of the number of solutions of binary integer programming (BIP) problems. Given a BIP, we create a dynamic programming (DP) table for a redundant knapsack constraint which is obtained by surrogate relaxation. We then consider a Lagrangian relaxation of the original problem to obtain an initial weight bound on the knapsack. This bound is then refined through subgradient optimization. The latter provides a variety of Lagrange multipliers which allow us to filter infeasible edges in the DP table. The number of paths in the final table then provides an upper bound on the number of solutions. Numerical results show the effectiveness of our counting framework on automatic recording and market split problems.

Keywords: solution counting, CP-based Lagrangian relaxation, surrogate relaxation, dynamic programming

1 Introduction

Solution counting has become a new and exciting topic in combinatorial research. Counting solutions of combinatorial problem instances is relevant for example for new branching methods [23, 24]. It is also relevant to give user feedback in interactive settings such as configuration systems. Moreover, it plays an ever more important role in post-optimization analysis to give the user of an optimization system an idea how many solutions there are within a certain percentage of the optimal objective value. The famous mathematical programming tool Cplex for example now includes a solution counting method. Finally, from a research perspective the problem is interesting in its own right as it constitutes a natural extension of the mere optimization task.

Solution counting is probably best studied in the satisfaction (SAT) community where a number of approaches have been developed to estimate the number of solutions of under-constrained instances. First attempts to count the number of solutions often simply consisted in extending the run of a solution finding systematic search after a first solution has been found [3]. More sophisticated randomized methods estimate upper and lower bounds with high probability. In [8], e.g., in a trial an increasing number of random XOR constraints are added to the problem. The upper and lower bounds

* This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

on the number of solutions depends on how many XORs can be added before the instance becomes infeasible, whereby the probability that the bound is correct depends on the number of trials where (at least or at most) the same number of XORs can be added before the instance changes its feasibility status.

An interesting trend in constraint programming (CP) is to estimate solution density via solution counting for individual constraints [23, 24]. Since the solution density information is used for branching, it is important that these methods run very fast. Consequently, they are constraint-based and often give estimates on the number of solutions rather than hard upper and lower bounds or bounds that hold with high probability.

In mathematical programming, finally, the IBM Cplex IP solution counter [5, 10] enumerates all solutions while aiming at finding diverse set of solutions, and the Scip solution counter finds the number of all feasible solutions using a technique to collect several solutions at once [1]. Stopped prematurely at some desired time-limit, these solvers provide lower bounds on the number of solutions.

Considering the literature, we find that a big emphasis has been laid on the computation of lower bounds on the number of solutions of a given problem instance. Apart from the work in [8] and the upper bounding routine for SAT in [11], we are not aware of any other approaches that provide hard or high probability upper bounds on the number of solutions. Especially solution counters that are part of the IBM Cplex and the Scip solver would benefit if an upper bound on the number of solutions could be provided alongside the lower bound in case that counting needs to be stopped prematurely.

With this study we attempt to make a first step to close this gap. In particular, we consider binary integer programs and propose a general method for computing hard upper bounds on the number of feasible solutions. Our approach is based on the exploitation of relaxations, in particular surrogate and Lagrangian relaxations. Experimental results on automatic recording and market split problems provide a first proof of concept.

2 Upper Bounds on the Number of Solutions for Binary Integer Programs

We assume that the problem instance is given in the format

$$\begin{aligned} (BIP) \quad & p^T x \geq B \\ & Ax \leq b \\ & x_i \in \{0,1\}. \end{aligned}$$

Wlog, we assume that the profit coefficients are integers. Although we could multiply the first inequality with minus one, we make it stand out as the original objective of the binary integer program (BIP) that was to be maximized. Usually, in branch and bound approaches, we consider relaxations to compute upper bounds on that objective. For example, we may solve the linear program (LP)

$$\begin{aligned} Maximize \quad & L = p^T x \\ & Ax \leq b \\ & 0 \leq x_i \leq 1 \end{aligned}$$

and check whether $L \geq B$ for an incumbent integer solution with value B to prune the search.

For our task of computing upper bounds on the number of solutions, relaxing the problem is the first thing that comes to mind. However, standard LP relaxations are not likely to be all that helpful for this task. Assume that there are two (potentially fractional) solutions that have an objective value greater or equal B. Then, there exist infinitely many fractional solutions that have the same property.

Consequently, we need to look for a relaxation which preserves the discrete character of the original problem. We propose to use the surrogate relaxation for this purpose. In the surrogate relaxation, we choose multipliers $\lambda_i \geq 0$ for each linear inequality constraint i and then aggregate all constraints into one. We obtain:

$$\begin{array}{ll} Maximize & S = p^T x \\ & \lambda^T A x \leq \lambda^T b \\ & x_i \in \{0,1\}. \end{array}$$

This problem is well known, it is a knapsack problem (that may have negative weights and/or profits). Let us set $w \leftarrow w_\lambda \leftarrow A^T\lambda$ and $C \leftarrow C_\lambda \leftarrow \lambda^T b$. Then, we insert the profit threshold B back into the formulation. This is sound as S is a relaxation of the original problem. We obtain a knapsack constraint

$$\begin{array}{ll} (KP) & p^T x \geq B \\ & w^T x \leq C \\ & x_i \in \{0,1\}. \end{array}$$

2.1 Filtering Knapsack Constraints

In [12], knapsack constraints were studied in great detail and exact pseudo-polynomial time filtering algorithms were developed which are based on a dynamic programming formulation of knapsack. First, the knapsack instance is modified so that all profits p_i are non-negative. This can be achieved by potentially replacing a binary variable x_i (in the context of knapsack we often refer to the index i as an *item*) with its 'negation' $x'_i = 1 - x_i$. Then, in a cell $M_{q,k}$ we store the minimum knapsack weight needed to achieve exactly profit q when only items $\{1, \ldots, k\}$ can be included in the knapsack (i.e., when all variables in $\{x_{k+1}, \ldots, n\}$ are set to 0). Then, the following recursion equation holds:

$$M_{q,k} = \min\{M_{q,k-1}, M_{q-p_k,k-1} + w_k\}. \tag{1}$$

To filter the constraint we interpret the DP as a weighted directed acyclic graph (DAG) where the cells are the nodes and nodes that appear in the same recursion are connected (see left graph in Figure 1). In particular, we define $G = (V, E, v)$ by setting

- $V_M := \{M_{q,k} \mid 0 \leq k \leq n\}$.
- $V := V_M \cup \{t\}$.
- $E_0 := \{(M_{q,k-1}, M_{q,k}) \mid k \geq 1,\ M_{q,k} \in V_M\}$.
- $E_1 := \{(M_{q-p_k,k-1}, M_{q,k}) \mid k \geq 1,\ q \geq p_k,\ M_{q,k} \in V_M\}$.
- $E_t := \{(M_{q,n}, t) \mid q \geq B,\ M_{q,n} \in V_M\}$.
- $E := E_0 \cup E_1 \cup E_t$.

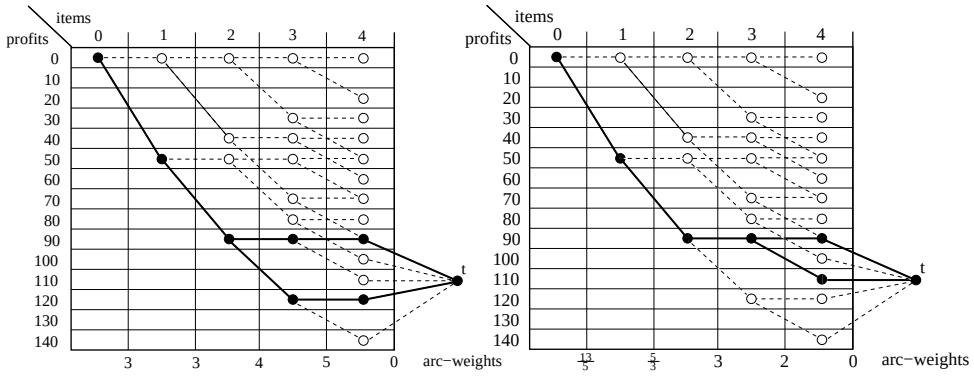

Fig. 1. The figure shows dynamic programming tables for a knapsack constraint with four variables, profits $p^T = (50, 40, 30, 20)$, and profit constraint threshold $B = 82$. In the left figure the weights are $w^T = (3, 3, 4, 5)$, and the knapsack's capacity is $C = 10$. In the right figure the weights are $w^T = (13/5, 5/3, 3, 2)$, and the capacity is $C = 19/3$. The node-labels are defined by their row and column number, the sink node t is marked separately. The value of non-horizontal arcs that cross a vertical line is given under that line, horizontal arcs have weight 0. Hollow nodes and dashed arcs mark those nodes and arcs that are removed by the filtering algorithm, because there exists no path from $M_{0,0}$ to t with weight lower or equal C that visits them.

- $v(e) := 0$ for all $e \in E_0 \cup E_t$.
- $v(M_{q-p_k,k-1}, M_{q,k}) := w_k$ for all $(M_{q-p_k,k-1}, M_{q,k}) \in E_1$.

We consider the DAG G because there is a one-to-one correspondence between paths from $M_{0,0}$ to t and variable instantiations that yield a profit greater than B. Moreover, the length of such a path is exactly the weight of the corresponding instantiation. Therefore, every path from $M_{0,0}$ (the *source*) to t (the *sink*) with length lower or equal C defines a feasible, improving solution (we call such paths *admissible*). Vice versa, every feasible, improving solution also defines an admissible path from source to sink with length lower or equal C.

The filtering algorithm in [12] removes edges from G that cannot be part of any admissible path. This is done using a filtering routine for shorter path constraints on DAGs from [13]: We first compute the shortest path distances from the source to all nodes using the topological ordering of the DAG, thus being able to handle shortest path problems even non-negative edge weights in time linear in the size of the graph. In an analogous second pass which begins at the sink we compute the shortest path distances to the sink. Equipped with both distances for each node, we can compute the shortest path lengths from source to sink through each edge in the graph – and remove all edges and nodes which cannot lie on any admissible path.

2.2 Upper Bounds on the Number of Solutions

In [12], the resulting DP is analyzed using a technique from [20, 21] to identify which variables cannot take value 0 or value 1. We do not perform this last step. Instead, we use the resulting DP to count the number of paths from source to sink using the technique in [23, 24]. Note that any solution to (BIP) fulfills the redundant constraint

(KP) and therefore corresponds to an admissible path in our DP. Moreover, two different solutions also define two different paths in the DP. Therefore, the number of paths in the DP gives an upper bound on the number of solutions in (BIP).

Now, the quality of the upper bound will depend on the choice of the initial vector λ. In ordinary optimization, we aim for a choice of λ for which the surrogate relaxation gives the tightest relaxation value. However, for the purpose of filtering we know that sub-optimal multipliers λ can provide better filtering effectiveness [14]. Consider the following example:

$$\begin{array}{ll} (EX) & 50x_1 + 40x_2 + 30x_3 + 20x_4 \geq 82 \\ & 3x_1 + x_2 + 3x_3 \leq 5 \\ & 2x_2 + x_3 + 5x_4 \leq 5 \\ & x_i \in \{0, 1\}. \end{array}$$

If we use $\lambda = (1, 1)^T$, e.g., then we get the knapsack constraint as shown in the left graph of Figure 1 with a relaxation value of 120 (as that is the highest profit visited by the remaining admissible paths). On the other hand, had we chosen $\lambda = (13/15, 6/15)^T$, we would have obtained the knapsack constraint in the right graph of Figure 1 with an improved upper bound of 110.

Comparing the two DPs, we find that the two choices for λ yield incomparable filtering effectiveness. Although the second set of multipliers gives a strictly better upper bound, it cannot remove the edge $(M_{90,3}, M_{110,4})$. On the other hand, the second choice for λ allows us to remove the edges $(M_{90,2}, M_{120,3})$ and $(M_{120,3}, M_{120,4})$. This effect has been studied before in [14]. The explanation for the different filtering behavior is that, in principle, each edge has its own vector λ that maximally challenges admissibility (as measured by the shortest path length through that edge).

In principle, we could employ a probing procedure. For each edge, we remove all edges on the same level, thus enforcing that each path from source to sink must pass through this edge. Then, we start with some selection for λ and compute the shortest path length according to the corresponding weights w_λ as well as the corresponding BIP solution x_λ. If $w_\lambda^T x_\lambda > C_\lambda$, then we can remove the edge. Otherwise, we modify λ to minimize $C_\lambda - w_\lambda^T x_\lambda$ as much as possible. From the theory of Lagrangian relaxation (see for example [2]) we know that finding the optimal choice for λ consists in minimizing a piecewise linear convex function. Consequently, we can use a subgradient search algorithm to find the vector $\lambda \geq 0$ which will minimize $C_\lambda - w_\lambda^T x_\lambda$ as much as possible and thus enable us to decide whether any λ exists that would allow us to remove the edge under consideration.

The problem with this procedure is of course that it takes way too much time to probe each individual edge. Instead, we follow the same method as in CP-based Lagrangian relaxation [15]. That is, we employ a subgradient search to find a vector λ that minimizes $C_\lambda - w_\lambda^T x_\lambda$ in the DP. Then, for each λ that the subgradient search considers, we use our edge-filtering algorithms to remove edges from the graph. That way, we hope to visit a range of different settings for λ that will hopefully remove a large percentage of edges in the DP that can be discarded.

Consider again our example (EX) from before. If we first prune the graph with respect to the weight vector w from the left graph in Figure 1 and then, in the pruned

graph, remove edges based on the weight vector w from the right graph in Figure 1, then we end up with only one path which corresponds to the only solution to (EX) which is $x = (1, 1, 0, 0)^T$.

2.3 The Algorithm

Algorithm 1 BIP Counting Algorithm

1: Negate binary variables with profit $p_i < 0$.
2: Set up the graph G for $\{x \in \{0,1\}^n \mid p^T x \geq B\}$.
3: Initialize λ.
4: **while** subgradient method not converged **do**
5: Set $w \leftarrow \lambda^T A$, $C \leftarrow \lambda^T b$.
6: Propagate $w^T x \leq C$ in G removing inadmissible edges.
7: Compute the solution x that corresponds to the shortest path from source to sink in (G, w).
8: Update λ according to the current gap $C - w^T x$ and the subgradient $Ax - b$.
9: **end while**
10: Count the number of paths from source to sink in G and return that number.

The complete procedure is sketched in Algorithm 1. Note how we first increase the number of solutions by considering the cardinality of the set $R \leftarrow \{x \in \{0,1\}^n \mid p^T x \geq B\}$ instead of $P \leftarrow \{x \in \{0,1\}^n \mid p^T x \geq B \;\&\; Ax \leq b\}$. Then, to reduce the number of solutions again, we heuristically remove edges from the DP that has exactly one path for each $x \in R$ by propagating constraints $\lambda^T Ax \leq \lambda^T b$ for various choices of λ in the DP. The resulting number of paths in the DP gives a hard upper bound on the number of solutions to the original BIP.

2.4 Strengthening the Bound – Cutting Planes, Tree Search, Compatibility Labels, and Generate and Check

A nice property of our approach is that we can use all the usual methods for strengthening linear continuous relaxations, such as preprocessing and especially adding valid inequalities, so-called cutting planes, to the BIP which tighten the continuous relaxation.

To strengthen the upper bound on the solution count further, we can embed our procedure in a branch-and-bound tree search algorithm which we truncate at some given depth-limit. The sum of all solutions at all leafs of the truncated tree then gives an upper bound on the number of solutions.

For very hard combinatorial counting problems we may consider doing even more. In our outline above, we use the profit constraint to define the graph G. In principle, we could use any vector μ of natural numbers and consider the constraint $(p^T - \mu^T A)x \geq B - \mu^T b$ to set up the DP. This is needed in particular when there is no designated objective function. We notice, however, that we do not need to restrict us to using just one DP. Instead, we can set up multiple DPs for different choices of μ.

The simplest way to strengthen the upper bound on the number of solutions is to take the minimum count over all DPs. However, we can do much better than that. Following

an idea presented in [9], we can compute compatibility labels between the different DPs: Let us denote with G_A and G_B the graphs that correspond to two different DPs for our problem. Our filtering algorithm ensures that each edge in the graph is visited by at least one admissible path. The compatibility labels from [9] aim to ensure that an edge in G_A is also supported by a (not necessarily admissible) path in G_B. More precisely, for each edge in G_A we ensure that there is a path from source to sink in G_A that visits the edge and which corresponds to a solution which also defines a path from source to sink in G_B.

Finally, if we have found an upper bound on the solution count that is rather small, we can generate all potential solutions which is very easy given our DAG G. Then, we test each assignment for feasibility and thus provide an exact count.

3 Numerical Results

3.1 Automatic Recording

We first consider the automatic recording problem (ARP) that was introduced in [15].

3.2 Problem Formulation

The technology of digital television offers to hide meta-data in the content stream. For example, an electronic program guide with broadcasting times and program annotation can be transmitted. An intelligent video recorder like the TIVO$^{\mathrm{tm}}$ system [19] can exploit this information and automatically record TV content that matches the profile of the system's user. Given a profit value for each program within a predefined planning horizon, the system has to make the choice which programs shall be recorded, whereby two restrictions have to be met:

- The disk capacity of the recorder must not be exceeded.
- Only one program can be recorded at a time.

While the problem originally emerged from automatic video recording, it has other applications, for example in satellite scheduling. Various algorithms for the ARP have been studied in [15, 14, 16, 17]. The problem can be stated as a binary integer program:

$$\begin{array}{lll} Maximize & p^T x & \\ & w^T x \leq K & \\ & x_i + x_j \leq 1 & \forall\, 0 \leq i \leq j \leq n, I_i \cap I_j \neq \emptyset \\ & x \in \{0,1\}^n & \end{array} \qquad \text{(ARP 1)}$$

where p_i and w_i represent the profit and the storage requirement of program i, K is the storage capacity, and $I_i := [startTime(i), endTime(i)]$ corresponds to the broadcasting interval of program i. The objective function maximizes the user satisfaction while the first constraint enforces the storage restrictions. Constraints of the form $x_i + x_j \leq 1$ ensure that at most one program is recorded at each point in time.

This formulation can be tightened by considering the conflict graph and adding the corresponding clique constraints to the formulation [15].

Definition 1. *The set $C \subseteq V$ is called a conflict clique iff $I_i \cap I_j \neq \emptyset \; \forall \; i, j \in C$. A conflict clique C is called maximal iff $\forall \; D \subseteq V, D$ conflict clique: $C \subseteq D \Rightarrow C = D$. Let $M := \{C_0, \ldots, C_{m-1}\} \subseteq 2^V$ the set of maximal conflict cliques.*

These clique constraints are obviously valid inequalities since, if $x_i + x_j \leq 1$ for all overlapping intervals, it is also true that $\sum_{i \in C_p} x_i \leq 1 \; \forall \; 0 \leq p \leq m$. We can therefore add the clique constraints to our original formulation.

$$
\begin{array}{lll}
Maximize & p^T x & \\
 & w^T x \leq K & \\
 & x_i + x_j \leq 1 & \forall \; 0 \leq i \leq j \leq n, I_i \cap I_j \neq \emptyset \\
 & \sum_{i \in C_p} x_i \leq 1 & \forall \; 0 \leq p \leq m \\
 & x \in \{0,1\}^n &
\end{array}
\qquad \text{(ARP 2)}
$$

Though being NP-complete on general graphs, finding maximal cliques on the graph defined by our application is simple:

Definition 2. *A graph $G = (V, E)$ is called an interval graph if there exist intervals $I_1, \ldots, I_{|V|} \subset \mathbb{R}$ such that $\forall v_i, v_j \in V : (v_i, v_j) \in E \iff I_i \cap I_j \neq \emptyset$.*

On interval graphs, the computation of maximal cliques can be performed in $O(n \log n)$ [7]. Hence, ARP 2 can be obtained in polynomial time.

3.3 Solution Counting for the ARP

We will now describe how we apply our counting algorithm to the ARP problem.

Initialization: The graph G for our ARP formulation is set up using the equation $w^T x \leq K$, where w_i represents the storage requirement of program i and K is the storage capacity.

Tree Search, and Generate and Test: To strengthen the quality of our bounds on the number of solutions, we employ a truncated tree search as described earlier. For branching, we select the variable with the highest knapsack efficiency p_i/w_i which is also selected in the shortest path in the DP according to the final multipliers λ. When we get total solution counts below 100 we generate all solutions and test them for feasibility.

Subgradient Optimization: At every choice point, we conduct the subgradient search using the object bundle optimization package from Frangioni [6]. On top of filtering with respect to the vectors λ that the subgradient optimizer visits, we also propagate the constraint $w^T x \leq K$ in the DP at every choice point. At leaf nodes, also choose randomly 3% of the original constraints in ARP 1 or ARP 2 and propagate them to prune the DPs one last time before counting the number of paths from source to sink.

3.4 Experimental Results

We used a benchmark set described in [15, 14] which can downloaded at [18]. This benchmark set consists of randomly generated instances which are designed to mimic features of real-world instances for the automatic recording of TV content. For our

		ARP-1		ARP-2	
Inst.	Gap	Count	Time	Count	Time
0	0%	2	20	2	3.2
1	0%	3	10	1	1.5
2	0%	1	16	1	2.8
0	1%	2.27E+10	90	1.60E+10	38.8
1	1%	3.26E+05	12	2.09E+05	3.2
2	1%	8.36E+07	33	3.69E+07	9.5
0	2%	7.51E+12	133	8.77E+11	73.8
1	2%	9.06E+05	13	4.56E+05	4.3
2	2%	2.87E+09	68	1.33E+09	24

		ARP-1		ARP-2	
Inst.	Gap	Count	Time	Count	Time
0	0%	39	1109	39	34
1	0%	203	933	35	20
2	0%	15	1146	15	22
0	1%	6.54E+43	2636	7.95E+35	353
1	1%	7.82E+10	1100	3.75E+10	73
2	1%	5.25E+23	314	1.05+23	294
0	2%	4.75E+59	5169	6.81E+52	992
1	2%	2.57E+13	3639	8.06E+12	221
2	2%	1.33E+26	6873	3.08E+24	893

Table 1. Numerical Results for the ARP Problem. We present the upper bound on the number of solutions and the CPU-time in seconds for the binary constraint model (ARP-1) and the maximal clique model (ARP-2). The table on the left is for the small sized data set (20-720) with 20 channels and 720 minute time horizon, and the table on the right is for the large sized data set (50-1440) with 50 channels and 1440 minute time horizon. In this experiment, we do not generate and check solutions for feasibility.

experiments, we use the class usefulness (CU) instances. We consider a small sized data set which spans half a day (720 minutes) and consists of 20 channels, and a large sized data set which spans a full day (1440 minutes) and consists of 50 channels. Profits for each program are chosen based on the class that a program belongs to. This class also determines the parameters according to which its length is randomly chosen. On average, these instances have 300 and 1500 programs, respectively. All experiments in this paper were performed on a machine with Intel Core 2 Quad Q6600, 2.4GHz CPUs and 2GByte of RAM operating Linux Debian 5.0.3 32-bit. On all experiments, we enforced a time limit of 3 hours CPU time.

Our first evaluation compares the effectiveness of the models described by ARP 1 and ARP 2 in terms of the upper bound on the solution count that they provide and the time they take. Specifically, we are interested in the increase of the number of solutions as we move away from the optimal value. To this end, we introduce the *Gap* parameter which indicates the percentage gap between a threshold and the optimal value. We only consider solutions that achieve an objective value above the threshold. We experiment with objective gaps of 0%, 1% and 2% and truncate the search at depth 5. Table 1 shows that the ARP 2 formulation which includes the clique cuts provides much better upper bounds than ARP 1 in substantially less time. This indicates that exploiting the common methods for strengthening LP relaxations can also be exploited effectively to compute superior upper bounds on the number of solutions of BIPs. The fact that ARP 2 actually runs faster can be attributed to the fact that the cutting planes allow much better edge-filtering effectiveness. Therefore, the DP contains much fewer edges higher up in the tree, which leads to much faster times per choice point.

We next compare our approach (UBound) with the Cplex IP solution counter which enumerates all solutions [10, 5] and the Scip solution counter which collects several solutions at a time. Note that Cplex and Scip provide only a lower bound in case they time out or reach the memory limit. We again consider objective gaps 0%, 1% and 2%.

	Cplex		Scip		Ubound	
Inst.	Count	Time	Count	Time	Count	Time
0	**2**	0.17	**2**	0.3	**2**	3.16
1	**1**	0.03	**1**	0.05	**1**	1.53
2	**1**	0.08	**1**		**1**	2.75
3	**1**	0.04	**1**	0.03	**1**	1.71
4	**1**	0.06	**1**	0.06	**1**	2.46
5	**12**	0.62	**12**	0.16	**12**	3.83
6	**6**	0.17	**6**		**6**	2.47
7	**1**	0.07	**1**	0.03	**1**	1.60
8	**1**	0.09	**1**	0.06	**1**	2.45
9	**3**	0.37	**3**	0.04	**3**	2.30

	Cplex		Scip		Ubound	
Inst.	Count	Time	Count	Time	Count	Time
0	**39**	182	**39**	1.91	**39**	34.3
1	35	T	**35**	100	**35**	20.7
2	**14**	0.98	**14**	1.54	**14** (15)	30.5
3	**6**	0.64	**6**	0.25	**6**	30.2
4	**20**	2.52	**20**	0.51	**20**	30.8
5	**1**	0.34	**1**	0.4	**1**	20.9
6	**33**	3.95	**33**	71	**33** (39)	27.5
7	**1**	0.49	**1**	0.31	**1**	58.0
8	**4**	2.16	**4**	1.95	**4**	69.6
9	**6**	27.1	**6**	1.81	**6**	43.8

Table 2. Numerical Results for the ARP Problem with 0% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds at depth 5. The table on the left is for the small sized data set (20-720) with 20 channels and 720 minute time horizon, and the table on the right is for the large sized data set (50-1440) with 50 channels and 1440 minute time horizon. 'T' means that the time limit has been reached. The numbers in bold show exact counts and the numbers in parenthesis are our upper bounds before we generate and check solutions for feasibility.

For 0% gap, we run our method with depth 5 which is adequate to achieve the exact counts. For higher gaps, we present the results for depths 5, 10, and 15.

Our results are presented in Table 2, Table 3, and Table 4. For the optimal objective threshold, UBound provides exact counts for all test instances. In terms of running time, UBound does not perform as quickly as the IBM Cplex and the Scip solution counter. There is only one notable exception to this rule, instance 50-1440-1. On this instance, Scip takes 100 seconds and Cplex times out after three hours while our method could have provided the 35 solutions to the problem in 20 seconds.

This discrepancy becomes more evident when we are interested in the number of solutions that are with 1% or 2% of the optimum. As we can see from Table 3 and Table 4 the number of solutions increases very rapidly even for those small objective gaps. Not surprisingly, the counts obtained by Cplex and Scip are limited by the number of solutions they can enumerate within the memory and the time constraints, yielding a count of roughly 1E+5 to 1E+7 solutions in most cases. Due to the explosion in the number of solutions, Cplex and Scip are never able to give exact counts for the large instances but only give a lower bound. Cplex hits the time cutoff in 17 out of 20 large instances and reaches the memory limit for the remaining 3, and Scip times out in all large instances. In most cases where Cplex or Scip are able to find the exact counts, UBound is able to provide tight upper bounds that are not more than an order of magnitude bigger. In Figure 2, we show how the upper and lower bounds obtained by UBound, Cplex, and Scip progress as they approach the exact count.

We also compared our approach with the method from [8] which provides very good bounds on the number of solutions for constraint satisfaction problems. The method is based on the addition of random XOR-constraints. Unfortunately, we found that, in combination with an integer programming problem, the method does not perform well.

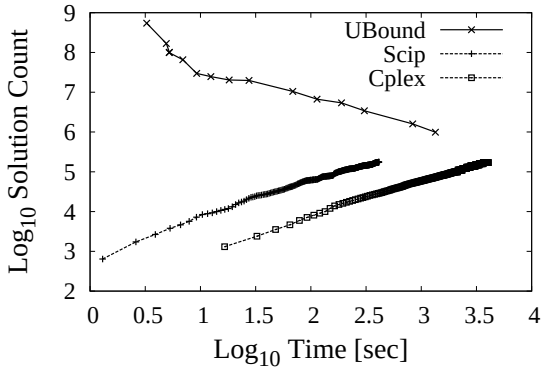

Fig. 2. Solution Count for the instance 20-720-2 with 1% objective gap. We present the progress of the upper bound obtained by UBound and the lower bounds obtained by Cplex and Scip as time progresses. The time and solution count are given on a logarithmic scale of base 10. We run UBound until depth 17 which is within the time that Cplex reaches the memory limit.

We tried using the vanilla code[1] which was designed for pure CSPs. It did not perform well for the ARP. So we modified the code, providing better branching variables for the tree search and using linear bounds to prune the search. That improved the performance. With this approach we are able to compute lower bounds, but computing these takes more time and the counts are worse than those provided by Cplex and Scip. Upper bounds take even more time as the XOR constraints involve more variables. We could not obtain upper bounds within the time limit of three hours. We conjecture that a tight integration between the XOR constraints and linear inequalities would be needed to make this approach, which gives very good results for CSPs, work well for optimization problems.

3.5 Market Split

We next consider the market split problem (MSP), a benchmark that was suggested for knapsack constraints in [20, 21].

3.6 Problem Formulation

The original definition goes back to [4, 22]: A large company has two divisions D_1 and D_2. The company supplies retailers with several products. The goal is to allocate each retailer to either division D_1 or D_2 so that D_1 controls A% of the company's market for each product and D_2 the remaining (100-A)%. Formulated as an integer program, the problem reads:

$$\begin{array}{cc} \sum_j a_{ij} x_j = \lfloor \frac{A}{100} \sum_j a_{ij} \rfloor & \forall\, 0 \leq i < m \\ x_j \in \{0,1\} & \forall\, 0 \leq j < n, \end{array}$$

whereby m denotes the number of products, n is the number of retailers, and a_{ij} is the demand of retailer j of product i. MSPs are generally very hard to solve, especially the randomly generated instances proposed by Cornuejols and Dawande where weight coefficients are randomly chosen in $[1, \ldots, 100]$ and $A = 50$. Special CP approaches for the MSP have been studied in [20, 21, 14, 9].

[1] Many thanks to Ashish Sabharwal for providing us the source code!

	Cplex		Scip		UBound					
					Depth 5		Depth 10		Depth 15	
Instance	Count	Time	Count	Time	Count	Time	Count	Time	Count	Time
20-720-0	5.20E+05	M	**1.01E+06**	2518	1.60E+10	38.8	1.97E+08	137	3.31E+07	1183
20-720-1	**3.15E+04**	175	**3.15E+04**	20.3	2.09E+05	3.16	1.48E+05	7.52	1.02E+05	40.9
20-720-2	1.77E+05	M	**1.77E+05**	414	3.69E+07	9.51	1.36E+07	45.2	2.87E+06	622
20-720-3	**2.09E+02**	3.39	**2.09E+02**	0.25	4.05E+02	4.19	2.99E+02	12.5	2.48E+02	40.5
20-720-4	**5.20E+03**	76	**5.20E+03**	6.7	1.13E+05	7.24	1.79E+04	23.1	1.02E+04	122
20-720-5	**2.00E+04**	174	**2.00E+04**	22.5	1.58E+12	22.2	6.81E+08	58.8	4.50E+04	228
20-720-6	**5.45E+04**	932	**5.45E+04**	153	2.00E+07	10.9	3.96E+06	46.5	1.68E+06	431
20-720-7	**9.80E+01**	1.68	**9.80E+01**	0.07	1.04E+02	2.82	1.04E+02	6.70	1.03E+02	16.7
20-720-8	**1.77E+05**	1386	**1.77E+05**	298	3.41E+09	40.7	3.42E+07	191	9.00E+06	899
20-720-9	**1.88E+03**	35.5	**1.88E+03**	1	3.66E+03	4.23	3.48E+03	17	2.99E+03	87.8
50-1440-0	1.95E+04	T	1.15E+07	T	7.95E+35	353	1.21E+34	2572	[1.21E+34]	T
50-1440-1	5.59E+04	T	1.11E+07	T	3.75E+10	73.8	2.21E+10	305	1.85E+10	3025
50-1440-2	7.63E+04	T	1.77E+06	T	1.05E+23	293	1.76E+21	2635	[1.76E+21]	T
50-1440-3	6.00E+04	T	9.48E+06	T	3.56E+16	149	2.34E+15	452	2.45E+14	3333
50-1440-4	7.13E+04	T	7.29E+05	T	4.15E+21	412	4.31E+19	1852	[4.31E+19]	T
50-1440-5	9.33E+04	M	1.04E+06	T	3.28E+10	90.4	7.06E+09	314	6.17E+09	4093
50-1440-6	1.20E+05	M	3.03E+06	T	7.53E+12	101	2.44E+12	350	4.12E+11	3483
50-1440-7	4.92E+04	T	1.96E+06	T	1.04E+20	396	6.03E+18	3037	[6.03E+18]	T
50-1440-8	8.90E+04	T	3.75E+05	T	5.56E+27	719	1.44E+25	3776	[1.44E+25]	T
50-1440-9	8.35E+04	M	9.55E+05	T	2.89E+14	259	2.01E+13	434	2.09E+06	578

Table 3. Numerical Results for the ARP Problem with 1% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds. 'T' means that the time limit has been reached and 'M' indicates a solver has reached the memory limit. The numbers in bold show exact counts and the numbers in square brackets denote the best count UBound could achieve within the time limit.

3.7 Solution Counting for the MSP

Initialization: Our MSP formulation does not have an objective function, thus we construct the graph G using the equation $\lambda^T Ax \geq \lambda^T b$, where $\lambda_i = 5^{i-1}$ as proposed in [20, 21].

Compatibility Labels, and Generate and Test: For the MSP, we strengthen the solution counts by employing the compatibility labels introduced in [9]. We additionally set up the DPs for the original equations in the problem. If there are $m > 3$ constraints in the MSP, we set up $m - 2$ DPs where the kth DP is defined by the sum of the kth constraint plus five times the k+first constraint plus 25 times the k+second constraint.

Often, the number of solutions to MSP instances is comparably low, and checking feasibility is very fast. In case that we find an upper bound of less than 50,000 we simply generate and check those solutions for feasibility. Therefore, each number that is less than 50,000 is actually an exact count.

	Cplex		Scip		UBound					
					Depth 5		Depth 10		Depth 15	
Instance	Count	Time	Count	Time	Count	Time	Count	Time	Count	Time
20-720-0	6.80E+05	M	1.14E+07	T	8.77E+11	73.8	9.23E+09	326	2.30E+09	4002
20-720-1	**1.87E+05**	969	**1.87E+05**	49.5	4.56E+05	4.31	4.01E+05	8.76	3.24E+05	46.2
20-720-2	3.00E+05	T	**8.77E+06**	6528	1.33E+09	24.3	1.65E+08	218	5.07E+07	2276
20-720-3	**4.95E+02**	5.77	**4.95E+02**	0.42	6.60E+02	5.80	5.26E+02	24.2	5.21E+02	75.8
20-720-4	**8.89E+04**	1335	**8.89E+04**	73.5	3.94E+06	10.3	3.26E+05	53.3	2.36E+05	274
20-720-5	3.30E+05	M	**3.32E+05**	618	1.27E+15	43.9	1.15E+13	277	1.86E+09	1540
20-720-6	2.80E+05	M	**3.12E+06**	1966	3.80E+08	19.3	9.20E+07	84.9	6.24E+07	911
20-720-7	**1.35E+02**	2.09	**1.35E+02**	0.07	1.38E+02	3.70	1.38E+02	9.94	1.37E+02	27.2
20-720-8	3.00E+05	M	1.39E+07	T	7.16E+11	82.3	4.91E+09	727	[4.91E+09]	T
20-720-9	**4.17E+03**	63.9	**4.17E+03**	2.33	4.88E+03	7.38	4.71E+03	29.1	4.57E+03	135
50-1440-0	3.03E+04	T	3.11E+06	T	6.81E+52	992	[6.81E+52]	T	[6.81E+52]	T
50-1440-1	5.58E+04	T	3.43E+06	T	8.06E+12	221	1.01E+12	1240	[1.01E+12]	T
50-1440-2	1.40E+05	M	8.97E+06	T	3.08E+24	893	[3.08E+24]	T	[3.08E+24]	T
50-1440-3	7.89E+04	T	1.52E+07	T	2.5E+43	460	2.25E+32	1802	[2.25E+32]	T
50-1440-4	1.00E+05	M	1.35E+06	T	8.89E+22	996	[8.89E+22]	T	[8.89E+22]	T
50-1440-5	9.28E+04	M	1.62E+06	T	3.03E+12	252	1.82E+11	1679	[1.82E+11]	T
50-1440-6	1.50E+05	M	1.68E+06	T	2.1E+34	341	1.53E+29	1607	[1.53E+29]	T
50-1440-7	7.66E+04	T	6.18E+06	T	6.9E+37	1281	[6.9E+37]	T	[6.9E+37]	T
50-1440-8	1.10E+05	M	6.73E+05	T	4.87E+30	2264	[4.87E+30]	T	[4.87E+30]	T
50-1440-9	4.65E+04	T	1.12E+07	T	6.91E+46	1075	2.74E+29	3482	[2.74E+29]	T

Table 4. Numerical Results for the ARP Problem with 2% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds. 'T' means that the time limit has been reached and 'M' indicates a solver has reached the memory limit. The numbers in bold show exact counts and the numbers in square brackets denote the best count UBound could achieve within the time limit.

3.8 Experimental Results

For the purpose of solution counting, we consider the Cornuejols-Dawande instances as described before. Many of these instances are actually infeasible. When there are m constraints, Cornuejols and Dawande introduce $10(m-1)$ binary variables. We introduce more variables to create less and less tightly constrained instances which have more solutions. We compare UBound again with the counts provided by IBM Cplex and Scip. As before, Cplex and Scip provide a lower bound in case they time out. We consider MSPs of orders 3 and 4 with an increasing number of variables between 24 and 38.

We present our results in Table 5. As we can see, UBound provides high quality upper bounds very quickly as shown in the counts given in brackets. Using the generate and test technique, on all instances we are able to provide exact counts in considerably less time than Cplex and Scip.

			Cplex		Scip		Ubound	
Ins	Order	#Vars	Count	Time	Count	Time	Count	Time
1	3	24	**2**	1.78	**2**	5.7	**2**	3.92
2	3	24	**0**	0.91	**0**	3.76	**0**	0.53
3	3	24	**0**	1.24	**0**	2.94	**0**	0.51
4	3	30	**32**	39	**32**	107	**32** (36)	13
5	3	30	**70**	70	**70**	117	**70** (82)	21
6	3	30	**54**	78	**54**	174	**54** (58)	25
7	3	36	**2.3K**	1962	**2.3K**	5118	**2.3K** (32K)	176
8	3	36	292	T	**2.3K**	9203	**2.3K** (23K)	164
9	3	36	569	T	**2K**	5656	**2K** (14K)	130

			Cplex		Scip		Ubound	
Ins	Order	#Vars	Count	Time	Count	Time	Count	Time
10	4	34	**2**	5707	**2**	1087	**2**	198
11	4	34	**0**	396	**0**	1088	**0**	189
12	4	34	**2**	109	**2**	955	**2**	190
13	4	36	**6**	1227	**6**	4175	**6**	301
14	4	36	**2**	753	**2**	2400	**2**	266
15	4	36	**6**	366	**6**	2470	**6**	278
16	4	38	**12**	4422	11	T	**12**	412
17	4	38	9	T	29	T	**36**	405
18	4	38	**44**	3391	43	T	**44**	401

Table 5. Numerical Results for the MSP Problem. We present the upper bound on the number of solutions found and the CPU-time taken in seconds for the binary constraint model and the maximal clique model. 'T' means that the time limit has been reached. The numbers in bold show exact counts. The numbers in parenthesis are our upper bounds before we generate and check solutions for feasibility.

Again, we compared our results also with the XOR approach from [8]. After the vanilla implementation from [8] did not provide competitive results, we devised an efficient code that can solve pure MSPs efficiently and added XOR constraints to it. Again, we found that the problems augmented by XORs are much harder to solve which resulted in the approach timing out on our entire benchmark. We attribute this behavior to our inability to integrate the XOR constraints tightly with the subset-sum constraints in the problem.

4 Conclusions

We presented a new method for computing upper bounds on the number of solutions of BIPs. We demonstrated its efficiency on automatic recording and market split problems. We showed that standard methods for tightening the LP relaxation by means of cutting planes can be exploited also to provide better bounds on the number of solutions. Moreover, we showed that a recent new method for integrating graph-based constraints more tightly via so-called compatibility labels can be exploited effectively to count solutions for market split problems.

We do not see this method so much as a competitor to the existing solution counting methods that are parts of IBM Cplex and Scip. Instead, we believe that these solvers could benefit greatly from providing upper bounds on the number of solutions. This obviously makes sense when the number of solutions is very large and solution enumeration must fail. However, as we saw on the market split problem, considering upper bounds can also boost the performance dramatically on problems that have few numbers of solutions. In this case, our method can be used to give a super-set of potential solutions whose feasibility can be checked very quickly.

References

1. T. Achterberg. SCIP - A Framework to Integrate Constraint and Mixed Integer Programming. `http://www.zib.de/Publications/abstracts/ZR-04-19/`
2. R.K. Ahuja, T.L. Magnati, J.B. Orlin. Network Flows. *Prentice Hall*, 1993.

3. E. Birnbaum and E.L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal Of Artificial Intelligence Research*, 10:457-477, 1999.
4. G. Cornuejols and M. Dawande. A class of hard small 0-1 programs. *Proc. of the 6th Int. IPCO Conference on Integer Programming and Combinatorial Optimization*, 284-293, 1998.
5. E. Danna, M. Fenelon, Z. Gu and R. Wunderling. Generating Multiple Solutions for Mixed Integer Programming Problems *Integer Programming and Combinatorial Optimization (IPCO 2007)*, Vol. 4513, 2007.
6. *Object Bundle Optimization Package* maintained by A. Frangioni. `www.di.unipi.it/optimize/Software/Bundle.html`
7. M.C. Golumbic. Algorithmic Graph Theory and Perfect Graphs. *Academic Press, New York*, 1991.
8. C.P. Gomes, W. Hoeve, A. Sabharwal, B. Selman. Counting CSP Solutions Using Generalized XOR Constraints. *22nd Conference on Artificial Intelligence (AAAI)*, 204-209, 2007.
9. T. Hadzic, E. O'Mahony, B. O'Sullivan, M. Sellmann. Enhanced Inference for the Market Split Problem. *21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 716-723, 2009.
10. IBM. *IBM CPLEX* Reference manual and user manual. V12.1, IBM 2009.
11. L. Kroc, A. Sabharwal and B. Selman. Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, 278-282, 2008.
12. M. Sellmann. Approximated Consistency for Knapsack Constraints. *Proc. of the 9th Int. Conference on the Principles and Practice of Constraint Programming (CP)*, 679-693, 2003.
13. M. Sellmann. Cost-Based Filtering for Shorter Path Constraints. *Proc. of the 9th Int. Conference on the Principles and Practice of Constraint Programming (CP)*, 694-708, 2003.
14. M. Sellmann. Theoretical Foundations of CP-based Lagrangian Relaxation. *Proc. of the 10th Int. Conference on the Principles and Practice of Constraint Programming (CP)*, 634-647, 2004.
15. M. Sellmann, T. Fahle. Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals of Operations Research (AOR)*, 17-33, 2003.
16. M. Sellmann. Approximated Consistency for the Automatic Recording Constraint. *Proc. of the 11th Int. Conference on the Principles and Practice of Constraint Programming (CP)*, 822-826, 2005.
17. M. Sellmann. Approximated Consistency for the Automatic Recording Constraint. *Computers and Operations Research*, Vol. 36(8) 2341-2347, 2009.
18. ARP: *A Benchmark Set for the Automatic Recording Problem* maintained by M. Sellmann. `http://www.cs.brown.edu/people/sello/arp-benchmark.html`
19. TIVO$^{\mathrm{tm}}$ System. `www.tivo.com`
20. M. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *3rd Int. Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, 113-124, 2001.
21. M. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118, 73-84, 2003.
22. H.P. Williams. Model Building in Mathematical Programming. *Wiley: Chicester*, 1978.
23. A. Zanarini and G. Pesant. Solution counting algorithms for constraint-centered search heuristics. *Proc. of the 13th Int. Conference on Principles and Practice of Constraint Programming (CP)*, Vol. 4714, 743–757, 2007.
24. A. Zanarini and G. Pesant. Solution counting algorithms for constraint-centered search heuristics. *Constraints*, Vol. 14(3), pp. 392-413.