

Approximated Consistency for the Automatic Recording Constraint*

Meinolf Sellmann

Brown University, Department of Computer Science
115 Waterman Street, P.O. Box 1910, Providence, RI 02912

`sello@cs.brown.edu`

April 8, 2008

We introduce the automatic recording constraint (ARC) that can be used to model and solve scheduling problems where tasks may not overlap in time and the tasks linearly exhaust some resource. Since achieving generalized arc-consistency for the ARC is NP-hard, we develop a filtering algorithm that achieves approximated consistency only. Numerical results show the benefits of the new constraint on three out of four different types of benchmark sets for the automatic recording problem. On these instances, run-times can be achieved that are orders of magnitude better than those of the best previous constraint programming approach.

Keywords: global constraints, optimization constraints, cost-based filtering, relaxed consistency, approximation algorithms

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

1 Introduction

The most efficient solvers for combinatorial optimization and combinatorial feasibility problems combine two key components: search and inference. From a theoretical point of view, efficient inference techniques that we develop to speed up the solution process for otherwise NP-hard problems are very appealing since we can limit ourselves to studying tractable subtasks for which we can develop provably polynomial algorithms. On the other hand, from a practical point of view, the identification of subtasks that can be tackled efficiently constitutes an extremely helpful guide in the process of engineering a competitive optimization system. Many successful optimization systems have been based on effective and fast inference techniques that speed up the solution process tremendously.

The famous historic example for this approach is of course the solution of linear continuous relaxations in mathematical programming. Here, bounds based on polynomial time computable linear relaxation bounds are used to identify large sub-optimal regions of the search space. While the proof that linear programming (LP) is in P represents one of the great landmarks in algorithmic computer science [9], today the solution of linear continuous relaxations is also the driving force behind the most successful integer programming (IP) solvers (like Cplex or Xpress-MP) which have had and continue to have enormous practical impact.

In constraint programming (CP), we see a lot of similarities to the situation in IP. While solving constraint satisfaction problems (CSPs) is NP-hard in general, efficient inference techniques have been developed that speed up the search for feasible solutions. The most famous and most general example are probably the arc-consistency algorithms for binary CSPs that were developed over almost three decades by now, starting with the pioneering work of Mackworth [14]. Until the present day, the notions of arc-consistency and maintained arc-consistency provide a general algorithm design principle for constraint programming.

A major difference to IP is of course the fact that there exists no standard form in CP. The set of constraints and constraint filtering algorithms that is provided by standard solvers (like Ilog Solver or Eclipse) is under constant review and extension. A major avenue in con-

straint programming research is the identification of relevant constraints whose provision in the constraint language facilitates the modelling of real-world problems while its associated filtering algorithm helps the solver to compute solutions more quickly. In this regard, the identification of tractable subtasks is an ongoing process in CP and finds its expression in the development of stronger constraints and their associated filtering algorithms.

1.1 Optimization Constraints

An important trend in the development of new constraints is the consideration of so-called global constraints that mark a radical departure from the traditional restriction to binary constraints by incorporating all or at least a substantial part of all variables. Global constraints help to simplify a problem description considerably and, at the same time, allow us to develop more effective filtering algorithms. The outstanding example for a global constraint remains the all-different constraint that replaces a quadratic number of binary not-equal constraints while also permitting more powerful filtering [16].

Especially in constraint optimization, global constraints play a decisive role. In order to develop an efficient optimization tool, we need to be able to assess whether we are still able to improve the objective function further. The idea from mathematical programming to incorporate relaxations for this purpose is essential. While in IP relaxations are mostly used for pruning (i.e. the detection and elimination of sub-trees that do not contain leaves with improving objective value), in CP it is only natural to use relaxation information also for filtering. This idea has led to the development of a special kind of global constraints, so-called optimization constraints [4, 15]. Roughly speaking, an optimization constraint expresses our wish to search for improving solutions only while enforcing feasibility for at least one of the constraints of the problem.

In essence, optimization constraints represent a conjunction of a constraint on the objective value and some constraint of the constraint program. Consequently, for many optimization constraints that are highly relevant in practice, achieving generalized arc-consistency turns out to be NP-hard. Due to this observation, weaker notions of consistency have been developed with the aim to get ourselves back into the realm of tractable inference

techniques. Relaxed consistency [2] for example enforces that all values be removed from the variables' domains whose assignment would cause a bound on the objective to drop below some threshold performance (as, for instance, represented by the performance of the currently best known feasible solution to our problem). Under certain conditions it can then be shown that, if the bound under consideration can be computed in polynomial time, achieving relaxed consistency is tractable.

The notion of relaxed consistency allows us to evaluate a filtering algorithm not only with respect to its running time, but also with respect to its filtering effectiveness: the stronger the bound with respect to which we can achieve relaxed consistency, the more effective we can expect filtering to be. In practice, the bounds that are considered often result from relaxations as they were introduced in mathematical programming. As one would expect, linear continuous relaxations play a very important role here [2, 3, 4, 10, 25], but also other methods like Lagrangian relaxation have been considered for filtering purposes [20, 24, 27].

1.2 Relaxed and Approximated Consistency

From a theoretical perspective, the unsatisfactory aspect of mathematical programming relaxations is that they usually do not come with a performance guarantee, i.e. the quality of the bounds is unknown. Obviously, it would be very nice if we could devise and exploit bounds with guaranteed accuracy. In [22, 23], we introduced the concept of approximated consistency which is a refined and stronger notion of relaxed consistency for optimization constraints. In the spirit of relaxed consistency, approximated consistency asks that all assignments are removed from consideration whose commitment would cause a bound with guaranteed accuracy to drop below the threshold.

While the work in [23] on approximated consistency for knapsack constraints was purely theoretical, we were able to show the practical benefits of approximated consistency when tackling the market split problem in [22]. Therefore, approximated consistency is not only a nice theoretical idea, filtering algorithms based on it can actually make a difference in practice as well. However, despite the positive results on market split, in the same paper we also showed that there could be no benefits achieved when using approximated consistency

for knapsack constraints in the context of the automatic recording problem (ARP). The ARP consists in the solution of a knapsack problem where items are associated with time intervals and only items can be selected whose corresponding intervals do not overlap.

1.3 Motivation and Contribution

While the ARP is an interesting problem in and by itself, the combination of a knapsack constraint with non-overlapping time-interval constraints can be identified as a subproblem in many more scheduling problems. For example, satellite scheduling can be viewed as a refinement of the automatic recording problem. Here, the task is to make a selection of given observation requests, whereby each observation takes a specific time interval and requires a certain capacity for storing image data. Another example is the automatic recording of TV content. A selection of programs to be recorded has to be made, whereby there are specific broadcasting times for each program, only one program can be recorded at any given time, and the storage capacity is limited.

Therefore, it is of general interest to study a global constraint that augments the knapsack constraint with time-interval consistency of selected items. This idea gives rise to the Automatic Recording Constraint (ARC), which we want to study in this paper. Obviously, as an augmentation of the knapsack constraint, achieving generalized arc-consistency for the ARC is NP-hard [6]. Consequently, we will develop a filtering algorithm for the constraint that does not guarantee backtrack-free search for the ARP, but that achieves at least approximated consistency with respect to bounds of arbitrary accuracy.

Extending our work published in [19], in Section 2 we review the notions of optimization constraints and approximated consistency as we introduced them in [2, 23]. Then, in Section 3, we develop approximation algorithms for the ARP, which we will exploit in Section 4 to devise a filtering algorithm for the ARC that achieves approximated consistency in quadratic time. Finally, in Section 5, we evaluate the new constraint in the context of automatic recording and show that the new filtering algorithms allow us to solve the ARP orders of magnitude faster than previous approaches based on constraint programming.

2 Approximated Consistency

We start out by reviewing some previously introduced notions. The original definitions can be found in [2, 23, 27]. First, let us define the Automatic Recording Problem and its corresponding constraint formally:

Definition 1 Given $n \in \mathbb{N}$, denote with $V = \{1, \dots, n\}$ the set of items, and with $\text{start}(i) < \text{end}(i) \forall i \in V$ the corresponding starting and ending times. With $w = (w_i)_{1 \leq i \leq n} \in \mathbb{N}_+^n$ we denote the storage requirements, $K \in \mathbb{N}_+$ denotes the storage capacity, and $p = (p_i)_{1 \leq i \leq n} \in \mathbb{N}^n$ the profit vector. Finally, let us define n binary variables $X_1, \dots, X_n \in \{0, 1\}$.

- We say that the interval $I_i := [\text{start}(i), \text{end}(i)]$ corresponds to item $i \in V$, and call two items $i, j \in V$ overlapping whose corresponding intervals overlap, i.e. $I_i \cap I_j \neq \emptyset$. We call $p_X := \sum_{i | X_i=1} p_i$ the user satisfaction (with respect to X).

- The Automatic Recording Problem (ARP) consists in finding an assignment $X = (X_1, \dots, X_n) \in \{0, 1\}^n$ such that

(a) The selection X can be stored within the given disc size, i.e.

$$\sum_i w_i X_i \leq K.$$

(b) at most one item must be selected at a time, i.e.

$$I_i \cap I_j = \emptyset \quad \forall i < j \text{ s.t. } X_i = 1 = X_j.$$

(c) X maximizes the user satisfaction, i.e.

$$p_X \geq p_Y \quad \forall Y \text{ respecting (a) and (b)}.$$

- Analogously, given a lower bound on the objective function $B \in \mathbb{N}$, the Automatic Recording Constraint (ARC) is defined over variables $X_1, \dots, X_n \in \{0, 1\}$ and has parameters w, K, p, B , and I_1, \dots, I_n .

$\text{ARC}(X_1, \dots, X_n, w_1, \dots, w_n, K, p_1, \dots, p_n, B, I_1, \dots, I_n)$ is true iff

(a) $\sum_i w_i X_i \leq K.$

(b) $I_i \cap I_j = \emptyset \quad \forall i \neq j \text{ s.t. } X_i = 1 = X_j.$

(c) $p_X > B.$

In less formal terms, the ARC requires us to find a selection of items such that the total weight limit is not exceeded, no two items overlap in time, and the total objective value is greater than that of the best known feasible solution.

The ARC belongs to the class of maximization constraints which we can use to express our wish to find improving solutions while enforcing feasibility with respect to some constraint of the problem. Formally, we define:

Definition 2 *Given $n \in \mathbb{N}$, let X_1, \dots, X_n denote some variables with finite domains $D_1 := D(X_1), \dots, D_n := D(X_n)$. Furthermore, given a constraint $\zeta : D_1 \times \dots \times D_n \rightarrow \{0, 1\}$, and an objective function $P : D_1 \times \dots \times D_n \rightarrow \mathbb{N}$, let $\bar{x}_i \in D_i \forall 1 \leq i \leq n$.*

- *Let $B \in \mathbb{N}$ denote a lower bound on the objective P to be maximized. Then, a function $\vartheta_{\zeta, P}[B] : D_1 \times \dots \times D_n \rightarrow \{0, 1\}$ with $\vartheta_{\zeta, P}[B](\bar{x}_1, \dots, \bar{x}_n) = 1$ iff $\zeta(\bar{x}_1, \dots, \bar{x}_n) = 1$ and $P(\bar{x}_1, \dots, \bar{x}_n) > B$ is called maximization constraint.*
- *Given a maximization constraint $\vartheta_{\zeta, P}[B]$ and some $\varepsilon \geq 0$, we say that $\vartheta_{\zeta, P}[B]$ is ε -approximate consistent (or ε -GAC), iff for all $1 \leq i \leq n$ and $\bar{x}_i \in D_i$ there exist $\bar{x}_j \in D_j$ for all $j \neq i$ such that $\vartheta_{\zeta, P}[B - \varepsilon P^*](\bar{x}_1, \dots, \bar{x}_n) = 1$, whereby $P^* = \max\{P(y_1, \dots, y_n) \mid y_i \in D_i, \zeta(y_1, \dots, y_n) = 1\}$.*

The second part of the definition specifies what we mean by saying that a maximization constraint like the ARC is approximately consistent. Note that enforcing generalized arc-consistency (GAC) on the ARC is NP-hard, which is easy to see by the fact that finding an improving solution would otherwise be possible in a backtrack-free search [5] or by simple reduction to the knapsack constraint [23].

Approximated consistency therefore only requires that a lower threshold that is diminished by some fraction of the overall best possible performance is guaranteed to be exceeded. At first, it sounds surprising that lowering the threshold allows us to filter efficiently since the constraint has not changed structurally. The reason why approximated consistency can in some cases reduce the worst-case filtering complexity is due to the fact that the filtering algorithm is still allowed to filter values with respect to the original threshold, it is just

not required to do so anymore. In that way, a grey area is created that can smoothen the filtering frontier in such a way that the filtering problem becomes tractable.

The first example for a successful application of the idea of approximated consistency was given in [22, 23] for knapsack constraints. The main problem when designing an approximate filtering algorithm for a maximization constraint is to find or develop an efficient algorithm that computes an upper bound with guaranteed accuracy. One possibility is to exploit existing approximation algorithms. If the optimization problem does not change its structure when an assignment is made (as it is the case for the ARP or the knapsack problem, but not, for instance, for the shortest path problem), then a polynomial k -approximation algorithm allows us to achieve k -consistency in polynomial time.

Following this idea, in our pursuit to develop a filtering algorithm for the ARC, let us first study the ARP and see whether we can develop a fast approximation algorithm for the problem.

3 ARP Approximation

Obviously, even if all items are pairwise non-overlapping (i.e., if restriction (b) in Definition 1 is obsolete), it remains to solve a knapsack problem. Thus, the ARP is NP-hard [6]. Let $p_{max} := \max\{p_i \mid 1 \leq i \leq n\}$. We develop a pseudo-polynomial algorithm running in $\Theta(n^2 p_{max})$ that will be used later to derive a fully polynomial time approximation scheme (FPTAS) for the ARP.

3.1 A Dynamic Programming Algorithm

The algorithm we develop in the following is similar to the teaching book dynamic programming algorithm for knapsack problems. Setting $\bar{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$ and $\psi := np_{max} + 1$, we compute a matrix $M = (m_{kl}) \in \bar{\mathbb{N}}^{n+1 \times \psi}$, $0 \leq k \leq \psi$, $0 \leq l \leq n$. In m_{kl} , we store the minimal knapsack capacity that is needed to achieve a profit greater or equal k using items lower or equal l only ($m_{kl} = \infty$ iff $\sum_{1 \leq i \leq l} p_i < k$).

We assume that V is ordered with respect to increasing ending times, i.e., $1 \leq i < j \leq n$ implies $e_i \leq e_j$. Further, denote with $last_j \in V \cup \{0\}$ the last non-overlapping node lower

than j , i.e.,

$$e_{last_j} < s_j \text{ and } e_i \geq s_j \quad \forall last_j < i \leq j.$$

We set $last_j := 0$ iff no such node exists, i.e., iff $e_0 \geq s_j$. Note that any item with index $i \in \{last_j + 1, \dots, j\}$ overlaps with item j . To simplify the notation, let us assume that $m_{k,0} = \infty$ for all $0 < k < \psi$, and $m_{k,0} = 0$ for all $k \leq 0$. Then,

$$m_{kl} = \min\{m_{k,l-1}, m_{k-p_l, last_l} + w_l\}. \tag{1}$$

The above recursion equation yields a dynamic programming algorithm: First, we sort the items with respect to their ending times and determine $last_i$ for all $0 \leq i < n$. Both can be done in time $\Theta(n \log n)$. Then, we build up the matrix column by column, and within each column from top to bottom. Finally, we compute $\max\{k \mid m_{k,n} \leq K\}$. The total running time of this procedure and the memory needed are obviously in $\Theta(|M|) = \Theta(n^2 p_{max})$.

3.2 A Fully Polynomial Time Approximation Scheme

As for knapsack problems, we can use the dynamic programming algorithm to derive an FPTAS. One way of achieving this goal is to scale the profit vector [12], a method that we exploited in [23, 26]. While scaling works well for knapsack problems, where the linear programming relaxation can easily be exploited to give a 2-approximation based on which we can choose rather large scaling factors, in the context of automatic recording we lack the ability to scale equally effectively. Consequently, for the ARP the scaling method has led to a running time in $O(n^3/\varepsilon)$ which is still the best known approximation scheme for the ARP. A cubic runtime is of course far from being very appealing for practical purposes. Therefore, in the following we develop an approximation scheme that runs in time $O(n^2 \ln(np_{max})/\varepsilon)$ and which is based on one of the oldest ideas to transform a dynamic program into an FPTAS.

The problem with our dynamic program is the fact that it contains too many rows. While profit scaling limits the total number of rows in the matrix, another avenue that we

could follow is to try to limit the total number of non-infinity entries per column and then store each column in a sparse list. This is one of the core ideas in [7]. To achieve sparser columns, when building the matrix column by column, we can interweave trimming steps where we eliminate column entries that achieve almost the same profit than another that is already in the list. If we do not trim too aggressively, then we can bound the error that we make and at the same time make sure that the number of non-infinity weight entries per column remains bounded.

In more formal terms, the procedure works as follows: According to Equation 1, each column depends solely on the column immediately to the left and the column that belongs to the last predecessor of the currently newly added item. We construct new sparse columns as lists of only non-infinity entries that are ordered with increasing profit. This can be done easily by running through the corresponding lists of columns that determine the entries in the new column. After a new column is created, we “trim” it by eliminating entries whose profit (the corresponding row of the matrix entry) is only slightly better than that of another entry in the column. Formally, we remove an entry m_{kl} in the list if and only if there exists a prior and not previously removed entry m_{jl} such that $j \geq (1 - \delta)k$ for some $1 > \delta = \delta(\varepsilon) > 0$. And whenever an entry m_{kl} is removed, its representative entry is set to $m_{jl} := \min\{m_{jl}, m_{kl}\}$. All this can be done in one linear top to bottom pass through the column.

What have we gained by this procedure? After the trimming, successive elements in the list differ by a factor of at least $1/(1 - \delta)$. Thus, each sparse column can contain at most

$$\log_{1/(1-\delta)}(np_{max}) = \frac{\ln(np_{max})}{-\ln(1-\delta)} \leq \frac{\ln(np_{max})}{\delta}$$

elements. Now, because of the trimming of the columns to the left, every new column, before it is trimmed itself, cannot contain more than twice this value. Consequently, the algorithm will only take time $O(\frac{n \ln(np_{max})}{\delta})$.

Now, what error have we introduced by trimming the columns? By induction on the column indices l , it can easily be shown that in the l th column, if there existed an entry m_{kl} in the original dynamic program, then there exists an entry m_{jl} in the trimmed version

such that $(1 - \delta)^l k \leq l \leq k$ and $m_{jl} \leq m_{kl}$. Consequently, the entry $m_{kn} \leq K$ that achieves the optimal profit k has a representative $m_{ln} \leq m_{kl} \leq K$ with $l \geq (1 - \delta)^n k$. When setting $\delta = \varepsilon/n$, then it follows

$$l \geq \left(1 - \frac{\varepsilon}{n}\right)^n k \geq (1 - \varepsilon)k.$$

Consequently, we achieve an FPTAS that runs in time $O\left(\frac{n^2 \ln(np_{max})}{\varepsilon}\right)$.

3.3 Approximation Independent of Large Numbers

On a side note, the theoretical performance can be further enhanced by making it independent of the largest profit. Assume that we scale the profits first by dividing them by $S := \frac{\varepsilon p_{max}}{2n}$ and rounding them down. Then, we know that for the optimal solution to the scaled problem, it holds

$$P_S^* \geq P^* - Sn \geq P^* - \frac{\varepsilon}{2} p_{max} \geq \left(1 - \frac{\varepsilon}{2}\right) P^*,$$

whereby P^* and P_S^* denote the optimal solutions to the original and the scaled problem, respectively, and we assume without loss of generality that $P^* \geq p_{max}$ [23].

Now, instead of solving the scaled problem directly, we approximate it with the help of the routine as defined above, but with the slight modification that we set $\delta := \frac{\varepsilon}{2n}$. The routine then takes time $O\left(\frac{n^2}{\varepsilon} \ln\left(\frac{2n^2}{\varepsilon}\right)\right) = O\left(\frac{n^2}{\varepsilon} \ln\left(\frac{n}{\varepsilon}\right)\right)$, and since the solution P_0 that we provide approximates P_S^* with relative accuracy $\varepsilon/2$, we have

$$P_0 \geq \left(1 - \frac{\varepsilon}{2}\right) P_S^* \geq \left(1 - \frac{\varepsilon}{2}\right)^2 P^* \geq (1 - \varepsilon) P^*,$$

for all $1 > \varepsilon > 0$.

4 Filtering Algorithms for the ARC

In order to achieve a filtering algorithm for the ARC based on the routine that we developed before, we closely follow the idea of defining a directed acyclic graph over the trimmed

dynamic programming matrix. The idea was first introduced in [28] and consequently lead to the filtering algorithms in [23].

4.1 GAC for the ARC

We define the weighted, directed, and acyclic graph for the untrimmed matrix as follows: Every non-infinity entry in the matrix defines a node in the graph. In accordance to Equation 1, each node has at most two incoming arcs: one emanating from the column immediately to the left, and another emanating from the column that corresponds to the last predecessor of the item that is newly added in the current column (whereby the column of the last predecessor may be identical to the column immediately to the left). That is, one incoming arc represents the decision not to use the newly added item (we refer to those arcs as zero-arcs), and the other incoming arc represents the decision to add the new item corresponding to the new column (we refer to those arcs as one-arcs). A zero-arc has associated weight 0, a one-arc has the same weight as the item corresponding to the column the target node belongs to.

Now, in order to express that we are only looking for solutions with profit greater or equal B , we add a sink node t and connect it to the graph by directing arcs to t from exactly those nodes in the last column that have profit greater or equal B (see Figure 1 (a-b)).

With this construction, we ensure a one-to-one correspondence between solutions to the ARP and paths in the graph: A feasible, improving solution corresponds exactly to a path from m_{00} to t that has weight lower or equal K . We call such paths *admissible*. The original numbers in the dynamic programming matrix now correspond to shortest-path distances from $m_{0,0}$ to the individual nodes. Thanks to the fact that our graph is directed and acyclic, we can apply the linear time filtering techniques in [8] that remove those and only those arcs that cannot be visited by any admissible path.

After filtering, every arc in the pruned graph can be part of a path from m_{00} to t with weight lower or equal K . Since arcs really correspond to decisions to include or exclude and item in our solution, there exists a one-arc (zero-arc) to a node in column i iff item i is included (excluded) in some improving, feasible solution. Consequently, by searching the

pruned graph for columns in which no node has incoming one-arcs, we can identify those and only those items that must be excluded in all improving, feasible solutions. The situation is only slightly more complicated when a column has no incoming zero-arcs. We are tempted to conclude that this implies that the item must be taken in all feasible, improving solutions. However, in contrast to knapsack approximation, for the ARC there exist arcs that cross several columns. If there still exists such an arc that can be part of an admissible path, then the items that belong to the columns that are bridged can obviously be excluded in some admissible solution. Consequently, if a column has only incoming one-arcs and no arc crosses the column, then and only then it must indeed be included in all feasible improving solutions. Without going into details, we just note that the detection of items that must be included can be performed in time $O(n \log n + |M|)$. Then, our algorithm achieves generalized arc consistency for the ARC, and the total runtime for this procedure is in $\Theta(|M|) = \Theta(n^2 p_{max})$.

4.2 Approximated Consistency for the ARC

While achieving generalized arc-consistency is appealing in terms of filtering effectiveness, the algorithm clearly suffers from the fact that both its running time and memory requirements are super-polynomial. Using the FPTAS that we developed earlier, we will now rectify these shortcomings.

The question that arises is how we can incorporate the idea of trimming a column. Trimming removes nodes from columns so as to make sure that the number of non-infinity entries stays polynomial in every column. We would like to trim, but we must make sure that by removing nodes we do not eliminate arcs from the graph that could actually belong to admissible paths. Otherwise we may end up filtering values from variable domains that could actually lead to improving, feasible solutions with respect to the ARC, i.e. our filtering algorithm could filter incorrectly, which we must prevent.

The algorithm that we propose uses the trimming idea as follows: In the graph, whenever a column entry would be removed by trimming, we keep the respective node. We add an arc with weight 0 to its dominating representative in the same column and ensure that the

trimmed node has no other outgoing arcs, especially none that target outside of the column that it belongs to. This implies that, for new columns that are generated later, the trimmed node is of no relevance, so that we can still keep the column fill-in under control. With this slight modification of the graph, we can ensure that it has polynomial size and that the filtering method achieves ε -consistency for the ARC (compare with Figure 1 (c)).

Let us formalize the idea by defining the graph corresponding to the trimmed dynamic program as follows:

Definition 3 *We define the weighted, directed, and acyclic graph $G(\delta) = (N, A, v)$ (whereby we always only consider nodes m_{qk} which have a non-infinity value in the dynamic program) by setting:*

- $N_R := \{m_{qk} \mid 0 \leq q \leq np_{max}, 0 \leq k \leq n, m_{qk} \text{ } \delta\text{-untrimmed}\}$.
- $N_T := \{m_{qk} \mid 0 \leq q \leq np_{max}, 0 \leq k \leq n, m_{qk} \text{ } \delta\text{-trimmed}\}$.
- $N := N_R \cup N_T \cup \{t\}$.
- $A_0 := \{(m_{q,k-1}, m_{qk}) \mid k \geq 1, m_{q,k-1} \in N_R, m_{qk} \in N_R \cup N_T\}$.
- $A_1 := \{(m_{q-p_k, last_k}, m_{qk}) \mid k \geq 1, q \geq p_k, m_{q-p_k, last_k} \in N_R, m_{qk} \in N_R \cup N_T\}$.
- $A_R := \{(m_{pk}, m_{qk}) \mid m_{pk} \in N_T, m_{qk} \in N_R, (1 - \delta)p \leq q < p\}$.
- $A_t := \{(m_{qn}, t) \mid q \geq B, m_{qn} \in N_R\}$.
- $A := A_0 \cup A_1 \cup A_R \cup A_t$.
- $v(e) := 0$ for all $e \in A_0 \cup A_R \cup A_t$.
- $v(m_{q-p_k, last_k}, m_{qk}) := w_k$ for all $(m_{q-p_k, last_k}, m_{qk}) \in A_1$.

Remark 1 *For an admissible path $(m_{00}, \dots, m_{pn}, t)$ in the graph, the sequence of arcs in A_0 and A_1 (whereby we ignore arcs in A_R and A_t) determines the corresponding solution to the ARP when we set all items that belong to skipped columns to 0. That corresponding solution*

then has the same weight as the path, and according to Section 3.2 for the corresponding solutions profit q it holds: $(1 - \varepsilon)q \leq p \leq q$.

Theorem 1 Given $1 > \varepsilon > 0$, we set $\delta := \varepsilon/n$.

1. If there exists a path $W = (m_{00}, \dots, m_{pn}, t)$ in $G(0)$ with $p \geq B$ and such that $v(W) \leq K$, then there exists a path $X = (m_{00}, \dots, m_{qn}, t)$ in $G(\delta)$ such that $q \geq (1 - \varepsilon)B$, $v(X) \leq K$, and the corresponding solutions to W and X are identical.
2. If there exists a path $X = (m_{00}, \dots, m_{qn}, t)$ in $G(\delta)$ with $q \geq (1 - \varepsilon)B$ and such that $v(X) \leq K$, then there exists a path $W = (m_{00}, \dots, m_{pn}, t)$ in $G(0)$ such that $p \geq (1 - \varepsilon)B$, $v(W) \leq K$, and the corresponding solutions to W and X are identical.

Proof:

1. We show the correctness of the statement by proving an even stronger result: Let $0 \leq k \leq n$. For a path $W = (m_{00}, \dots, m_{pk})$ in $G(0)$ there exists a path $X = (m_{00}, \dots, m_{qk})$ in $G(\delta)$ such that the weight of both paths and their corresponding solutions are identical and it holds: $q \geq (1 - \delta)^k p$.

We prove this by induction over k : For $k = 0$ we can set $V = (m_{00})$ and we are done. Now, assume that the induction hypothesis holds for all $0 \leq l \leq k$. Given $W = (m_{00}, \dots, m_{rl}, m_{p,k+1})$, we then know that there exists $X' = (m_{00}, \dots, m_{sl})$ such that $s \geq (1 - \delta)^l r \geq (1 - \delta)^k r$. In case that m_{sl} is not trimmed in $G(\delta)$, then we can follow the same decision to include or exclude a item $k + 1$ as it was done to move from m_{rl} to $m_{p,k+1}$ in $G(0)$ and we do not lose any more relative accuracy. On the other hand, if m_{sl} is trimmed, then we extend X' by first moving to the node m_{tl} representing m_{sl} for which it must hold

$$t \geq (1 - \delta)s \geq (1 - \delta)^{k+1}r.$$

Then we take the same decision to include or exclude item $k + 1$ as in W which brings us to some node $m_{q,k+1}$. A simple case distinction on whether item $k + 1$ is included or excluded then shows that the relation on t and r translates to q and p .

Finally, by exploiting the result that we just proved, we know that there exists a path $X = (m_{00}, \dots, m_{q,n}, t)$ in $G(\delta)$ that has the same weight and corresponding solution as W and such that

$$q \geq (1 - \delta)^n p \geq (1 - \varepsilon)p \geq (1 - \varepsilon)B.$$

2. We can define path W in $G(0)$ simply by following the same sequence of arcs in A_0 and A_1 . Then, the weight of W and X are the same while the profit of W is at least that of X . ■

As a direct consequence, by applying shorter-path filtering to the trimmed graph with bound $(1 - \varepsilon)B$ and by applying the same filtering algorithm as described in Section 4.1, according to Theorem 1 (1) we achieve a correct filtering algorithm for the ARC, and according to Theorem 1 (2) we are sure to eliminate all assignments that would cause the best optimal solution to drop below $(1 - \varepsilon)B$. Assuming that B is given as a lower bound on the objective, i.e. $B \leq P^*$, we finally have:

Corollary 1 *Approximated consistency for the ARC can be achieved in time $O(\frac{n^2}{\varepsilon} \ln \frac{n}{\varepsilon})$.*

5 Numerical Results

All experiments in this paper were conducted on an AMD Athlontm 1.2 GHz Processor with 500 MByte RAM running Linux 2.4.10, and we used the GNU C++ compiler version g++ 2.91. For our experiments, we use a benchmark set that we made accessible to the research community in [21]. This benchmark was generated using the same generator that was used for the experimentation in [26, 22]:

The test-instances are generated by random processes. However, the way how the generation is performed is designed to mimic features of real-world instances for the automatic recording of TV content which were established within the EU-funded UP-TV project on ubiquitous and personalized television [29]. Each set of instances is generated by specifying the time horizon (half a day or a full day) and the number of channels (20 or 50). The generator sequentially fills the channels by starting each new item one minute after the last.

For each new item, a *class* is being chosen randomly. That class then determines the interval from which the length is chosen randomly. The generator considers 5 different classes. The lengths of items in the classes vary from 5 ± 2 minutes to 150 ± 50 minutes. The disk space necessary to store each item equals its length, and the storage capacity is randomly chosen as 45%–55% of the entire time horizon.

To achieve a complete instance, it remains to choose the associated profits of items. Four different strategies for the computation of an objective function are available:

- For the *class usefulness (CU)* instances, the associated profit values are determined with respect to the chosen class, where the associated profit values of a class can vary between zero and 600 ± 200 .
- In the *time strongly correlated (TSC)* instances, each 15 minute time interval is assigned a random value between 0 and 10. Then the profit of an item is determined as the sum of all intervals that item has a non-empty intersection with.
- For the *time weakly correlated (TWC)* instances, that value is perturbed by a noise of $\pm 20\%$.
- Finally, in the *strongly correlated (SC)* data, the profit of an item simply equals its length.

We solve the ARP by ordinary depth-first search augmented by ARC-filtering in every choice point. The branching variable is chosen according to a fixed variable ordering that is determined by increasing ending times of the items. Which branch we follow first is determined by the best admissible path in the ARC. The results that we achieve with this simple approach are given in Table 1.

When looking at instances with objective CU, TWC, or TSC, we find that using the new global automatic recording constraint and its approximate filtering algorithm results in run-times that are orders of magnitude better than the previous best CP-approach based on CP-based Lagrangian relaxation [26]. For the TWC and TSC instances, the number of choice points that need to be investigated when using bounds with guaranteed accuracy is greatly reduced. Interestingly, when looking at the results for the day-long CU instances,

we find that the new filtering algorithm actually needs to investigate more choice points, but is still much faster, which implies, of course, that the new filtering algorithm actually performs faster than CP-based Lagrangian relaxation, which reduces the time per choice point needed.

Interestingly, the memory requirements, one of the major drawbacks of approximated consistency for knapsack constraints [22], were well within limits. As a matter of fact, to some extent increasing the accuracy could actually help to reduce the memory requirements since more effective filtering resulted in far smaller search trees. On the other hand, it is easy to calculate that solving the larger problem instances simply by means of dynamic programming (i.e. with no approximation error at all) is not feasible within the given memory limits.

The median values that we report also show that using bounds of greater accuracy helps a lot to reduce the variance in runtime. On the downside, the results on the SC instances are really bad: The SC instances are actually the most easy ones for CP-based Lagrangian relaxation, but even at an accuracy of 0.2% approximated filtering was unable to solve the day-long instances in this class.

6 Conclusions

We introduced a new global constraint, the automatic recording constraint (ARC) that can be used to model and solve problems like satellite scheduling or automatic recording of TV contents where some resource is exhausted linearly while conflicts in time need to be prevented. Since achieving generalized arc-consistency for the ARC is NP-hard, we developed a filtering algorithm that achieves approximated consistency. Numerical results show the benefits of the new constraint on three out of four different types of benchmark sets for the automatic recording problem. For these sets, the run-times could be reduced by orders of magnitude. However, the last set that is particularly easy to solve for CP-based Lagrangian relaxation has caused severe problems to the new approach. This shows that different filtering techniques can perform very differently on problems that can exhibit various structures.

7 Acknowledgements

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

References

- [1] T. Fahle, U. Junker, S.E. Karisch, N. Kohl, M. Sellmann, B. Vaaben. Constraint programming based column generation for crew assignment. *Journal of Heuristics*, 8(1):59-81, 2002.
- [2] T. Fahle and M. Sellmann. Cost-Based Filtering for the Constrained Knapsack Problem. *Annals of Operations Research*, 115:73–93, 2002.
- [3] F. Focacci, A. Lodi, M. Milano. Cutting Planes in Constraint Programming: An Hybrid Approach. *Proceedings of CP-AI-OR'00*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:45–51, 2000.
- [4] F. Focacci, A. Lodi, M. Milano. Cost-Based Domain Filtering. *Principles and Practice of Constraint Programming (CP)* Springer LNCS 1713:189–203, 1999.
- [5] E.C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, 1982.
- [6] M. R. Garey, D. S. Johnson. Computers and Intractability, A Guide to the Theory of NP-Completeness. *Freeman*, San Francisco, 1979.
- [7] O.H. Ibarra and C.E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM*, 22(4):463–468, 1975.
- [8] U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, M. Sellmann. A Framework for Constraint programming based column generation. *Principles and Practice of Constraint Programming (CP)*, Springer LNCS 1713:261–274, 1999.

- [9] L.G. Khachian. A polynomial algorithm in linear programming (in Russian). *Doklady Akad. Nauk SSSR*, 244(5): 1093–1096, 1979.
- [10] H-J. Kim and J. N. Hooker. Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. *Annals of Operations Research* 115:95–124, 2002.
- [11] V. Kumar. Algorithms for Constraints Satisfaction problems: A Survey. *The AI Magazine, by the AAAI*, 13:32-44, 1992.
- [12] E.L. Lawler. Fast Approximation Algorithm for Knapsack Problems. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 206–213, 1977.
- [13] Y. Liu. On the Fully Polynomial Approximation Algorithm for the 0-1 Knapsack Problem. *Theory of Computing Systems*, 35:559-564, 2002.
- [14] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
- [15] M. Milano. *Integration of Mathematical Programming and Constraint Programming for Combinatorial Optimization Problems*, Tutorial at CP2000, 2000.
- [16] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. *12th National Conference on Artificial Intelligence, AAAI*, pp. 362–367, 1994.
- [17] J.-C. Régin. Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, 7(3-4):387–405, 2002.
- [18] S. Sahni. Approximate algorithms for the 0/1 Knapsack Problem. *Journal of the ACM*, 22(1):115–124, 1975.
- [19] M. Sellmann. Approximated Consistency for the Automatic Recording Constraint. *11th intern. Conference on the Principles and Practice of Constraint Programming (CP)*, LNCS:3709:822–826, 2005.

- [20] Meinolf Sellmann. Theoretical Foundations of CP-based Lagrangian Relaxation. *Proceedings of the 10th intern. Conference on the Principles and Practice of Constraint Programming (CP)*, Springer LNCS 3258:634-647, 2004.
- [21] ARP: A Benchmark Set for the Automatic Recording Problem, maintained by M. Sellmann, <http://www.cs.brown.edu/people/sello/arp-benchmark.html>.
- [22] M. Sellmann. The Practice of Approximated Consistency for Knapsack Constraints. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI)*, AAAI Press, pp. 179-184, 2004.
- [23] M. Sellmann. Approximated Consistency for Knapsack Constraints. *CP*, Springer LNCS 2833: 679–693, 2003.
- [24] M. Sellmann. An Arc-Consistency Algorithm for the Weighted All Different Constraint. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:744–749, 2002.
- [25] M. Sellmann. Reduction Techniques in Constraint Programming and Combinatorial Optimization. *PhD Thesis*, University of Paderborn, Germany, <http://www.upb.de/cs/-sello/diss.ps>, 2002.
- [26] M. Sellmann and T. Fahle. Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals of Operations Research*, 118:17-33, 2003.
- [27] M. Sellmann, T.Fahle. Coupling Variable Fixing Algorithms for the Automatic Recording Problem. *Annual European Symposium on Algorithms (ESA)*, Springer LNCS 2161: 134–145, 2001.
- [28] M. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 113–124, 2001.

[29] UP-TV: Ubiquitous Personalized Television. <http://wwwcs.uni-paderborn.de/~pc2/projects/up-tv.html>.

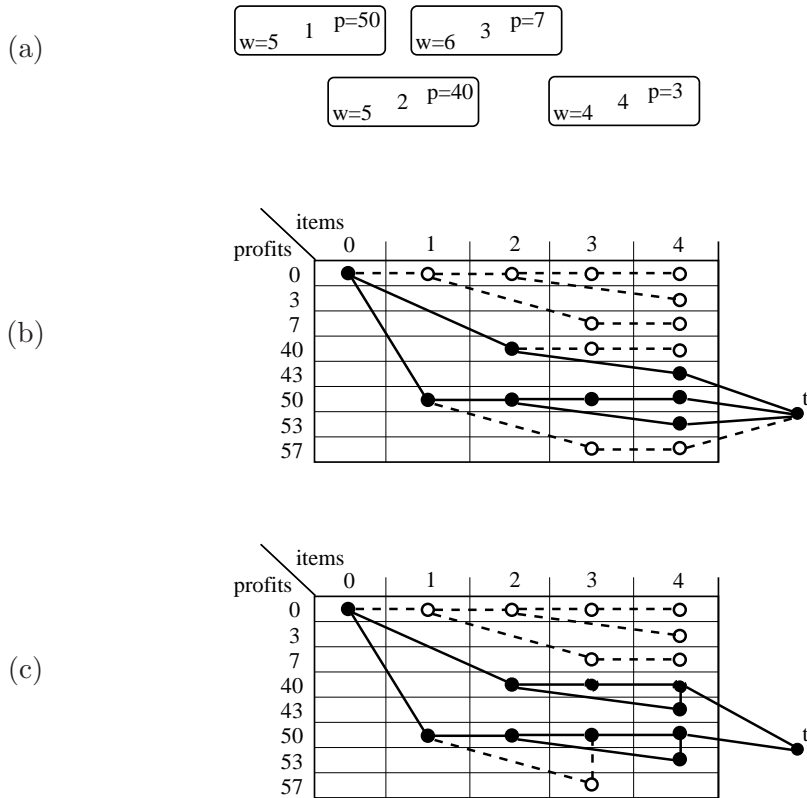


Figure 1: Consider the example as given in Figure (a): We assume that a profit of at least 41 (i.e. $B=41$) must be achieved and we cannot afford more weight than 10 (i.e. $K=10$). Figure (b) sketches the filtering graph over the dynamic programming matrix. Dashed lines and hollow nodes depict parts of the graph that cannot be part of any admissible path as identified by our shorter-path filtering routine. Observe that, in the filtered graph, nodes in column 3 have only incoming zero-arcs, consequently item 3 must be excluded. While in the filtered column 1, nodes only have incoming one-arcs, it cannot be deduced that item 1 must be included, since there exists an admissible arc from column 0 to column 2. In Figure (c) we show the trimmed version when setting $\varepsilon = 1/2$ and $\delta = 1/8$. Note that the filtering bound drops accordingly. Observe how nodes in columns 3 and 4 are trimmed, and how it is important that we keep them in the graph: If we would simply remove the trimmed nodes in column 4, the algorithm would deduce that item 4 must be excluded, which is incorrect.

| obj. | #channels/ horizon | ϕ # items | $\varepsilon = 1\%$ | | $\varepsilon = 0.5\%$ | | $\varepsilon = 0.2\%$ | | CP-based LR | |
|------|-----------------------|-------------------|---------------------|-------------|-----------------------|-------------|-----------------------|-------------|-------------|-------------|
| | | | CPs | Time | CPs | Time | CPs | Time | CPs | Time |
| CU | 20/720 | 305 | 4785 (2160) | .8 (.45) | 90.6 (61) | .122 (.125) | 1 (1) | .115 (.115) | 44.1 (20) | .437 (.16) |
| | 50/720 | 771 | 733K (15K) | 287 (6.73) | 2506 (384) | 1.13 (.465) | 1 (1) | .317 (.305) | 535 (330) | 10.3 (9.17) |
| | 20/1440 | 610 | 52521K (8149K) | 13K (2K) | 103K (72K) | 31.1 (19.0) | 931 (297) | .769 (.63) | 368 (105) | 9.19 (1.84) |
| | 50/1440 | 1512 | - (-) | - (-) | 9030K (1072K) | 6618 (787) | 48K (12K) | 32.4 (8.22) | 35K (2K) | 2452 (156) |
| TWC | 20/720 | 305 | 5 (7) | .11 (.11) | 1.1 (1.5) | .108 (.11) | 1 (1) | .11 (.11) | 45.1 (29) | .296 (.20) |
| | 50/720 | 771 | 11.1 (4) | .313 (.305) | 1.1 (1.5) | .314 (.31) | 1 (1) | .31 (.305) | 469 (107) | 15.8 (1.61) |
| | 20/1440 | 610 | 48.4 (22) | .552 (.54) | 9.8 (4) | .511 (.51) | 1.6 (2) | .508 (.505) | 1318 (67.5) | 12.8 (.82) |
| | 50/1440 | 1512 | 3379 (414) | 6.25 (2.57) | 78.3 (12) | 1.38 (1.35) | 4.4 (8) | 1.33 (1.36) | 1081 (127) | 52.1 (4.14) |
| TSC | 20/720 | 307 | 15.5 (7) | .114 (.115) | 1.6 (2) | .11 (.115) | 1 (1) | .122 (.125) | 108 (64.5) | .801 (.45) |
| | 50/720 | 781 | 43.1 (30) | .332 (.32) | 3.2 (6) | .312 (.31) | 1 (1) | .328 (.33) | 5392 (740) | 103 (20.3) |
| | 20/1440 | 610 | 428 (156) | .816 (.615) | 11.2 (12) | .51 (.51) | 1.6 (2) | .513 (.51) | 1535 (115) | 29.1 (1.94) |
| | 50/1440 | 1511 | 6239 (1220) | 9.27 (3.23) | 125 (68) | 1.44 (1.41) | 1.5 (3.5) | 1.35 (1.37) | 114K (1.3K) | 9K (60.6) |
| SC | 20/720 | 318 | - (-) | - (-) | - (-) | - (-) | 1 (1) | .113 (.11) | 13.7 (14) | 0.109 (.09) |
| | 50/720 | 789 | - (-) | - (-) | - (-) | - (-) | 1 (1) | .322 (.32) | 16.5 (15) | 0.325 (.34) |
| | 20/1440 | 610 | - (-) | - (-) | - (-) | - (-) | - (-) | - (-) | 24.9 (21.5) | .395 (.34) |
| | 50/1440 | 1523 | - (-) | - (-) | - (-) | - (-) | - (-) | - (-) | 27.3 (21) | 1.12 (.995) |

Table 1: Numerical results for the ARP benchmark. We apply our algorithm varying the approximation guarantee of the Knapsack filtering algorithm between 1% and 0.2% and compare against the CP-based Lagrangian relaxation method developed in [26]. The first two columns specify the settings of the random benchmark generator: the first value denotes the objective variant taken, and it is followed by the number of TV channels and the planning time horizon in minutes. Each set identified by these parameters contains 10 instances, the average number of items per set is given in the third column. For each algorithm, we report the average and, in brackets, the median cpu-time in seconds and number of choice points visited. A '-' denotes that a benchmark set of 10 instances could no be solved within 20 hours of cpu-time.