Solving the Temporal Knapsack Problem via Recursive Dantzig-Wolfe Reformulation

Alberto Caprara
University of Bologna, Italy

Fabio Furini
LAMSADE, Université Paris-Dauphine 75775 Paris, France

Enrico Malaguti
DEI, University of Bologna 40136 Bologna, Italy

Emiliano Traversi
LIPN, Université Paris 13 93430 Villetaneuse, France

Abstract
The Temporal Knapsack Problem (TKP) is a generalization of the standard Knapsack Problem where a time horizon is considered, and each item consumes the knapsack capacity during a limited time interval only. In this paper we solve the TKP using what we call a Recursive Dantzig-Wolfe Reformulation (DWR) method. The generic idea of Recursive DWR is to solve a Mixed Integer Program (MIP) by recursively applying DWR, i.e., by using DWR not only for solving the original MIP but also for recursively solving the pricing sub-problems. In a binary case (like the TKP), the Recursive DWR method can be performed in such a way that the only two components needed during the optimization are a Linear Programming solver and an algorithm for solving Knapsack Problems. The Recursive DWR allows us to solve Temporal Knapsack Problem instances through computation of strong dual bounds, which could not be obtained by exploiting the best-known previous approach based on DWR.

© 2011 Published by Elsevier Ltd.

Keywords: Temporal Knapsack Problem, Dantzig-Wolfe Reformulation, Mixed Integer Programs

1. Introduction

In this paper we solve the Temporal Knapsack Problem (TKP) using what we call a Recursive Dantzig-Wolfe Reformulation (DWR) method. The TKP is a generalization of the standard Knapsack Problem where a time horizon is considered and each item consumes the knapsack capacity during a limited time interval only. The knapsack capacity cannot be exceeded at any time instant. We compare the Recursive DWR method to the “natural” formulation of the TKP as a Mixed-Integer Program (MIP) and to the best previous approach based on DWR.

DWR of a MIP is usually tackled through column generation, and it requires solving a number of pricing sub-problems which may be modeled again as MIPs of smaller sizes. In a standard DWR one would tackle the sub-problems by exploiting their special structure, or as simple MIPs through a general purpose solver.
In this work we investigate instead the idea of solving the TKP by recursively applying DWR. We move from the consideration that if DWR is faster than a general purpose solver in solving some specific families of MIPs, as it is the case for the TKP (see Caprara, Furini and Malaguti [1]), then it can be recursively applied in order to achieve incremental computing time reduction. In the DWR of the TKP, the pricing subproblems are TKPs of smaller sizes (see [2]). The Recursive DWR method can be applied until the reformulation results either into single-constraint sub-problems (i.e., “simple” knapsack problems) or into smaller TKP sub-problems (“easier” to solve). Finally, note that not only for the original MIP but also for each intermediate step of the procedure we are interested in integer solutions, and thus the overall method needs to be integrated in a branching scheme, thus resulting in a Branch-and-Price algorithm (B&P).

The TKP is formally defined as follows. A set of \( n \) items is given, the \( i \)-th of which has size \( w_i \), a profit \( p_i \), and is active only during a time interval \( [s_i, t_i] \). A subset of items has to be packed into a knapsack of capacity \( C \) such that the total profit is maximized and the knapsack capacity is not exceeded at any point in time. In order to satisfy the previous requirement, it is enough to impose that a capacity constraint is satisfied at the start time \( s_i \) of each item \( i \). Let \( S_j := \{ i : s_i \leq j \text{ and } j < t_i \} \) denote the set of active items at time \( j \), and \( x_i \) a binary variable equal to 1 if item \( i \) is selected. A binary program for the problem reads:

\[
\text{max} \left\{ \sum_{i=1}^{n} p_i x_i : \sum_{i \in S_j} w_i x_i \leq C, \ j = s_1, \ldots, s_n, \ x \in \{0,1\}, \ i = 1, \ldots, n \right\}. 
\]  

(1)

In the TKP each variable appears with the same positive coefficient in a subset of consecutive constraints only. In addition, the TKP has the property that, by selecting any subset of the constraints, one still gets a TKP. By noting that some of the capacity constraints may be dominated and hence redundant, we define by \( N \subseteq \{1, \ldots, n\} \) as the index subset of the non-dominated constraints (see [3] for further details).

Literature review. We review papers dealing with the TKP and then we contextualize the Recursive DWR comparing it with previous related approaches.

As far as the TKP is concerned, Arkin and Silverberg [4] consider the problem of scheduling jobs with predefined starting and ending times on \( C \) identical machines, by maximizing the value of the completed jobs, where each machine can process one job at a time. This corresponds to a polynomial-time solvable TKP in which all the requests are identical. Chen, Hassin, and Tzur [5] sketch a dynamic programming approach for the TKP. In addition, they propose a polynomial-time approximation algorithm for the special case of the problem in which the profit of an item \( i \) is proportional to \((t_i - s_i)w_i\). Calinescu et al. [6] show a randomized polynomial-time approximation algorithm for the TKP with guarantee arbitrarily close to 2. Bansal et al. show that the restricted case of TKP in which the item weights and the capacity are integers in \([0, L]\) and \( L \leq 2^{\text{poly} \log n} \), has a polynomial-time approximation scheme. The result holds for the generalization of TKP in which the available capacity is not necessarily a constant \( C \) but may vary over time. Darame, Pferschy, and Schauer discuss some generalizations of the problem, review complexity results for special cases of TKP, and prove that the problem remains NP-hard for the case in which all the items’ profits are identical. Finally, the strong NP-hardness of TKP was proved by Bonsma, Schulz, and Wiese [7]. The existence of a polynomial-time approximation scheme remains open. The main computational investigation on the TKP is by Caprara, Furini and Malaguti [8], where the problem is solved by DWR, and the subproblems are associated with groups of constraints. In the paper, the strength of alternative reformulations, obtained by varying the sizes of the groups, are considered.

As far as the idea of recursively applying DWR is concerned, related approaches (denoted as “nested decomposition”) have been considered for linear programs having a special structure. In a so-called multistage linear program, the constraints can be partitioned into an ordered set of stages, in such a way that each variable appears with a non-zero coefficient in the constraints of at most two consecutive stages. Hence, the constraint matrix has a staircase structure and these problems are also known as staircase linear programs. Glasssey [9] and Ho and Manne [10] present nested DW decomposition schemes for multistage linear programs, where each stage operates as a master problem for the following stage, and as a subproblem for the preceding. Local and global optimality conditions and solution procedures are discussed in the papers. In a dual perspective, Witter [11] applies a similar scheme to the dual of a multistage linear program.
Notice instead that the constraint matrix of the TKP does not have a staircase structure as the one defined in [? ? ? ?], since potentially a variable can appear in all the constraints. In the method we propose, at each level of the recursion, we do not have a stage hierarchy in which a stage acts as a subproblem for the preceding one. Instead, all subproblems are independently solved, and the overall coherence for the value of common variables is guaranteed by a master problem (see next section for further details). Finally, we do not restrict to linear programs, but explicitly consider the integrality of the variables, and embed our decompositions in a branching scheme, thus defining a Branch-and-Price algorithm.

2. The Recursive DWR method

We review the main concepts of DWR applied to the TKP following the notation of [? ]. For more details on DWR and column generation we refer the reader to Desaulniers et al. [? ] and Vanderbeck and Wolsey [? ].

Given a subset \( Q \subseteq N \) let:

\[
P_Q := \text{conv} \left\{ x \in [0,1]^{|Q|} : \sum_{i \in S_j} w_i x_i \leq C_j, \ j \in Q \right\}
\]

denote the convex hull of the feasible 0-1 solutions to the capacity constraints in \( Q \). \( V_Q \) the set of vertices of \( P_Q \), and \( y^Q_v \) for \( v \in V_Q \) the coefficients in a convex combination of the vertices in \( V_Q \). One may apply separately this reformulation to any collection \( Q \) of subsets of \( N \). Assuming for simplicity that \( Q \) covers \( N \) (i.e., each \( j \in N \) belongs to some \( Q \in \mathcal{Q} \)), this leads to the overall DWR reformulation of TKP:

\[
\max \left\{ \sum_{i=1}^{n} p_i x_i = \sum_{v \in V_Q} y^Q_v x, Q \in \mathcal{Q}, i \in \bigcup_{Q \in \mathcal{Q}} S_j, \sum_{v \in V_Q} y^Q_v = 1, Q \in \mathcal{Q}, x \in [0,1], i = 1, \ldots, n, y^Q_v \geq 0, Q \in \mathcal{Q}, v \in V_Q \right\}
\]

Since there is an exponential number of \( y^Q_v \) variables for each \( Q \in \mathcal{Q} \) and \( v \in V_Q \), column generation techniques are necessary to solve the linear programming relaxation of the formulation. The variables with a positive reduced profit can be generated by solving a suited TKP with constraints restricted to subset \( Q \) and defining as profits the optimal values of the associated dual variables (see [? ]).

Caprara et al. [? ] considered the DWR for the TKP, and investigated the relation between the strength of the bound of the corresponding linear relaxation and the size of the blocks of constraints. As one may expect, experiments showed that increasing the size of the blocks, that is, reducing the number of blocks, improves the strength of the linear relaxation bound but, at the same time, determines sub-problems which are more difficult to solve. One of the strongest linear relaxation bounds that can be obtained is the one obtained by splitting the constraints into two blocks only (having half of the constraints in the first block and half in the second). In the following, we denote this bound as the two-block bound.

The Recursive DWR solves the linear programming relaxation of the TKP pricing problems by recursively applying the DWR. In order to obtain integer solution at each level of the recursion, a branching scheme is enforced. Notice that the quality of the linear programming bound obtained at the root node of the branching scheme depends uniquely on the initial partition of the constraints. Since at each level of recursion the Recursive DWR method splits the TKP constraints into two more blocks and imposes integrality on the variables, potentially the Recursive DWR method can lead to sub-problems of one constraint only, or the recursion can be stopped after a given number of levels leading to TKP subproblems of multiple constraints.

In general, there are two main design decisions to be taken in order to perform the Recursive DWR method: (i) how to decompose each level, that is, how many blocks, and which constraint goes in each block; (ii) how many levels of recursion to apply before using a MIP solver. These decisions can be represented via a rooted tree, called Sub-problem Tree in the following, where each node as a sub-problem associated with a subset of the constraints of its father node. Hence, the root node represents the original problem and the child nodes of the root node represent the first level of sub-problems, their child nodes represent the second level and so on until the last level of sub-problems is represented by the leaves. In this paper, we consider sub-problem trees that are binary (at each level only two blocks are created) and balanced (each of the two blocks has approximately the same size). At each node, the two sets
of constraints associated with its child nodes represent a complete partition of the father constraint set. Since this scheme is applied at each level of the recursion, at the root node the linear programming relaxation corresponds to the two-block bound. We call level 0 the original problem, the first level of reformulation will consist of two problems with \( \frac{2}{3} \) constraints each, the second level of four problems with \( \frac{2}{4} \), and so on for a maximum of \( \log_2 n = w \) levels, when we have only one constraint per block.

**Sub-problem Tree (SP-T in the following).** A tree structure is then necessary to implement the Recursive DWR and it is used for storing information about the different levels of the decomposition. The number of levels \( w \) to apply and the way of decomposing the problem are both fixed parameters that the algorithm takes as input. The preprocessing phase consists in creating \( \sum_{k=0}^{w-1} 2^k \) “master problems” and \( n \) “leaf” problems, and then initializing them with a feasible solution. The root node contains the original master problem. The two child nodes of the root contain the associated pricing problem, they are again solved via DWR and hence two second-level master problems are associated to them. The same applies to all other levels of the SP-T, except for the last level. If the last level consists of just one constraint then the problem reduces to a KP, otherwise, a MIP solver can be used for solving the pricing problem. Note that in each pricing problem the objective function depends on the dual variables \( \Pi \) of its father node.

The Recursive DWR method can be implemented using two procedures: B&P and DW_BOUND (described in Algorithm ?? and Algorithm ??, respectively) that are recursively calling each other:

- **B&P(UB, IndNode, \( \overline{R} \))**: this recursive depth first search procedure is used for solving via B&P the original problem or any pricing problem. The procedure receives in input the current best upper bound \( UB \), the current node \( \text{IndNode} \) of the SP-T and a vector \( \overline{R} \) of costs (i.e., the original objective function coefficients for the first call and the dual variables values \( \Pi \) except for the first call, in which they correspond to the original objective function coefficients). The procedure returns the optimal solution and the optimal solution value, or \((\emptyset, UB)\) if the current best bound \( UB \) can not be improved. The following additional procedures are used:
  - \( \text{PROB(IndNode)} \): this procedure returns the problem associated to a given node \( \text{IndNode} \).
  - \( \text{INT.CHECK(Sol)} \): this procedure returns \( \text{TRUE} \) if \( \text{Sol} \) is integer, \( \text{FALSE} \) otherwise.
  - \( \text{X_TO_FIX()} \): this procedure returns the variable selected for branching.
  - \( \text{FIX.VAR(IndNode, } X_i, \text{ Value)} \): this procedure fixes to \( \text{Value} \) (i.e., 0 or 1) the variable \( X_i \) on the problem \( \text{PROB(IndNode)} \) and its child nodes in the SP-T.
  - \( \text{FREE.VAR(IndNode, } X_i)} \): this procedure resets lower and upper bounds of the variable \( X_i \) to their original values.

- **DW_BOUND(IndNode, \( \overline{R} \))**: this procedure is used to solve to optimality the LP relaxation of the master problem associated to the node \( \text{IndNode} \) of the SP-T, where \( \overline{R} \) denotes the objective function coefficients associated with \( \text{IndNode} \) (i.e., the dual variables values \( \Pi \) except for the first call, in which they correspond to the original objective function coefficients). It returns the optimal solution and the corresponding value; and it uses the following additional procedures:
  - \( \text{TYPE(IndNode)} \): this procedure is used for checking whether \( \text{IndNode} \) is a LEAF of the SP-T.
  - \( \text{SET.OBJ.COEFS(}\overline{R}, \text{ PROB(IndNode)})) \): this procedure is used for setting the objective function coefficients of \( \text{PROB(IndNode)} \) to \( \overline{R} \).
  - \( \text{SOLVE_LP(PROB(IndNode))} \): this procedure is used for solving the LP relaxation of \( \text{PROB(IndNode)} \). It returns the primal and dual optimal solutions and their value.
  - \( \text{SOLVE_LEAF(PROB(IndNode))} \): this procedure is used for solving the problem \( \text{PROB(IndNode)} \) if \( \text{IndNode} \) is a LEAF, it returns the optimal solution and the optimal solution value for \( \text{PROB(IndNode)} \) (i.e., solving \( \text{PROB(IndNode)} \) as a MIP).
  - \( \text{SP_LEFT(IndNode), SP_RIGHT(IndNode)} \): this procedure is used to retrieve the left and right child node of \( \text{IndNode} \) in the SP-T.

4
The above described implementation is started by calling $B_P(\infty, 0, P)$, where $P$ is the original cost vector $(p_i, i = 1, \ldots, n)$ and 0 stands for the root node of the SP-T. Several nested B&P are then invoked, and, during the solution process, all the sub-problems of the SP-T are adjusted accordingly to the current fixing via the procedure $\text{FIX} \text{VAR}$. Since at each level of the recursion the original variables are explicitly kept in the reformulation, i.e., we use the explicit master format (see [? ] for further details), we can branch on the original variables. This is particularly effective since it preserves the sub-problem structure during the branching scheme. In our implementation, we decided to branch on the variable with the largest fractional value in the current linear programming relaxation. Finally, two computational techniques are used to speed up the convergence of the Recursive DWR: the variables generated at each iteration are stored, in order to potentially be reused at the next pricing iterations; and each master problem is initialized with a feasible solution.

**Algorithm 1: $B_P$**

```plaintext
input : UB, IndNode, R.
output: either one optimal integer solution of PROB(IndNode), or (\emptyset, UB) if no improved solution is found.

// computing current lower bound via column generation
(Sol, Val)← DW_BOUND(IndNode, R);

// bounding
if Val ≥ UB then return (\emptyset, UB);

// updating feasible solution
if INT_CHECK(Sol) then return (Sol, Val);

X_i = X_TO_FIX();
BestIntSol ← \emptyset;
FIX_VAR(IndNode, X_i, 1);

// depth first search, right side
(SolOutBP, ValOutBP)← B_P(UB, IndNode, R);
if SolOutBP ≠ \emptyset then BestIntSol ← SolOutBP; UB ← ValOutBP;
FREE_VAR(IndNode, X_i);

// bounding
if Val ≥ UB then return (BestIntSol, UB);

FIX_VAR(IndNode, X_i, 0);

// depth first search, left side
(SolOutBP, ValOutBP)← B_P(UB, IndNode, R);
if SolOutBP ≠ \emptyset then BestIntSol ← SolOutBP; UB ← ValOutBP;
FREE_VAR(IndNode, X_i);

return (BestIntSol, UB);
```

### 2.1. Recursive DWR for general MIPs

The Recursive DWR method we propose, in principle, could be used to solve generic MIPs. However, this would ask to solve a number of non-trivial design decisions, hence our implementation and experiments are restricted to the TKP, for which the method works very well. This is due to the structure of the TKP, which limits the number of design decisions, as well as to the fact that the problem is well suited for applying DWR.

Specifically, the advantages and drawbacks of the Recursive DWR method are:

- It potentially allows to compute very strong linear programming relaxation bounds, hence, problems may often be solved at the root node of a Branch-and-Price algorithm; for the TKP, the two-block bound is quite difficult to compute (sometimes impossible) within the framework presented in Caprara et al. [? ] while it can be computed by using the Recursive DWR method;
Algorithm 2: DW\_BOUND

\[
\text{input} \quad \text{IndNode}, R \\
\text{output} \quad \text{the optimal solution } \text{Sol} \text{ of the current LP relaxation and the corresponding value } \text{Val}.
\]

\begin{verbatim}
SET_OBJ_COEFS(R, PROB(IndNode))
ColAdd = TRUE;
// adding columns with negative reduced costs
while ColAdd = TRUE do
    ColAdd = FALSE;
    // solving the problem and getting the related information
    (Sol, II, Val) ← SOLVE_LP(PROB(IndNode));
    // solving left sub-problem
    if TYPE(SP\_LEFT(IndNode)) = LEAF then
        (SolLeft, UBLeft) ← SOLVE_LEAF(PROB(SP\_LEFT(IndNode)));
    else
        (SolLeft, UBLeft) ← B\_P(∞, SP\_LEFT(IndNode), II)
    if UBLeft < 0 then add column(s) to PROB(IndNode); ColAdd ← TRUE;
    // solving right sub-problem
    if TYPE(SP\_RIGHT(IndNode)) = LEAF then
        (SolRight, UBRight) ← SOLVE_LEAF(PROB(SP\_RIGHT(IndNode)));
    else
        (SolRight, UBRight) ← B\_P(∞, SP\_RIGHT(IndNode), II)
    if UBRight < 0 then add column(s) to PROB(IndNode); ColAdd ← TRUE;
end
return (Sol, Val);
\end{verbatim}

- it implies a computational overhead for solving not only the original reformulation but also (recursively) the sub-problems through column generation and B&P;
- it has an efficacy strongly related to the capability to define a “good” decomposition for the MIP to be solved (including the subsequent column generation MIPs); this area of research was recently investigated in [? ?];
- aside from the methodology of decomposition, in order to be effective, it requires to carefully determine the depth of the recursion to apply, i.e., when to stop the recursion and solve the resulting problem in a “traditional” fashion.

In summary, for applying the Recursive DWR to other problems, several design issues must be addressed, and we expect the method to be effective for problems with structures similar to the one of the TKP. In the TKP the key element is that constraints matrix is sparse and each variables appears in a limited number of consecutive constraints, determining in this way relatively independent subproblems.

3. Computational Experiments on the Temporal Knapsack Problem

In this section, we show that for the TKP it is possible to deal with large sub-problems by recursively decomposing the sub-problems rather than solving them directly. Indeed, the Recursive DWR allows us to compute the two-block bound, i.e., the linear programming relaxation of the DWR in which the constraints are split into two blocks only. For our experiments we consider some of the hardest odd-numbered TKP instances from Caprara et al. [? ], namely we select instances from the first instance class and groups from V to X. All computational tests were conducted by using 1 core of a core i5-650 at 3.20GHz with 8Gb RAM, under a linux operating system. The general-purpose solver used was IBM ILOG CPLEX 12.3, with the default parameter setting. A recursive depth-first exploration strategy is used and branching is performed on the original xvariables.

In a first set of experiments, we study the impact of the number of levels of recursion on the computational effectiveness of the method when computing the two-block bound by allowing up to 1 hour of computing time. We evaluate the performance first by counting the number of time limits that are reached when computing the two-block bound and secondly by the average computing time (the averages include the time limits of 1 hour). We group together instances with the same number of constraints thus forming 5 homogeneous subsets. In Table ??, we report the number

\[
\text{\begin{tabular}{|c|c|c|c|c|c|}
\hline
\text{Algorithm} & \text{Problems} & \text{Time} & \text{Limits} & \text{Avg. Time} & \text{Time Limit} \\
\hline
\text{DWR} & \text{TKP} & \text{1h} & \text{50} & \text{15m} & \text{1h} \\
\hline
\end{tabular}}
\]
of time limits and the average computing times, obtained by stopping the recursion after a specified number of levels. The table indicates that the number of levels and the computing time are correlated. When the number of levels is low (i.e., up to 5–6) the average computing time decreases, while for larger number of levels the computing time increases. The initial reduction of computing time is due to the fact that the sub-problems become smaller and thus easier to solve while the additional effort of optimizing over a larger number of levels remains low. This trend inverts when the number of levels increases too much and the number of sub-problems becomes too large. From the table it emerges that the best choice for the recursion level depends on the size of the instances, i.e., in order to avoid time limits, 3 levels are necessary for instances with up to 1536 rows, and 4 levels are instead necessary for instances with 1792 rows. To clarify this issue we investigate the size of the leaf sub-problems that are then solved directly with CPLEX.

The previous choice corresponds to stop the recursion when the sub-problems have less than 200 constraints. In Table ?? we report the number of reached time limits according to different maximum leaf size.

In Table ?? we report a comparison between three different approaches to solve the TKP, i.e., the CPLEX MIP solver directly applied to model (??), the algorithm presented in Caprara et al. [? ] (denoted as CFM in the following) and the Recursive DWR method. For a fair comparison, we test CFM and Recursive DWR method with two blocks, obtaining in this way the same root node bound for both methods, i.e., the two-block bound. The value of the obtained bound is then the same in the two cases, but CFM computes it by reformulating the problem into two sub-problems, which are then solved by CPLEX, while the Recursive DWR method iteratively reformulates the sub-problems as well, until subproblems have less than 200 constraints. The first three columns report the instance features, i.e., the name and the total number of variables and constraints. Then we report the time needed by CPLEX to solve the instances to proven optimality. If the time limit of 1 hour is reached, we report the percentage exit gap computed with respect to the best bound and the best heuristic solution value found. For the CFM algorithm, we report the average number of variables (vars<sub>leaf</sub>) and constraints (cons<sub>leaf</sub>) appearing in the final sub-problems (i.e., those solved by CPLEX) and the computing time to solve the root node. This table shows that quite often this method is not able to compute the two-block bound value within a time limit of one hour. Then the table reports the results of the Recursive DWR method; specifically the number of levels, the average number of variables and constraints for each sub-problem and the computing time. Finally the table reports the total computing time needed by the Recursive DWR method to perform the entire Branch-and-Price, i.e., solving the instances to optimality. In bold text, we report the best computing time among the tested algorithms. The Recursive DWR method outperforms the others in all except three cases in which CPLEX applied to the original formulation is the best option. It is worth noticing that in all cases, except two, the Recursive DWR method closes the instances at the root node (i.e., the root node solution is integer), in the remaining two cases the root node gap is less than 0.018%.

Table 1. Impact of the level of recursion on the computing time for the two-block bound.

<table>
<thead>
<tr>
<th>Rows</th>
<th>level of recursion – # time limits</th>
<th>level of recursion – average time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>768</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1280</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1536</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1792</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2. Number of time limits in function of the maximum number of constraints for the leaf subproblems.

<table>
<thead>
<tr>
<th>cons</th>
<th>max leaf size</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>768</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1280</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1536</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1792</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 3. Comparison between the different approaches.

<table>
<thead>
<tr>
<th>Instance</th>
<th>opt</th>
<th>LP gap</th>
<th>time</th>
<th>ROOT gap</th>
<th>time</th>
<th>DW gap</th>
<th>time</th>
<th>vars</th>
<th>#lp</th>
<th>#ilp</th>
</tr>
</thead>
<tbody>
<tr>
<td>I41</td>
<td>30866</td>
<td>14.024</td>
<td>0.0</td>
<td>0.865</td>
<td>1.5</td>
<td>0.000</td>
<td>92.2</td>
<td>142</td>
<td>231</td>
<td>386</td>
</tr>
<tr>
<td>I43</td>
<td>40934</td>
<td>13.502</td>
<td>0.0</td>
<td>1.113</td>
<td>1.8</td>
<td>0.018</td>
<td>978.5</td>
<td>381</td>
<td>669</td>
<td>1024</td>
</tr>
<tr>
<td>I45</td>
<td>50324</td>
<td>13.585</td>
<td>0.0</td>
<td>1.077</td>
<td>3.2</td>
<td>0.000</td>
<td>258.4</td>
<td>108</td>
<td>149</td>
<td>248</td>
</tr>
<tr>
<td>I47</td>
<td>59255</td>
<td>14.040</td>
<td>0.0</td>
<td>0.753</td>
<td>2.9</td>
<td>0.000</td>
<td>303.2</td>
<td>160</td>
<td>242</td>
<td>368</td>
</tr>
<tr>
<td>I49</td>
<td>69530</td>
<td>13.400</td>
<td>0.0</td>
<td>1.077</td>
<td>4.0</td>
<td>0.000</td>
<td>232.21</td>
<td>1039</td>
<td>4479</td>
<td>6754</td>
</tr>
<tr>
<td>I51</td>
<td>71998</td>
<td>11.561</td>
<td>0.0</td>
<td>0.585</td>
<td>2.4</td>
<td>0.000</td>
<td>275.7</td>
<td>151</td>
<td>237</td>
<td>412</td>
</tr>
<tr>
<td>I53</td>
<td>97056</td>
<td>11.877</td>
<td>0.0</td>
<td>0.382</td>
<td>3.1</td>
<td>0.000</td>
<td>897.8</td>
<td>125</td>
<td>385</td>
<td>634</td>
</tr>
<tr>
<td>I55</td>
<td>120505</td>
<td>12.238</td>
<td>0.0</td>
<td>0.502</td>
<td>5.3</td>
<td>0.004</td>
<td>1823.4</td>
<td>415</td>
<td>643</td>
<td>1074</td>
</tr>
<tr>
<td>I57</td>
<td>142486</td>
<td>12.310</td>
<td>0.0</td>
<td>0.541</td>
<td>6.2</td>
<td>0.000</td>
<td>2523.9</td>
<td>211</td>
<td>305</td>
<td>534</td>
</tr>
<tr>
<td>I59</td>
<td>165076</td>
<td>12.245</td>
<td>0.0</td>
<td>0.658</td>
<td>7.7</td>
<td>0.000</td>
<td>1510.7</td>
<td>395</td>
<td>712</td>
<td>1086</td>
</tr>
<tr>
<td>I71</td>
<td>40982</td>
<td>11.322</td>
<td>0.0</td>
<td>0.554</td>
<td>1.3</td>
<td>0.000</td>
<td>118.3</td>
<td>192</td>
<td>225</td>
<td>374</td>
</tr>
<tr>
<td>I73</td>
<td>52447</td>
<td>12.138</td>
<td>0.0</td>
<td>0.612</td>
<td>1.8</td>
<td>0.000</td>
<td>276.9</td>
<td>152</td>
<td>243</td>
<td>404</td>
</tr>
<tr>
<td>I75</td>
<td>66179</td>
<td>12.113</td>
<td>0.0</td>
<td>0.678</td>
<td>2.4</td>
<td>0.000</td>
<td>395.4</td>
<td>104</td>
<td>205</td>
<td>332</td>
</tr>
<tr>
<td>I77</td>
<td>81982</td>
<td>11.320</td>
<td>0.0</td>
<td>0.575</td>
<td>3.2</td>
<td>0.000</td>
<td>949.0</td>
<td>177</td>
<td>327</td>
<td>536</td>
</tr>
<tr>
<td>I79</td>
<td>95037</td>
<td>11.599</td>
<td>0.0</td>
<td>0.562</td>
<td>3.9</td>
<td>0.000</td>
<td>1056.8</td>
<td>807</td>
<td>1992</td>
<td>2894</td>
</tr>
<tr>
<td>I81</td>
<td>71426</td>
<td>11.687</td>
<td>0.0</td>
<td>0.483</td>
<td>2.3</td>
<td>0.000</td>
<td>253.6</td>
<td>119</td>
<td>213</td>
<td>308</td>
</tr>
<tr>
<td>I83</td>
<td>96115</td>
<td>10.845</td>
<td>0.0</td>
<td>0.314</td>
<td>2.5</td>
<td>0.000</td>
<td>948.4</td>
<td>154</td>
<td>438</td>
<td>604</td>
</tr>
<tr>
<td>I85</td>
<td>119233</td>
<td>11.549</td>
<td>0.0</td>
<td>0.324</td>
<td>4.0</td>
<td>0.000</td>
<td>784.8</td>
<td>183</td>
<td>233</td>
<td>388</td>
</tr>
<tr>
<td>I87</td>
<td>142056</td>
<td>10.683</td>
<td>0.0</td>
<td>0.479</td>
<td>5.2</td>
<td>0.000</td>
<td>961.7</td>
<td>148</td>
<td>215</td>
<td>334</td>
</tr>
<tr>
<td>I89</td>
<td>167916</td>
<td>10.753</td>
<td>0.0</td>
<td>0.423</td>
<td>7.2</td>
<td>0.000</td>
<td>1221.5</td>
<td>322</td>
<td>709</td>
<td>970</td>
</tr>
<tr>
<td>I91</td>
<td>42685</td>
<td>11.950</td>
<td>0.0</td>
<td>0.856</td>
<td>1.7</td>
<td>0.000</td>
<td>172.7</td>
<td>139</td>
<td>212</td>
<td>352</td>
</tr>
<tr>
<td>I93</td>
<td>54437</td>
<td>11.538</td>
<td>0.0</td>
<td>0.666</td>
<td>1.9</td>
<td>0.000</td>
<td>390.2</td>
<td>195</td>
<td>331</td>
<td>498</td>
</tr>
<tr>
<td>I95</td>
<td>68432</td>
<td>10.958</td>
<td>0.0</td>
<td>0.676</td>
<td>2.8</td>
<td>0.000</td>
<td>1161.6</td>
<td>331</td>
<td>478</td>
<td>742</td>
</tr>
<tr>
<td>I97</td>
<td>80122</td>
<td>11.823</td>
<td>0.0</td>
<td>0.640</td>
<td>3.5</td>
<td>0.000</td>
<td>2869.5</td>
<td>650</td>
<td>1231</td>
<td>1852</td>
</tr>
<tr>
<td>I99</td>
<td>92380</td>
<td>12.287</td>
<td>0.0</td>
<td>0.800</td>
<td>5.5</td>
<td>0.000</td>
<td>2424.1</td>
<td>880</td>
<td>2106</td>
<td>2894</td>
</tr>
</tbody>
</table>

Table 4. Additional computational details.
Table 5. Comparison between CPLEX and the Recursive DWR method used as a complete MIP solver.

<table>
<thead>
<tr>
<th>Instance</th>
<th>var</th>
<th>cons</th>
<th>time</th>
<th>levels</th>
<th>varr</th>
<th>consr</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1-32</td>
<td>115</td>
<td>32</td>
<td>5</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>119.1</td>
</tr>
<tr>
<td>I2-32</td>
<td>217</td>
<td>32</td>
<td>0.1</td>
<td>5</td>
<td>30</td>
<td>1</td>
<td>118.3</td>
</tr>
<tr>
<td>I3-32</td>
<td>138</td>
<td>32</td>
<td>0.1</td>
<td>5</td>
<td>35</td>
<td>1</td>
<td>44.4</td>
</tr>
<tr>
<td>I4-32</td>
<td>238</td>
<td>32</td>
<td>0.1</td>
<td>5</td>
<td>30</td>
<td>1</td>
<td>119.1</td>
</tr>
<tr>
<td>I5-32</td>
<td>153</td>
<td>32</td>
<td>0.1</td>
<td>5</td>
<td>40</td>
<td>1</td>
<td>141.6</td>
</tr>
<tr>
<td>I1-64</td>
<td>205</td>
<td>64</td>
<td>0.2</td>
<td>6</td>
<td>30</td>
<td>1</td>
<td>1315.0</td>
</tr>
<tr>
<td>I2-64</td>
<td>418</td>
<td>64</td>
<td>0.1</td>
<td>6</td>
<td>30</td>
<td>1</td>
<td>528.7</td>
</tr>
<tr>
<td>I3-64</td>
<td>258</td>
<td>64</td>
<td>0.9</td>
<td>6</td>
<td>35</td>
<td>1</td>
<td>TL</td>
</tr>
<tr>
<td>I4-64</td>
<td>454</td>
<td>64</td>
<td>0.7</td>
<td>6</td>
<td>35</td>
<td>1</td>
<td>552.1</td>
</tr>
<tr>
<td>I5-64</td>
<td>288</td>
<td>64</td>
<td>0.1</td>
<td>6</td>
<td>40</td>
<td>1</td>
<td>2631.0</td>
</tr>
<tr>
<td>I1-128</td>
<td>374</td>
<td>128</td>
<td>0.3</td>
<td>7</td>
<td>30</td>
<td>1</td>
<td>TL</td>
</tr>
<tr>
<td>I2-128</td>
<td>830</td>
<td>128</td>
<td>1.6</td>
<td>7</td>
<td>30</td>
<td>1</td>
<td>TL</td>
</tr>
<tr>
<td>I3-128</td>
<td>492</td>
<td>128</td>
<td>3.3</td>
<td>7</td>
<td>35</td>
<td>1</td>
<td>TL</td>
</tr>
<tr>
<td>I4-128</td>
<td>886</td>
<td>128</td>
<td>8.6</td>
<td>7</td>
<td>35</td>
<td>1</td>
<td>1621.8</td>
</tr>
<tr>
<td>I5-128</td>
<td>531</td>
<td>128</td>
<td>2.3</td>
<td>7</td>
<td>40</td>
<td>1</td>
<td>TL</td>
</tr>
</tbody>
</table>

In Table ??, we report additional computational details for the instances considered. The first two columns report the instance name and its optimal solution value. Then we present three percentage gaps, the first two gaps (LP gap and ROOT gap) are the linear programming relaxation gap, and the root node relaxation (linear programming relaxation plus default cuts generated at the root node), respectively, obtained when tackling Model (??) directly with CPLEX. The third gap (DW gap) is obtained by the two-block bound, computed by the Recursive DWR method. All the gaps have been computed using the following formula: (opt-bound)/opt*100, where opt is the optimal value of the instance. We report beside each gap the corresponding computing time. From the table it clearly emerges that LP gap corresponds to a fast relaxation of “poor” quality while ROOT gap denotes a better relaxation which requires “some” additional computational effort. DW gap is by far the smallest one, the computational time to obtain the associated bound is the largest but the quality is sufficient to prove optimality in all except two instances. In the last part of the table we report some further details of the Recursive DWR method. We report the total number of variables generated during the optimization (vars). Finally the table reports the number of linear programming relaxations solved (#lp), i.e., the number of times a master problem has been solved and the number of integer programming problems solved (#ilp), i.e., the number of times a leaf problem has been solved. We report in bold text the instances solved at the root node by the Recursive DWR method. In summary, the Recursive DWR method has been able to outperform both CPLEX and CFM, computationally proving that a suitable number of reformulation levels helps in better solving the instances of TKP.

4. Conclusions

In this study we investigated the possibility to solve the Temporal Knapsack Problem by applying the DWR recursively to solve the column generation sub-problems as well, in what we defined a Recursive DWR method. For the TKP we showed that the Recursive DWR method can be effective in solving hard instances that could not be solved by state-of-the-art approaches or MIP solvers applied to the original formulation. The TKP belongs to a class of structured MIPs called dynamic MIP. Dynamic MIPs are an extension of classical MIPs by a time horizon and a
time interval in which each variable is active (see Bergner et al. [? ? ]). Since the Recursive DWR method has been effective on the TKP, other problems belonging to the same class of dynamic MIPs are natural candidates for further applications of the method in future works. Finally, the effectiveness of the Recursive DWR method deeply relies on the quality of the decomposition. We are confident that advances in the research on the topic will make the Recursive DWR method applicable to broader categories of MIPs.

Acknowledgments. To the great sadness of the other authors, Alberto Caprara prematurely passed away while working on this problem and after contributing, as it often happened, some powerful insights on its structure. Alberto will be missed by his coauthors of the present paper as an amazing scientist and great friend. The authors thank two anonymous referees for their careful reading and insightful comments.

References