Dynamic programming approaches for the temporal knapsack problem

François Clautiaux, Boris Detienne, Gaël Guillot

Dynamic programming approaches for the temporal knapsack problem

François Clautiaux, Boris Detienne, Gaël Guillot
Université de Bordeaux, UMR CNRS 5251, Inria Bordeaux Sud-Ouest

Submitted for publication on February 8th, 2019

Abstract

In this paper, we address a problem called temporal knapsack problem. In this generalization of the classical knapsack problem, selected items enter and leave the knapsack at fixed dates. We model this problem as an exponential size dynamic program, which is solved using a method called *Successive Sublimation Dynamic Programming* (SSDP), proposed by Ibaraki. This method starts by relaxing a set of constraints from the initial problem, and iteratively reintroduces them when needed.

We show that a direct application of SSDP to the temporal knapsack problem does not lead to an efficient method. Several techniques are developed to solve difficult instances from the literature: detecting unnecessary states at early stages, choosing the right dimensions to use, partial enumeration in the dynamic program, among others. Using these different refinements, our method is able to improve the best results from the literature on classical benchmarks.

1 Introduction

In this paper, we solve the Temporal Knapsack Problem (TKP) using an iterative approach based on dynamic programming. TKP is a generalization of the well-known knapsack problem, where the knapsack is considered along a time period, and items are added to the knapsack only during a given time interval, which is different for each item, and defined in the data. Formally, the problem can be stated as follows.

Problem 1 (Temporal Knapsack Problem)**.** *Let $\mathcal{I} = \{1, \dots, n\}$ be a set of items. Each item has a profit $p_i \in \mathbb{R}^+$, a size $w_i \in \mathbb{Z}^+$, and time interval $[t_i^-, t_i^+)$, where $t_i^-, t_i^+ \in \mathbb{Z}^+$ and $t_i^- < t_i^+$. Let also $W \in \mathbb{Z}^+$ be the size of the knapsack.*

A feasible selection of items is a subset $\mathcal{J}$ of $\mathcal{I}$ such that for any value of $t \in \mathbb{Z}^+$, the sum of the sizes of the items in $\mathcal{J}$ whose time interval contains t is less than W.

The Temporal Knapsack Problem *consists in selecting a feasible selection of $\mathcal{I}$ whose sum of profits is maximum.*

In its general version, this problem is NP-hard in the strong sense ([BSW14]). The first results proposed for this problem were mostly theoretical: a polynomially solvable version of the problem was studied in [AS87], and approximation results were proposed in [CCKR02]. Subsequent works focus on exact methods. The most recent are based on branch-and-price algorithms: the method originally proposed in [CFM13] and improved in [CFMT16] exploits the fact that at a given time period, only a subset of items may belong to the knapsack. This calls for a decomposition method by time periods, where the column generation subproblem is a simple knapsack problem, whereas the master handles consistency between the contents of the bin during consecutive time periods. The authors study the trade-off between the size of the master and subproblem programs size. They show that it is more efficient to consider longer time periods, and solve the larger subproblem with a general purpose MIP solver. These results were improved in [GI14] by adding stabilization techniques to improve the branch-and-price method. To our knowledge, altough a sketch of dynamic program was proposed in [CHT02], no practical method based on dynamic programming have been proposed so far.

In this manuscript, we propose a new exact method for solving the problem. Our method is based on a pseudo-polynomial and exponential size dynamic program, which is solved using so-called Successive Sublimation Dynamic Programming (SSDP) method, originally proposed in [Iba87].

SSDP consists in solving a relaxation of the original dynamic program, fixing some variables, and reintroducing incrementally the relaxed constraints, until an optimality proof is reached. The effectiveness of the method is highly dependent on the capability to reuse information from the previous steps

in the current one (primal and dual bounds, variable fixing). This technique has been shown to be efficient for solving hard combinatorial optimization problem, mostly in the scheduling field (see *e.g.* [TFA09]). Recently this technique has also been generalized to hypergraphs by [CSVV18] for solving a two-dimensional knapsack problem.

Obtaining an efficient specialization of SSDP for solving TKP is not straightforward. We show numerically that a basic application of this technique to TKP is not competitive compared to state-of-the-art solvers. However several advanced algorithmic techniques allow a significant improvement on the computational results. The most important features of our algorithm are: a good choice of the constraints to reintroduce at each step of the algorithm, partial enumeration, constraint propagation rules, and feasibility tests in relaxations based on some pre-computed partial solutions.

We implemented our algorithms and compared them empirically against the state-of-the-art TKP solver developped by [GI14], using classical instances for this problem. We show that our method is able to solve significantly more instances than the literature.

In section 2, we introduce the problem formally, and state some basic MIP and recursive formulations. In section 3, we describe an application of SSDP to TKP. Section 4 describes the various refinements of the method that were necessary to obtain competitive results. We report our computational experiments in Section 5 before offering some brief concluding remarks and suggestions for future research in the conclusion.

2 Integer programming and dynamic programming models

In this section, we recall a basic yet effective MIP formulation for the problem. We also propose a new recursive formulation, which is used to introduce the exponentially large dynamic program on which all methods proposed in this manuscript are based.

2.1 Integer programming formulation

We now recall the classical integer programming formulation (see *e.g.* [CFM13, CFMT16]) for TKP. In this model, each binary variable x_i is equal to one if item i is selected, zero otherwise, similarly to the classical knapsack problem.

$$\max \sum_{i \in \mathcal{I}} p_i x_i \tag{1}$$

$$\text{s.t.} \sum_{i \in \mathcal{I}: t_i^- \leq t < t_i^+} w_i x_i \leq W, \quad t = 1, \ldots, \max_{i \in \mathcal{I}} t_i^+ \tag{2}$$

$$x_i \in \{0,1\}, \quad i \in \mathcal{I} \tag{3}$$

Constraints (2) state that for any time period t, items that belong to the knapsack during t satisfy the capacity constraint. This model can be reduced to a polynomial-size integer linear program by observing that some of constraints (2) are not necessary for the validity of the model. Indeed, consider consecutive time slots $t, t+1, \ldots, t'$ during which no item time window ends. Then Constraint (2) at rank t' implies those at ranks $t, t+1, \ldots, t'-1$, which are linearly dominated. The non-dominated constraints actually correspond with all the maximal subsets of items with overlapping time windows, that can be computed using basic graph concepts. Consider graph $G^{\text{int}} = (\mathcal{I}, E^{\text{int}})$ where $(i,j) \in E^{\text{int}}$ if $[t_i^-, t_i^+) \cap [t_j^-, t_j^+) \neq \emptyset$. The maximal subsets of items sought are the maximal cliques in G^{int}. By construction, G^{int} is an interval graph, thus the number of maximal cliques in G^{int} is not larger than n, and can be computed in a time that is linear in $|E^{\text{int}}|$ (see [Fra76]). Let $\mathcal{Q}$ be the set of maximal cliques $Q_1, \ldots, Q_m$ of G^{int}. The simplified model can be written as follows.

$$\max \sum_{i \in \mathcal{I}} p_i x_i \tag{4}$$

$$\text{s.t.} \sum_{i \in Q} w_i x_i \leq W, \quad Q \in \mathcal{Q} \tag{5}$$

$$x_i \in \{0,1\}, \quad i \in \mathcal{I} \tag{6}$$

This does not change the combinatorial difficulty of the instance, but makes some dominances appear more clearly. Note that modern MIP solvers would turn automatically model (1)–(3) into (4)–(6) based on straightforward linear dominance.

2.2 A dynamic programming formulation

Before describing our dynamic program, let us propose an alternative MIP formulation for TKP. It will be useful to assess the quality of some relaxations for the dynamic program, and is also used to warm-start our optimization methods. In this model, we see the problem as a succession of events (an item may enter the knapsack, or an item may leave the knapsack) where a decision has to be taken (actually adding the item, or actually removing the item).

Let $\mathcal{E} = (e_1, \ldots, e_{2n})$ be an ordered list of indices of so-called *events*. Each event e is related to an item $i(e) \in \mathcal{I}$ and a type $r(e) \in \{\text{in}, \text{out}\}$. Let $\mathcal{E}^{\text{in}} = \{e \in \mathcal{E} : r(e) = \text{in}\}$ and $\mathcal{E}^{\text{out}} = \{e \in \mathcal{E} : r(e) = \text{out}\}$ be respectively the events related to entering and exiting items. Let also $\hat{t}(e)$ be the time slot related to event e, *i.e.* respectively $t^+_{i(e)}$ if $e \in \mathcal{E}^{\text{out}}$ and $t^-_{i(e)}$ if $e \in \mathcal{E}^{\text{in}}$. Indices e in $\mathcal{E}$ are ordered as follows: $e \prec e'$ if $\hat{t}(e) < \hat{t}(e')$ or $(\hat{t}(e) = \hat{t}(e') \wedge e \in \mathcal{E}^{\text{out}} \wedge e' \in \mathcal{E}^{\text{in}})$ (ties are broken arbitrarily). Let also $2n+1$ be the index of an additional dummy event whose time slot is greater than that of any event. Finally let $\mathcal{I}(e) = \{i \in \mathcal{I} : t^-_i \leq \hat{t}(e) < t^+_i\}$ be the set of items that may belong to the knapsack when e occurs.

The new MIP model has one binary, and one real variable per event. For each event e, we define a binary variable y_e that indicates whether the action related to event e is performed or not. If $e \in \mathcal{E}^{\text{in}}$, this decision corresponds with adding e to the current solution. If $e \in \mathcal{E}^{\text{out}}$, the decision corresponds with removing e from the knapsack. In a valid solution, an item leaves the knapsack if and only if it enters the knapsack in a previous event. Each variable ϕ_e $(e = 1, \ldots, 2n)$ is equal to the resource consumption at the end of event e.

$$\max \sum_{e \in \mathcal{E}} \frac{1}{2} p_{i(e)} y_e \tag{7}$$

$$\phi_1 = w_{i(1)} y_1 \tag{8}$$

$$\phi_e = \phi_{e-1} + w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{in}} \setminus \{1\} \tag{9}$$

$$\phi_e = \phi_{e-1} - w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{out}} \tag{10}$$

$$\phi_e \leq W \quad e = 1, \ldots, 2n \tag{11}$$

$$\phi_{2n} = 0 \tag{12}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e') \tag{13}$$

$$y_e \in \{0, 1\}, \quad e = 1, \ldots, 2n \tag{14}$$

$$\phi_e \in \mathbb{R}_+, \quad e = 1, \ldots, 2n \tag{15}$$

The objective function is similar to that of model (4)–(6). The only difference is that the profit is split between the two events related to each item. Note that the repartition of profit in two equal parts is arbitrary, and any pair of real values whose sum is $p_{i(e)}$ would be valid. Constraints (8)–(10) ensure that the capacity consumption at the end of each event is consistent with the contents of the knapsack. Constraints (11) and (12) guarantee that the capacity constraints are satisfied. Constraints (13) state that if an item enters the knapsack, it has to leave it. We call constraints (13) *consistency constraints*. Note that constraint (12) is redundant when no other constraint is relaxed.

We now present our dynamic program, which is also based on the concept of event. The model works similarly to model (7)–(15) in the sense that the current capacity is updated event by event recursively, one has to ensure that the capacity constraint remains satisfied, and decisions related to items are consistent. Let $\mathbf{d} \in \{0, 1\}^n$ be a binary vector indicating which items are in the knapsack. Let also $\boldsymbol{\varepsilon}_k \in \{0, 1\}^n$ be the vector whose components are all zero except component k, which is equal to 1. When two vectors are considered, the addition and substraction symbols $+$ and $-$ respectively stand for the component-wise addition and substraction.

Each state of the dynamic program is a triple $(e, w, \mathbf{d})$, where e is the current event, w the current resource consumption, and $\mathbf{d}$ defined as above. The Bellman function for each state is computed as

follows.

$$\alpha(e,w,\mathbf{d}) = \begin{cases} \max\left\{\alpha(e+1,w,\mathbf{d}), \frac{1}{2}p_{i(e)} + \alpha\left(e+1, w+w_i, \mathbf{d}+\boldsymbol{\varepsilon}_{i(e)}\right)\right\} & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_i \leq W \\ \alpha(e+1,w,\mathbf{d}) & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_i > W \\ \frac{1}{2}p_{i(e)} + \alpha\left(e+1, w-w_i, \mathbf{d}-\boldsymbol{\varepsilon}_{i(e)}\right) & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 1 \\ \alpha(e+1,w,\mathbf{d}) & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 0 \\ 0 & \text{if } e = 2n+1 \end{cases} \tag{16}$$

When an *in* event is considered, two choices are possible: selecting the item or not (the former can be made only if the remaining capacity is large enough). When an *out* event is considered, only one choice is possible, depending on value $\mathbf{d}_{i(e)}$. The optimal value of the TKP is $\alpha(1,0,\mathbf{0})$, where $\mathbf{0}$ is the null vector of dimension n. Note that value w is redundant in this formulation, since it can be deduced from vector $\mathbf{d}$.

3 Applying Successive Sublimation Dynamic Programming to TKP

Clearly solving directly dynamic program (16) cannot lead to any practical method. To obtain an efficient algorithm, we use a technique called *Successive Sublimation Dynamic Programming* (SSDP in the remainder), introduced in [Iba87]. In this section, we explain how SSDP can be adapted to solve TKP. We first describe the generic algorithm, emphasizing the main points to be studied, namely choosing a relaxation, solving the relaxation, and updating the relaxation to obtain a refined model. We then address each point specifically.

3.1 Presentation of the generic algorithm

SSDP is a dual method that iteratively solves relaxations of a dynamic program. The dual bound is improved by refining the relaxation, until the duality gap to a known primal bound is closed. This approach defines the relaxations by projecting the state space onto smaller subspaces. The bound obtained is possibly reinforced using a Lagrangian relaxation of the constraints discarded by the projection of the state-space. To contain the combinatorial explosion and try to avoid the curse of dimensionality, some unnecessary states and transitions are identified and removed from subsequently built relaxations.

Algorithm 1: SSDP

1. **Initialization**. Construct an initial relaxation ;
2. **Solving the relaxation**. Solve optimally the relaxation using dynamic programming. If the optimal solution of the relaxation is feasible and has a cost equal to some primal bound, STOP ;
3. **Filtering**. Remove non-optimal states and transitions ;
4. **Sublimation**. Construct a new relaxation and go back to *step 2*.

SSDP is a generic method which applies to many combinatorial problems. When it is applied to a new problem, several ad-hoc key ingredients have to be designed. The most important are the set of relaxed constraints, and the type of relaxation used. Another major ingredient is the algorithm used to solve each relaxed problem, and its capability to eliminate unfeasible/non-optimal partial solutions. Finally, an effective method to update the relaxation at each step is mandatory.

3.2 Relaxation used for TKP

The size of the state space in (16) is exponential according to the size of $\mathbf{d}$: the size of the binary vector $\mathbf{d}$ is n, so the state space size is in $O(n \times W \times 2^n)$. Our relaxation consists in not keeping track of some items, and thus considering a smaller vector $\mathbf{d}$. In this case, it may happen that an item enters the knapsack and does not leave it, or the opposite. This is stated in the following observation.

Observation 1. *Projecting out vector* $\mathbf{d}$ *in* (16) *is equivalent to relaxing consistency constraints* (13) *in* (7)– (15).

A combinatorial relaxation of Constraints (13) would lead to upper bounds of poor quality. We use Lagrangian relaxation instead, which offers stronger bounds, and is in our case still computationally

attractive. At a given iteration of the algorithm, the relaxation is based on the set $\mathcal{J}$ of items that have to satisfy (13) (*i.e.* whose presence will be accounted in vector $\mathbf{d}$). Let $\boldsymbol{\pi} \in \mathbb{R}^n$ be the vector of Lagrangian multipliers associated with Constraints (13) at ranks $\mathcal{I} \setminus \mathcal{J}$. To simplify the notation, we assume that $\boldsymbol{\pi}$ and $\mathbf{d}$ are always of size n. Within this setting, for a given set $\mathcal{J}$, and a given vector of multipliers $\boldsymbol{\pi}$, the Lagrangian dual function can be cast as:

$$L_{\mathcal{J}}(\boldsymbol{\pi}) = \max \sum_{e \in \mathcal{E}^{in}} \left(\frac{1}{2}p_{i(e)} + \boldsymbol{\pi}_{i(e)}\right) y_e + \sum_{e \in \mathcal{E}^{out}} \left(\frac{1}{2}p_{i(e)} - \boldsymbol{\pi}_{i(e)}\right) y_e \tag{17}$$

$$(9) - (12), (14), (15) \tag{18}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e'), i(e) \in \mathcal{J} \tag{19}$$

For a set $\mathcal{J}$ and $\boldsymbol{\pi} \in \mathbb{R}^n$, the value of $L_{\mathcal{J}}(\boldsymbol{\pi})$ is an upper bound on the optimum of (7)-(15) and can be computed with the help of the following dynamic program. The decisions are the same as before, only the cost changes to account for Lagrangian multipliers, and two choices are possible for *out* events related to untracked items. To get rid of many subcases, we consider that $\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d}) = -\infty$ if $w > W$ or $w < 0$. The new recursion is as follows.

$$\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d}) = \begin{cases} \max\left\{\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w, \mathbf{d}), \frac{1}{2}p_{i(e)} + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w + w_{i(e)}, \mathbf{d} + \boldsymbol{\varepsilon}_{i(e)})\right\} & \text{if } e \in \mathcal{E}^{\text{in}},\ i(e) \in \mathcal{J} \\ \max\left\{\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w, \mathbf{d}), \frac{1}{2}p_{i(e)} + \boldsymbol{\pi}_{i(e)} + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w + w_i(e), \mathbf{d})\right\} & \text{if } e \in \mathcal{E}^{\text{in}},\ i(e) \notin \mathcal{J} \\ \frac{1}{2}p_{i(e)} + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w - w_{i(e)}, \mathbf{d} - \boldsymbol{\varepsilon}_{i(e)}) & \text{if } e \in \mathcal{E}^{\text{out}} \wedge i(e) \in \mathcal{J} \wedge \mathbf{d}_{i(e)} = 1 \\ \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w, \mathbf{d}) & \text{if } e \in \mathcal{E}^{\text{out}} \wedge i(e) \in \mathcal{J} \wedge \mathbf{d}_{i(e)} = 0 \\ \max\left\{\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w, \mathbf{d}), \frac{1}{2}p_{i(e)} - \boldsymbol{\pi}_{i(e)} + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w - w_{i(e)}, \mathbf{d})\right\} & \text{if } e \in \mathcal{E}^{\text{out}} \wedge i(e) \notin \mathcal{J} \\ 0 & \text{if } e = 2n+1 \end{cases} \tag{20}$$

The first two cases are related to the same decision (selecting item i or not). The only difference is that we keep track of the presence of the item only if $i(e) \in \mathcal{J}$ (first case), whereas only the knapsack resource consumption is impacted in the second case. The next two cases are the same as in the original recursion. The next case applies when $i(e) \notin \mathcal{J}$: one does not know whether $i(e)$ is in the knapsack or not, so there is a choice between removing item $i(e)$ or not. For a set $\mathcal{J}$ and a given vector $\boldsymbol{\pi} \in \mathbb{R}^n$, the value of the dual function is $L_{\mathcal{J}}(\boldsymbol{\pi}) = \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(1, 0, \mathbf{0})$.

In the remainder of the paper, we use a more compact expression of the dynamic program (notation-wise). For this purpose, we introduce so-called *actions* corresponding to possible decisions made at each state of the dynamic program. There are $3n$ different actions, 3 per item i: add item i (denoted a_i^+), remove item i (denoted a_i^-), and do not select item i (denoted $a_i^=$). Let $\mathcal{A}(e)$ be the set of possible actions for event $e \in \mathcal{E}$. If $e \in \mathcal{E}^{\text{in}}$, $\mathcal{A}(e) = \{a^+_{i(e)}, a^=_{i(e)}\}$. If $e \in \mathcal{E}^{\text{out}}$, $\mathcal{A}(e) = \{a^-_{i(e)}, a^=_{i(e)}\}$.

Each action a has a weight $\hat{w}(a)$ corresponding with the consumption/production of resource w of this action, an item resource consumption $\mathbf{d}(a)$ and a profit $\hat{p}(a)$ collected when this action is chosen. The value of $\hat{w}(a_i^+)$ is w_i, its profit $\hat{p}(a_i^+)$ is $\frac{1}{2}p_i$ and $\mathbf{d}(a_i^+) = \boldsymbol{\varepsilon}_i$. The value of $\hat{w}(a_i^=)$ is 0, $\hat{p}(a_i^=) = 0$, and $\mathbf{d}(a_i^=) = \mathbf{0}$. The value $\hat{w}(a_i^-)$ is $-w_i$, the profit $\hat{p}(a_i^-)$ is $\frac{1}{2}p_i$ and $\mathbf{d}(a_i^-) = -\boldsymbol{\varepsilon}_i$. The modified profit of action a is $\tilde{p}^{\boldsymbol{\pi}}(a) = \hat{p}(a) + \boldsymbol{\pi}^\top \mathbf{d}(a)$. When a specific state $(e, w, \mathbf{d})$ is considered, the set of possible actions $\mathcal{A}(e)$ can be refined by taking into account the value of $\mathbf{d}$, yielding the set $\mathcal{A}_{\mathcal{J}}(e, w, \mathbf{d}) \subseteq \mathcal{A}(e)$:

$$\mathcal{A}_{\mathcal{J}}(e, w, \mathbf{d}) = \{a \in \mathcal{A}(e) : 0 \leq w + \hat{w}(a) \leq W \text{ and } \forall i \in \mathcal{J}, 0 \leq \mathbf{d}_i + \mathbf{d}(a)_i \leq 1\}$$

We also use notation $\mathbf{d}^{\mathcal{J}}(a) = \mathbf{d}(a)$ if $\{e : a \in \mathcal{A}(e)\}$ is such that $i(e) \in \mathcal{J}$, $\mathbf{0}$ otherwise. We are now ready to rewrite (20) in a more compact way.

$$\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d}) = \begin{cases} 0 & \text{if } e = 2n+1 \\ \max\limits_{a \in \mathcal{A}_{\mathcal{J}}(e, w, \mathbf{d})} \left\{\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1, w + \hat{w}(a), \mathbf{d} + \mathbf{d}^{\mathcal{J}}(a)) + \tilde{p}^{\boldsymbol{\pi}}(a)\right\} & \text{otherwise} \end{cases} \tag{21}$$

For $i \in \mathcal{J}$, the contribution of $\boldsymbol{\pi}_i$ cancels out in the cost of all solutions of the relaxation. So either the presence of an item is recorded in the state space, or its Lagrangian cost is accounted. This formulation is equivalent to (20) and allows to explain some refinements to this DP in a simpler way.

3.3 Solving the relaxation and filtering

In step 2 of the SSDP algorithm, the relaxation we need to solve optimally is defined by a set $\mathcal{J}$ and a vector of Lagrangian multipliers $\boldsymbol{\pi}^*_{\mathcal{J}}$, which is a near-optimal vector of Lagrangian multipliers obtained by solving approximately the so-called Lagrangian dual problem $\min_{\boldsymbol{\pi}\in\mathbb{R}^n}\{L_{\mathcal{J}}(\boldsymbol{\pi})\}$. In the case of a maximization problem, the Lagrangian function is known to be convex, which implies that minimizing this function can be done using a subgradient algorithm, or one of its refinements (see *e.g.* [BA00]). Similarly to several other works ([IN94, TFA09]), the iterative method to find near-optimal multipliers will also be used to eliminate some dynamic programming states and transitions that cannot belong to optimal solutions of (16).

Practically speaking, $L_{\mathcal{J}}(\boldsymbol{\pi})$ is computed using the graph representation of (21), modified to save the filtering information from previous iterations (more details are given in Section 3.4). In the Directed Acyclic (Multi-)Graph (DAG) $G^{\boldsymbol{\pi}}_{\mathcal{J}} = (V_{\mathcal{J}}, H_{\mathcal{J}}, \tilde{p}^{\boldsymbol{\pi}})$, the set of nodes $V_{\mathcal{J}}$ represents the reachable states, and the set of arcs $H_{\mathcal{J}}$ represents the feasible actions that can be applied from each state. We uniquely identify each element of $V_{\mathcal{J}}$ using the label of the corresponding DP state. For an arc $u \in H_{\mathcal{J}}$, we denote by $a(u)$ the action related to u. An arc u related to action $a = a(u)$ performed from state $(e, w, \mathbf{d})$ connects its tail $\tau(u) = (e, w, \mathbf{d})$ to its head $h(u) = (e+1, w+\hat{w}(a), \mathbf{d}+\mathbf{d}^{\mathcal{J}}(a))$. The cost of arc u is $\tilde{p}^{\boldsymbol{\pi}}(a)$. We denote by $s^0 = (1, 0, \mathbf{0})$ the initial state (and source of $G^{\boldsymbol{\pi}}_{\mathcal{J}}$), and by $s^{\Omega} = (2n+1, 0, \mathbf{0})$ the unique terminal state (and sink node of $G^{\boldsymbol{\pi}}_{\mathcal{J}}$). The value of $L_{\mathcal{J}}(\boldsymbol{\pi})$ is the maximum cost of a path from s^0 to s^{Ω}. We solve the Lagrangian dual problem using Volume algorithm proposed in [BA00]. This approximate method builds a sequence of solutions $(\boldsymbol{\pi}^{(t)}_{\mathcal{J}})_t$ that converges to the optimum. This means that one has to solve repeatedly the maximum cost path problem in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$, which can be done in $O(|V_{\mathcal{J}}| + |H_{\mathcal{J}}|)$ using Bellman's algorithm.

Observation 2. *A solution to a relaxation based on $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ is basically defined as a path, which corresponds to a sequence of actions. Conversely, any sequence of actions corresponds to exactly one or zero path in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$. In the remainder of this section, a* solution *will indifferently refer to one or the other representation.*

Observation 3. *Problem* (7)-(15) *is equivalent to the problem defined by graph $G^{\boldsymbol{\pi}}_{\mathcal{I}}$, for all $\boldsymbol{\pi} \in \mathbb{R}^n$. Indeed, any path in $G^{\boldsymbol{\pi}}_{\mathcal{I}}$ defines a feasible solution of* (7)-(15) *with the same cost since the contributions of Lagrangian multipliers cancel out.*

Now, we recall a result used in [Iba87, IN94] and used to remove unnecessary states and actions in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$. For this purpose, let us remark that for any node $(e, w, \mathbf{d}) \in V_{\mathcal{J}}$, Bellman function value $\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d})$ is equal to the maximum cost of a path in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ from $(e, w, \mathbf{d})$ to s^{Ω}. Likewise, we define $\gamma^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d})$ as the maximum cost of a path from s^0 to $(e, w, \mathbf{d})$, which can be computed in a similar way.

Proposition 1 ([Iba87])**.** *For $\mathcal{J} \subseteq \mathcal{I}$, let $u \in H_{\mathcal{J}}$ such that $\tau(u) = (e^1, w^1, \mathbf{d}^1)$ and $h(u) = (e^2, w^2, \mathbf{d}^2)$, and $\boldsymbol{\pi} \in \mathbb{R}^n$. The following value is an upper bound on the cost of any path in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ going through u:*

$$\gamma^{\boldsymbol{\pi}}_{\mathcal{J}}(e^1, w^1, \mathbf{d}^1) + \tilde{p}^{\boldsymbol{\pi}}(u) + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e^2, w^2, \mathbf{d}^2)$$

The above proposition is useful to remove arcs from the graph of the current relaxation, as well as arcs corresponding with the same action from subsequently built relaxations. However, to our knowledge, the validity of this permanent removal is only implicitly assumed. We now give a formal proof in our specific context. Note that, by construction, the set of states $V_{\mathcal{J}}$ yielded by (21) is composed of elements $(e, w, \mathbf{d})$ such that $\mathbf{d}_i = 0$ for all $i \notin \mathcal{J}$. For the sake of readability, we denote by $\mathbf{proj}_{\mathcal{J}}(\mathbf{d})$ the projection of $\mathbf{d}$ onto the relevant space for relaxations defined by $\mathcal{J}$: $(\mathbf{proj}_{\mathcal{J}}(\mathbf{d}))_i = \mathbf{d}_i$ if $i \in \mathcal{J}$, and $(\mathbf{proj}_{\mathcal{J}}(\mathbf{d}))_i = 0$ if $i \notin \mathcal{J}$. Conversely, $\mathbf{proj}^{-1}_{\mathcal{J}}(\mathbf{d})$ denotes the set of binary vectors that project onto $\mathbf{d}$ when only components in $\mathcal{J}$ are considered: $\mathbf{proj}^{-1}_{\mathcal{J}}(\mathbf{d}) = \{\mathbf{d}' \in \{0,1\}^n : \mathbf{d}'_i = \mathbf{d}_i, i \in \mathcal{J}\}$.

The following lemma shows that the set of actions from each state of a refined relaxation related to $\mathcal{J}$' is a subset of the actions from its projected state in the looser relaxation related to $\mathcal{J}$.

Lemma 1. *Let us consider $e \in \mathcal{E}$, $w \in \{0, \ldots, W\}$, $\mathcal{J}$ and $\mathcal{J}$' such that $\mathcal{J} \subseteq \mathcal{J}' \subseteq \mathcal{I}$, $\mathbf{d} \in \{0,1\}^n$ and $\mathbf{d}' \in \mathbf{proj}^{-1}_{\mathcal{J}}(\mathbf{d})$. Then $\mathcal{A}_{\mathcal{J}}(e, w, \mathbf{d}) \supseteq \mathcal{A}_{\mathcal{J}'}(e, w, \mathbf{d}')$.*

Proof. Proof. Assume $a \in \mathcal{A}_{\mathcal{J}'}(e, w, \mathbf{d}')$. Then $0 \leq w + \hat{w}(a) \leq W$, since this relation does not depend on set $\mathcal{J}$. Moreover, for all $i \in \mathcal{J}'$ we have that $0 \leq \mathbf{d}'_i + \mathbf{d}^{\mathcal{J}'}(a)_i \leq 1$, so for all $i \in \mathcal{J}$, $0 \leq \mathbf{d}'_i + \mathbf{d}^{\mathcal{J}'}(a)_i \leq 1$. For all $i \in \mathcal{J}$, $\mathbf{d}'_i = \mathbf{d}_i$, and $\mathbf{d}^{\mathcal{J}'}(a)_i = \mathbf{d}(a)_i = \mathbf{d}^{\mathcal{J}}(a)_i$. It follows that $0 \leq \mathbf{d}_i + \mathbf{d}^{\mathcal{J}}(a)_i \leq 1$, which completes the proof. □ □

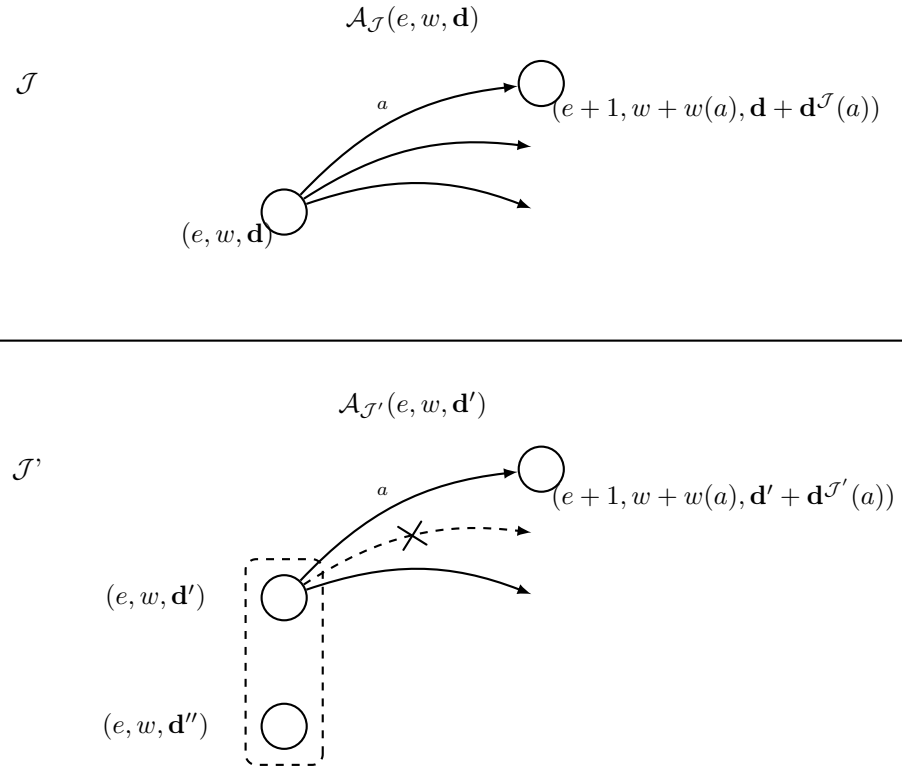

Figure 1: Illustration of Lemmas 1 and 2: $(e,w,\mathbf{d}')$ and $(e,w,\mathbf{d}'')$ correspond to the same label $(e,w,\mathbf{d})$ in the looser relaxation related to $\mathcal{J}$, i.e. they belong to the set $\{(e,w,\bar{\mathbf{d}}) : \bar{\mathbf{d}} \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d})\}$. Some of the actions out of $(e,w,\mathbf{d})$ are not consistent with both $(e,w,\mathbf{d}')$ and $(e,w,\mathbf{d}'')$. That is why the value of Bellman function at $(e,w,\mathbf{d}')$ and $(e,w,\mathbf{d}'')$ cannot be larger than that at $(e,w,\mathbf{d})$.

The next lemma shows formally that for a given vector of multipliers $\boldsymbol{\pi}$ the lagrangian cost of performing an action from a specific state cannot increase when the relaxation is refined.

Lemma 2. *Let us consider $e \in \mathcal{E}$, $w \in \{0,\dots,W\}$, and $\mathbf{d} \in \{0,1\}^n$. Let also $\mathcal{J}$and $\mathcal{J}$' be two sets such that $\mathcal{J} \subseteq \mathcal{J}' \subseteq \mathcal{I}$, $\boldsymbol{\pi}$ a vector of Lagrangian multipliers, and $\mathbf{d}' \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d})$. Then $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e,w,\mathbf{d}) \geq \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e,w,\mathbf{d}')$ and $\gamma_{\mathcal{J}}^{\boldsymbol{\pi}}(e,w,\mathbf{d}) \geq \gamma_{\mathcal{J}'}^{\boldsymbol{\pi}}(e,w,\mathbf{d}')$.*

Proof. Proof. We proceed by induction on e to prove the part of the proposition involving α. A straightforward adaptation of the proof yields the result for γ. At rank $e = 2n+1$, we have $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e,0,\mathbf{d}) = \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e,0,\mathbf{d}') = 0$ and $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e,w,\mathbf{d}) = \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e,w,\mathbf{d}') = \infty$, $w > 0$.

At rank $e \in \{1,\dots,2n\}$, assume that $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e+1,w,\bar{\mathbf{d}}) \geq \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e+1,w,\bar{\mathbf{d}}')$ for all $w \in \{0,\dots,W\}$, $\bar{\mathbf{d}} \in \{0,1\}^n$ and $\bar{\mathbf{d}}' \in \mathbf{proj}_{\mathcal{J}}^{-1}(\bar{\mathbf{d}})$. We first show that for a given action $a \in \mathcal{A}_{\mathcal{J}'}(e,w,\mathbf{d}')$, the profit resulting from applying a from state $(e,w,\mathbf{d}')$ cannot be larger than the profit resulting from applying a from its projected state $(e,w,\mathbf{d})$. For all $i \in \mathcal{J}$, since $\mathbf{d}'_i = \mathbf{d}_i$ and $(\mathbf{d}^{\mathcal{J}'}(a))_i = (\mathbf{d}^{\mathcal{J}}(a))_i$, we have $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e+1, w+\hat{w}(a), \mathbf{d}+\mathbf{d}^{\mathcal{J}}(a)) \geq \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e+1, w+\hat{w}(a), \mathbf{d}'+\mathbf{d}^{\mathcal{J}'}(a))$, which is equivalent to $\alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e+1, w+\hat{w}(a), \mathbf{d}+\mathbf{d}^{\mathcal{J}}(a)) + \tilde{p}^{\boldsymbol{\pi}}(a) \geq \alpha_{\mathcal{J}'}^{\boldsymbol{\pi}}(e+1, w+\hat{w}(a), \mathbf{d}'+\mathbf{d}^{\mathcal{J}'}(a)) + \tilde{p}^{\boldsymbol{\pi}}(a)$. Besides, from Proposition 1, $\mathcal{A}_{\mathcal{J}}(e,w,\mathbf{d}) \supseteq \mathcal{A}_{\mathcal{J}'}(e,w,\mathbf{d}')$. Hence, the result holds. □ □

The proposition below is crucial for the efficiency of SSDP algorithm: it shows that once an arc is eliminated from a graph at a given iteration, all corresponding arcs can be removed from the graphs built during subsequent iterations.

Proposition 2. *Let LB be a valid lower bound for the problem, $\mathcal{J} \subseteq \mathcal{I}$, $u \in H_{\mathcal{J}}$ related to action a such that $\tau(u) = (e^1, w^1, \mathbf{d}^1)$ and $h(u) = (e^2, w^2, \mathbf{d}^2)$, and $\boldsymbol{\pi} \in \mathbb{R}^n$. If $\gamma_{\mathcal{J}}^{\boldsymbol{\pi}}(e,w,\mathbf{d}^1) + \tilde{p}^{\boldsymbol{\pi}}(u) + \alpha_{\mathcal{J}}^{\boldsymbol{\pi}}(e',w',\mathbf{d}^2) < LB$, then the shortest path problem in $G_{\mathcal{J}}^{\boldsymbol{\pi}}$ without arc u is a relaxation of (7)-(15). Moreover, for any $\mathcal{J}' \supseteq \mathcal{J}$ and $\boldsymbol{\pi}' \in \mathbb{R}^n$, the shortest path problem $G_{\mathcal{J}'}^{\boldsymbol{\pi}'}$ without any arc related to action a from states $(e,w,\mathbf{d}')$ such that $\mathbf{d}' \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d})$ is a relaxation of (7)-(15) as well.*

Proof. Proof. First, we prove the validity of the proposition for the same vector $\boldsymbol{\pi}$ and a relaxation refined by enforcing a larger set of consistency constraints $\mathcal{J}'$. Let us consider arc $v \in \mathcal{H}_{\mathcal{J}'}$, related to action a such that $\tau(v) = (e^1, w^1, \mathbf{d}^3)$ with $\mathbf{d}^3 \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d}^1)$. Then $h(v) = (e^2, w^2, \mathbf{d}^3 + \mathbf{d}^{\mathcal{J}'}(a))$. Since $\mathbf{d}^3 \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d}^1)$ and $\mathcal{J}' \supseteq \mathcal{J}$, for all $i \in \mathcal{J}$, $(\mathbf{d}^3 + \mathbf{d}^{\mathcal{J}'}(a))_i = (\mathbf{d}^1 + \mathbf{d}^{\mathcal{J}}(a))_i = \mathbf{d}^2_i$. Hence, $\mathbf{d}^3 + \mathbf{d}^{\mathcal{J}'}(a) \in \mathbf{proj}_{\mathcal{J}}^{-1}(\mathbf{d}^2)$ and Proposition 2 implies that $\gamma^{\boldsymbol{\pi}}_{\mathcal{J}'}(e^1, w^1, \mathbf{d}^3) + \tilde{p}^{\boldsymbol{\pi}}(v) + \alpha^{\boldsymbol{\pi}}_{\mathcal{J}'}(e^2, w^2, \mathbf{d}^3 + \mathbf{d}^{\mathcal{J}'}(a)) < LB$. Arc v being part of a feasible solution of the relaxation defined by $G^{\boldsymbol{\pi}}_{\mathcal{J}'}$ that is optimal for the problem would contradict LB being a lower bound. It follows that v can be removed from $G^{\boldsymbol{\pi}}_{\mathcal{J}'}$ that shall still define a relaxation of (7)-(15).

Second, we prove the validity of the proposition for a fixed set of consistency constraints $\mathcal{J}$ and a different vector of multipliers $\boldsymbol{\pi}'$. Proposition 1 shows that arc u cannot be part of a feasible solution of the relaxation associated with $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ that would be optimal (and feasible) for the problem, since that would imply that LB is not a lower bound. Hence, no solution using u in $G^{\boldsymbol{\pi}'}_{\mathcal{J}}$, $\boldsymbol{\pi}' \in \mathbb{R}^n$ can be optimal for the problem, and removing u does not remove optimal solutions of the problem from those relaxations. □ □

An interesting feature of Bellman's algorithm is that it is able to compute the shortest path from the source s^0 to all vertices in one pass. Values $\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d})$ and $\gamma^{\boldsymbol{\pi}}_{\mathcal{J}}(e, w, \mathbf{d})$ can be computed for all nodes $s \in V_{\mathcal{J}}$ in two passes using respectively a forward and a backward dynamic programming algorithm.

3.4 Sublimation and convergence

In SSDP, the sublimation phase consists in strenghtening the current relaxation by enforcing some constraints that are violated in the current solution. At a given iteration of SSDP, the set of items in the state space $\mathcal{K}$ is constructed by adding items to the set $\mathcal{J}$ of items that were considered in the previous iteration.

We denote by $\mathcal{A}^{\ell}_{\mathcal{J}}(e, w, \mathbf{d})$ the set of actions that have not been filtered for state $(e, w, \mathbf{d})$ at the end of the step related to set $\mathcal{J}$. Proposition 2 allows us to design the following truncated dynamic program.

$$\alpha^{\boldsymbol{\pi}}_{\mathcal{K}}(e, w, \mathbf{d}) = \begin{cases} 0 & \text{if } e = 2n+1 \\ \max\limits_{a \in \mathcal{A}_{\mathcal{K}}(e,w,\mathbf{d}) \cap A^{\ell}_{\mathcal{J}}(e,w,\mathbf{proj}_{\mathcal{J}}(\mathbf{d}))} \left\{ \alpha^{\boldsymbol{\pi}}_{\mathcal{K}}(e+1, w + \hat{w}(a), \mathbf{d} + \mathbf{d}^{\mathcal{K}}(a)) + \tilde{p}^{\boldsymbol{\pi}}(a) \right\} & \text{otherwise} \end{cases} \tag{22}$$

From an algorithmic point of view, it means that the DAG of the previous iteration is kept, and when it comes to generate the possible actions from state s, the vertex corresponding with the projection of s in the previous state space has to be retrieved, and only the actions belonging to the filtered graph (*i.e.* the graph obtained by iteratively removing arcs of too large Lagrangian cost) are considered.

The maximum number of iterations of the algorithm is n: indeed, at least one item index is added to $\mathcal{J}$ at step 4. When $\mathcal{J} = \mathcal{I}$, the relaxation solved at step 2 is actually equivalent to (7)-(15) (Observation 3). However a feasible solution may be found at step 2 when $\mathcal{J} \neq \mathcal{I}$. In the latter case, the cost of this solution in model (17)-(19) is equal to its cost in (7)-(15), so that it provides both a dual and a primal bound with the same value and the algorithm terminates with this optimal solution.

4 Refinements of SSDP to solve efficiently TKP

Preliminary computational experiments showed that a direct implementation of SSDP for TKP is not able to produce results that can compete with state-of-the-art TKP solvers. This can be explained by several issues: the method takes a large computing time to compute the first relaxation, many states that are not useful are generated when the first relaxation is computed, and the gap is not reduced enough when only one dimension is reintroduced in the sublimation phase.

We now propose several techniques that we use to deal with these issues, and improve the performance of SSDP for solving TKP.

4.1 Attaching additional information to the states

Let $\mathcal{J}$ be the set of dimensions that are currently taken into account in the dynamic program. For a given state $s = (e, w, \mathbf{d})$ such that $e \in \mathcal{E}^{\text{out}}$, if $i(e) \notin \mathcal{J}$, two choices are possible: removing $i(e)$ from

the knapsack, or not. However, it may happen that s cannot be reached by any sequence of unfiltered actions that have added $i(e)$ to the knapsack. Similarly, it may happen that all of them have added $i(e)$ to the knapsack. In such cases, we know for sure whether item $i(e)$ is in the knapsack or not.

Instead of disregarding completely all items that do not belong to $\mathcal{J}$, we attach to each state s an additional vector $\mathbf{d}^\eta(s) \in \{0,1,\emptyset\}^n$. If $\mathbf{d}^\eta(s)_i = \emptyset$, we do not know if i is in the knapsack, if $\mathbf{d}^\eta(s)_i = 0$, we know that i is not in the knapsack, and if $\mathbf{d}^\eta(s)_i = 1$, we know that i is in the knapsack. Clearly, if $i \in \mathcal{J}$, $\mathbf{d}^\eta(s)_i$ cannot be $\emptyset$, since the presence of i is recorded into the state space.

When $i \notin J$, we set $\mathbf{d}^\eta(s^0)_i = 0$, and we compute $\mathbf{d}^\eta(s)_i$ recursively for each other state s as follows:

$$\mathbf{d}^\eta(s)_i = \begin{cases} 1 & \text{if } \forall u \in H_{\mathcal{J}} : h(u) = s, \quad \big(\mathbf{d}^\eta(\tau(u))_i = 1 \text{ and } \mathbf{d}(a(u))_i = 0\big) \text{ or } \mathbf{d}(a(u))_i = 1 \\ 0 & \text{if } \forall u \in H_{\mathcal{J}} : h(u) = s, \quad \big(\mathbf{d}^\eta(\tau(u))_i = 0 \text{ and } \mathbf{d}(a(u))_i = 0\big) \text{ or } \mathbf{d}(a(u))_i = -1 \\ \emptyset & \text{otherwise} \end{cases}$$

Practically speaking, vector $\mathbf{d}^\eta(s)$ is computed on the fly while the DAG is created. Note that $\mathbf{d}^\eta(s)$ is not part of the label definition: this is an additional information that is attached to s. We attach another information to each state s, which corresponds with redundant constraints. For each state $s = (e, w, \mathbf{d})$, we define $q^\eta_{\min}(s)$ (resp. $q^\eta_{\max}(s)$) as a lower (resp. upper) bound on the number of items that can be into the knapsack in partial solutions that correspond to this state. These values can be computed recursively, similarly to vector $\mathbf{d}^\eta(s)$. Here, $\mathbf{1}$ stands for the unit vector of size n.

$$\begin{aligned} q^\eta_{\min}(s^0) &= 0 \\ q^\eta_{\min}(s) &= \min_{u \in H_{\mathcal{J}} : h(u) = s} \{q^\eta_{\min}(\tau(u)) + \mathbf{1}^\top \mathbf{d}(a(u))\} \\ q^\eta_{\max}(s^0) &= 0 \\ q^\eta_{\max}(s) &= \max_{u \in H_{\mathcal{J}} : h(u) = s} \{q^\eta_{\max}(\tau(u)) + \mathbf{1}^\top \mathbf{d}(a(u))\} \end{aligned}$$

For each event e, let $Q^{\max}(e) = \{\max |S| : |S| \subseteq \mathcal{I}(e), \sum_{i \in S} w_i \leq W\}$ be the maximum number of items that can belong to the knapsack when this event occurs (this value can be computed in linear time of $|\mathcal{I}(e)|$ for each e when the elements of this set are sorted by non-decreasing order of their size). Obviously, for state $(e, w, \mathbf{d})$, the number of items should be in $[0, Q^{\max}(e)]$. From an ILP perspective, this redundant constraint can be added as the following system of variables/constraints to (7)–(15) in order to strengthen relaxations.

$$q_e = q_{e-1} + y_e \quad e \in \mathcal{E}^{\text{in}} \tag{23}$$
$$q_e = q_{e-1} - y_e \quad e \in \mathcal{E}^{\text{out}} \tag{24}$$
$$q_0 = 0 \tag{25}$$
$$q_e \leq Q^{\max}(e) \quad e = 1, \ldots, 2n \tag{26}$$
$$q_{2n+1} = 0 \tag{27}$$
$$q_e \in \mathbb{R}_+, \quad e = 0, \ldots, 2n+1 \tag{28}$$

From a DP perspective, we use the information attached to the states to reduce the size of the relaxed problems using the feasibility tests exposed in Section 4.3.

4.2 Dominance property

In the classical knapsack problem, an item i is dominated by item j if $p_i \leq p_j$, $w_i \geq w_j$ and one of the two inequalities is strict. In this case, from any feasible solution S where item i is chosen but not item j, we can build another solution where j is chosen and whose profit is not smaller than that of S. Thus, among solutions that include i, only those including j as well need to be considered. For TKP, dominance relations must take into account the temporal aspect.

Proposition 3. *Item i is dominated by item j if $p_i \leq p_j$, $w_i \geq w_j$, $t^-_i \leq t^-_j$, $t^+_i \geq t^+_j$ and one of the four inequalities is strict.*

Proof. Proof. Since j is not less profitable and not larger than i, and its time window is included in that of i, one can replace i with j in any solution and obtain another feasible solution with a not smaller profit. □ □

This dominance property is useful when one knows exactly the contents of the knapsack. However this is not the case when solving a relaxation. Then, for a given state s, vector $\mathbf{d}^\eta(s)$ containing the mandatory and forbidden items is useful, since it may assure the presence of some item whose related consistency constraint is relaxed. The dominance relation is used as follows: if i is dominated by j and $\mathbf{d}^\eta(s)_i = 1$ then action $a^=_j$ is not allowed from s.

This dominance relation can be exploited more effectively by modifying the recurrence equations as follows. Let us consider $e \in \mathcal{E}^{\text{in}}$, state $(e, w, \mathbf{d}) \in V_\mathcal{J}$ and action $a^=_{i(e)} \in \mathcal{A}(e)$. Then, in equations (21) and (22), the index $e+1$ related to $a^=_{i(e)}$ can be replaced with $e' = \min\{f \in \mathcal{E} : f > e, f \in \mathcal{E}^{\text{out}}$ or $i(f)$ is not dominated by $i(e)\}$.

4.3 Feasibility tests

In the sequel, a *feasible state* is defined as a state that can be generated from s^0 by applying a feasible sequence of actions following recurrence equations (16). We denote by V the set of feasible states. Moreover, we recall that $V_\mathcal{J}$ is the set of DP states considered while solving the relaxation related to $\mathcal{J}$. A sufficient condition for the global solving process to be valid is that for all $\mathcal{J} \subseteq \mathcal{I}$, $V_\mathcal{J}$ contains the set of states generated by applying from s^0 at least one optimal sequence of actions. Thus, any state in $V_\mathcal{J}$ that is not the projection of a feasible state in V can be removed without impairing the validity of the algorithm. We now describe several techniques used to detect infeasibilities of states at early stages of the method. The following results are stated without proof. The first feasibility test checks that the number of items in the knapsack is consistent.

Proposition 4. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $s = (e, w, \mathbf{d}) \in V_\mathcal{J}$. If $|i \in \mathcal{I} : \mathbf{d}^\eta(s)_i \neq 0| < q^\eta_{\min}(s)$ or $|i \in \mathcal{I} : \mathbf{d}^\eta(s)_i = 1| > q^\eta_{\max}(s)$ then $\mathbf{proj}^{-1}_\mathcal{J}(s) \cap V = \emptyset$ and s can be removed from $V_\mathcal{J}$.*

Another feasibility test is based on the set of possible weights of subsets of items that can belong to a knapsack at a given event. For this purpose, we precompute for each event e the following set: $\mathcal{F}(e) = \{\sum_{i\in S} w_i : S \subseteq \mathcal{I}(e), \sum_{i\in S} w_i \leq W\}$, which corresponds with all reachable weights of a subset of items. Each of these sets can be computed in $\mathcal{O}(nW)$ using a straightforward dynamic programming algorithm. Proposition (5) follows from the definition of $\mathcal{F}$.

Proposition 5. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $s = (e, w, \mathbf{d}) \in V_\mathcal{J}$. If $w \notin \mathcal{F}(e)$ then $\mathbf{proj}^{-1}_\mathcal{J}(s) \cap V = \emptyset$ and s can be removed from $V_\mathcal{J}$.*

This rule can be improved by considering additional information gathered from $\mathbf{d}^\eta(s)$. We precompute, for each event e and each item i, $\mathcal{F}^+(e, i)$ and $\mathcal{F}^-(e, i)$ which are respectively the possible weights that can be reached using item i, and the weights reachable without item i. We have $\mathcal{F}^+(e, i) = \{\sum_{j\in S\cup\{i\}} w_j : S \subseteq \mathcal{I}(e), \sum_{j\in S} w_j \leq W - w_i\}$ and $\mathcal{F}^-(e, i) = \{\sum_{j\in S} w_j : S \subseteq \mathcal{I}(e)\backslash\{i\}, \sum_{j\in S} w_j \leq W\}$.

Proposition 6. *Let $\mathcal{J} \subseteq \mathcal{I}$, $s = (e, w, \mathbf{d}) \in V_\mathcal{J}$ and $i \in \mathcal{I}(e)$. If $\mathbf{d}^\eta(s)_i = 1$ and $w \notin \mathcal{F}^+(e, i)$ then $\mathbf{proj}^{-1}_\mathcal{J}(s) \cap V = \emptyset$ and s can be removed from $V_\mathcal{J}$. Likewise, if $\mathbf{d}^\eta(s)_i = 0$ and $w \notin \mathcal{F}^-(e, i)$ then s can be removed from $V_\mathcal{J}$.*

The following result allows detecting states that can be generated only be removing an item from the knapsack without adding it first. In such cases, the weight recorded in the state might become less than the weight of the items whose presence in the knapsack is known for sure.

Proposition 7. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $s = (e, w, \mathbf{d}) \in V_\mathcal{J}$. If $\sum_{i\in\mathcal{I}:\mathbf{d}^\eta(s)_i=1} w_i > w$ then $\mathbf{proj}^{-1}_\mathcal{J}(s) \cap V = \emptyset$ and s can be removed from $V_\mathcal{J}$.*

The following feasibility test also integrates the bounds on the number of items in the knapsack to ensure the consistency of the labels with respect to the weight. It compares lower and upper bounds on the weight of the knapsack with the weight defining the label.

Proposition 8. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $s = (e, w, \mathbf{d}) \in V_\mathcal{J}$. Let $S^1 = \{i \in \mathcal{I}(e) : \mathbf{d}^\eta(s)_i \neq 0\}$ be the set of items potentially in the knapsack. If $\max\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \leq q^\eta_{\max}(s)\right\} < w$ or $\min\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \geq q^\eta_{\min}(s)\right\} > w$ then $\mathbf{proj}^{-1}_\mathcal{J}(s) \cap V = \emptyset$ and s can be removed from $V_\mathcal{J}$.*

4.4 Partial enumeration of actions

Combining several consecutive actions into single compound actions allows enforcing locally some constraints on items even if their consistency is not checked through the DP state space in current relaxation.

Our implementation of this idea uses an input parameter k^{enum}, which controls the depth of the enumeration of consecutive actions. More precisely, from a given state, instead of computing the two possible actions, we compute the $O(2^{k^{\text{enum}}})$ possible sequences of k^{enum} successive actions $(a_1, \ldots, a_{k^{\text{enum}}})$. This allows to exclude some sequences that are infeasible (sequences $(\ldots, a^+_{i(e)}, \ldots, a^=_{i(e)})$ or $(\ldots, a^=_{i(e)}, \ldots, a^-_{i(e)})$ for some e).

4.5 Criteria for choosing constraints to reintroduce

At each sublimation step, dimensions related to violated constraints are integrated into the state space. At a given iteration, let us denote $\boldsymbol{\pi}^q$ the vector of Lagrangian multipliers that achieves the q^{th} best dual bound during the relaxation solution step, and let $\boldsymbol{y}^q$ be the solution found when solving (17)-(19) to compute $L_{\mathcal{J}}(\boldsymbol{\pi}^q)$. Let $\mathcal{J}^{\neq} = \left\{i \in \mathcal{I} \setminus \mathcal{J} : \exists q \in \{1, \ldots, k^{\text{nbsol}}\}, y^q_{e^{in}(i)} \neq y^q_{e^{out}(i)}\right\}$ be the set of items whose consistency constraint is violated in one of the solutions of best relaxations solved for a fixed $\mathcal{J}$. Below we describe several criteria for estimating the computational attractiveness of adding a specific constraint. The next subsection proposes strategies for selecting constraints based on these criteria.

Note that, since the magnitude of the violation of constraints is always one in our relaxation, it is not significant when it comes to choosing a relaxed constraint to be reintroduced, contrary to application of SSDP in the context of scheduling. The first criterion (**Lagrangian Multipliers**) is to use the best Lagrangian multipliers $\boldsymbol{\pi}^1$ found to determine the profit related to each consistency constraint: $\psi^1_i = |\boldsymbol{\pi}^1_i|$.

The idea behind the second criterion (**Network Size**) is to control the number of states in the network after the sublimation step. For each constraint, we compute an estimation of the number of states added if the corresponding dimension – and only that one – is included into the state space. Remark that, when adding a single constraint i, only such states s where $\mathbf{d}^\eta(s)_i = \emptyset$ can yield two different states after sublimation. Hence, the second criterion we define is the opposite (smaller is better) of an upper bound on the number of additional states due to i: $\psi^2_i = -|s \in V_{\mathcal{J}} : \mathbf{d}^\eta(s)_i = \emptyset|$.

The third criterion (**Number of violations**) favors the constraints that are violated in many "good" solutions. For this purpose, we compute the frequency of the constraint violation in the k^{nbsol} best Lagrangian problem solutions: $\psi^3_i = \frac{1}{k^{\text{nbsol}}} \sum_{q=1}^{k^{\text{nbsol}}} \left|y^q_{e^{in}(i)} - y^q_{e^{out}(i)}\right|$.

4.6 Reintroducing batches of constraints

The sublimation step is a time-consuming procedure. However, preliminary experiments have shown that in many cases, adding only one dimension is not sufficient to ensure a significant increase in the dual bound. While adding several dimensions at each iteration is more efficient in many cases, adding many dimensions may increase dramatically the size of the network. Therefore, we face a bi-criterion optimization problem, where one has to consider the expected increase in the dual bound, and the expected increase in the size of the network.

The following proposition provides an upper bound on the number of labels in the network given a set of consistency constraints enforced in the state space.

Proposition 9. *Let $\mathcal{J}$ and $\mathcal{K}$ such that $\mathcal{J} \subseteq \mathcal{K} \subseteq \mathcal{I}$. It holds that $|V_{\mathcal{K}}| \leq \sum_{e \in \mathcal{E}} \left(2^{|(\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e)|} |\{(e, w, \mathbf{d}) \in V_{\mathcal{J}}\}|\right)$.*

Proof. Proof. From equations (22) and the definition of $\mathbf{d}^{\mathcal{K}}$, the set of labels created in $V_{\mathcal{K}}$ from a given label $(e, w, \mathbf{d}) \in V_{\mathcal{J}}$ is included in the set

$$\begin{aligned}\{(e, w, \mathbf{d}') :& \mathbf{d}'_i = \mathbf{d}_i \ \forall i \in \mathcal{J}, \\ & \mathbf{d}'_i \in \{0, 1\} \ \forall i \in (\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e), \\ & \mathbf{d}'_i = 0 \ \forall i \in \mathcal{I} \setminus (\mathcal{J} \cup (\mathcal{K} \cap \mathcal{I}(e)))\}\end{aligned}$$

whose cardinality is $2^{|(\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e)|}$. Summing up over all states in $V_{\mathcal{J}}$ yields the result. □ □

This result shows that adding a set $\mathcal{K} \setminus \mathcal{J}$ of constraints related to items with pairwise disjoint time windows is particularly attractive: in such cases, for all events $e \in \mathcal{E}$ we have that $|(\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e)| \leq 1$. It follows that the network will grow by a constant factor only, as stated formally in the next corollary. Let us remind that $(\mathcal{I}, E^{\text{int}})$ is the interval graph related to intervals $[t^-_i, t^+_i), i \in \mathcal{I}$ (see Section 2).

Corollary 1. *Let $\mathcal{J}$ and $\mathcal{K}$ such that $\mathcal{J} \subseteq \mathcal{K} \subseteq \mathcal{I}$, and $\mathcal{K} \setminus \mathcal{J}$ is a stable set in graph G^{int}. Then $|V_{\mathcal{K}}| \leq 2|V_{\mathcal{J}}|$.*

This observation guided our strategies to choose the set of constraints that are added during sublimation step. The first strategy, that we call **Weighted stable set**, consists in adding a set of constraints that form a stable set in G^{int}. In order to avoid obtaining a too large DP, we limit the expected network growth during the sublimation step (the maximum value allowed is denoted by MAXGROWTH). We aim at maximizing a criterion based on those presented in the previous section (more details on how ψ is defined are given in the computational experiments section). The problem of constraint selection can be cast as follows.

$$\max \sum_{i \in \mathcal{J}^{\neq}} \psi_i x_i \tag{29}$$

$$\sum_{i \in \mathcal{J}^{\neq}} \psi_i^2 x_i \leq \text{MAXGROWTH} \tag{30}$$

$$x_i + x_j \leq 1, \forall (i,j) \in E^{\text{int}} \cap (\mathcal{J}^{\neq} \times \mathcal{J}^{\neq}) \tag{31}$$

$$x_i \in \{0,1\}, \forall i \in \mathcal{J}^{\neq} \tag{32}$$

One can identify this problem as the disjunctive knapsack with interval conflicts, which is NP-complete, but can be solved in pseudo-polynomial time through dynamic programming (see [SV13]).

The Weighted stable set strategy has the disadvantage of sometimes adding too few new constraints, leading to a slow convergence of the overall algorithm. Our second strategy, called **Cardinality Constrained Stable Set** aims at circumventing this drawback. It consists in solving first the model above with unit profits to find a stable set of maximum cardinality (which we denote by $C_{\max}$). We then seek a stable set of cardinality larger than $C_{\max} * k^{\text{rstable}}$ where k^{rstable} is a parameter in $(0,1]$, by solving the following cardinality constrained maximum weight stable set problem:

$$\max \sum_{i \in \mathcal{J}^{\neq}} \psi_i x_i \tag{33}$$

$$\sum_{i \in \mathcal{J}^{\neq}} x_i \geq C_{\max} * k^{\text{rstable}} \tag{34}$$

$$x_i + x_j \leq 1, \forall (i,j) \in E^{\text{int}} \cap (\mathcal{J}^{\neq} \times \mathcal{J}^{\neq}) \tag{35}$$

$$x_i \in \{0,1\}, \forall i \in \mathcal{J}^{\neq} \tag{36}$$

The third strategy, called **k-coloring**, not only considers stable sets among available dimensions, but also dimensions that were added in the previous iterations. The rationale behind this technique is to consider the stable sets globally instead of focusing on a single one and disregarding the dimensions that were added before. At the end of the first iteration, a variable k^{color} is set to 1. At each iteration, we seek for the maximum weight k^{color}-colorable subset of $\mathcal{J}^{\neq} \cup \mathcal{J}$, where $\mathcal{J}$ is the set of already added constraints. If the solution is different from $\mathcal{J}$, then we add the new constraints selected. If the solution only contains $\mathcal{J}$, then we increase the value of k^{color} by one unit, and solve the model again. Obviously, there is always a feasible solution with a new consistency constraint to add for this new value of k^{color}. We solve repeatedly the following model, where z_{ij} are binary variables indicating that item i is assigned to color j.

$$\max \sum_{i \in \mathcal{J}^{\neq}} \sum_{j=1,\ldots,k^{\text{color}}} \psi_i z_{ij} \tag{37}$$

$$z_{ij} + z_{\ell j} \leq 1, \forall (i,\ell) \in E^{\text{int}} \cap ((\mathcal{J}^{\neq} \cup \mathcal{J}^{+}) \times (\mathcal{J}^{\neq} \cup \mathcal{J}^{+})), j \in \{1,\ldots,k^{\text{color}}\} \tag{38}$$

$$\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} \leq 1, \forall i \in \mathcal{J}^{\neq} \tag{39}$$

$$\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} = 1, \forall i \in \mathcal{J}^{+} \tag{40}$$

$$z_{ij} \in \{0,1\}, \forall i \in \mathcal{J}^{\neq} \cup \mathcal{J}^{+}, j \in \{1,\ldots,k^{\text{color}}\} \tag{41}$$

Finally, each strategy can be modified according to the following observation. In the first iterations, one would like to close the gap between the primal and dual bounds as fast as possible to allow a

better efficiency of filtering procedure. When the size of the network becomes large, the most important criterion becomes its size, since a too large network may lead to intractable Lagrangian subproblems. Therefore, when the size of the network is larger than a given threshold, we only base our strategy on the expected size of the network. We call this strategy **Hybrid**.

4.7 Implementation issues

The efficiency of the filtering step depends heavily on the fact that a good primal solution is known. In general, during the optimization of the Lagrangian multipliers, it may happen that a primal solution is computed as a side product of the method. However, one cannot rely on this for TKP, since many constraints are often violated in a solution of a relaxation. To produce a lower bound for our problem, we solve heuristically model (4)–(6) by giving a small amount of time to a general purpose integer linear programming solver. We found out that the solver is able to obtain good quality solutions quickly, but is not able to prove its optimality, even if it is given a large amount of time.

A good dual solution is also useful to warmstart Volume algorithm, which may take a large amount of time to converge when the computed DAG are large. To find a good set of multipliers, we solve the LP relaxation of (7)–(13), and use the optimal dual values of constraints (13). Solving the LP relaxation is fast and provides a good starting point for Volume.

When dominance rules are applied, many states have only one out-going action. This leads to unnecessary computational effort in Bellman's algorithm. To reduce the computational burden, we remove chains in the initial graph by linking the two extremities of the chain directly. This can be done on-the-fly when out-going actions are computed from a state.

We implemented a parallel version of Bellman's algorithm. We first compute the longest path (in term of number of arcs) from s^0 to all vertices. All vertices at the same distance are stored in a common bucket. The treatment of vertices in the same bucket can be done in parallel.

5 Computational experiments

In this section, we provide experimental results for our methods. For each refinement of the method, we evaluate its impact on the performance of the general algorithm. Finally, we compare our results to those of [GI14] and to the results obtained by an all-purpose commercial Integer Linear Programming solver. In this section, we consider an instance as solved if the algorithm finds an optimal solution and proves its optimality.

All our experiments are conducted using 2 Dodeca-core Haswell Intel Xeon E5-2680 v3 2,5 GHz with 128Go RAM. For each instance, our code was run on 6 threads and a 32 Go RAM limit. All models considered in subroutines are solved with IBM ILOG Cplex 12.7.

We use instances proposed in [CFM13], composed of two groups. For instances in the first group (I), results are not reported since [GI14] and our methods can solve all instances to optimality in a small amount of time. For the 100 instances in the second group (called U), the number of items is 1000, the size of the knapsack ranges from 500 to 520. Each item has a profit and a weight between 1 and 100.

The goal of our experiments is twofold. We first want to determine the best parameters for our method, and the impact of the different improvements that we have proposed. We also want to assess the efficiency of our method against the best methods from the literature.

In the sequel we present aggregated results. Detailed tables can be found in appendix (online supplement).

5.1 Parameters of the method

In this section, we evaluate the impact of the improvements that we have proposed in this document. For this purpose, we report the results obtained by the best combination of techniques (called MCF* below), and methods obtained from MCF* by deactiviting some features. We deactivated the initialization of the Lagrangian multipliers (subsection 4.7), the partial enumeration technique (subsection 4.4) and dominance and feasibility tests (subsections 4.2 and 4.3).

For each method obtained, we report in Figure 1 the number of instances solved along the time, with a maximum of three hours.

A first remark is that all techniques have a significant impact on the number of instances solved after three hours. It transpires from these experiments that the most important technique is the warm-start of the Lagrangian multipliers, followed by partial enumeration, and dominance/feasibility tests.

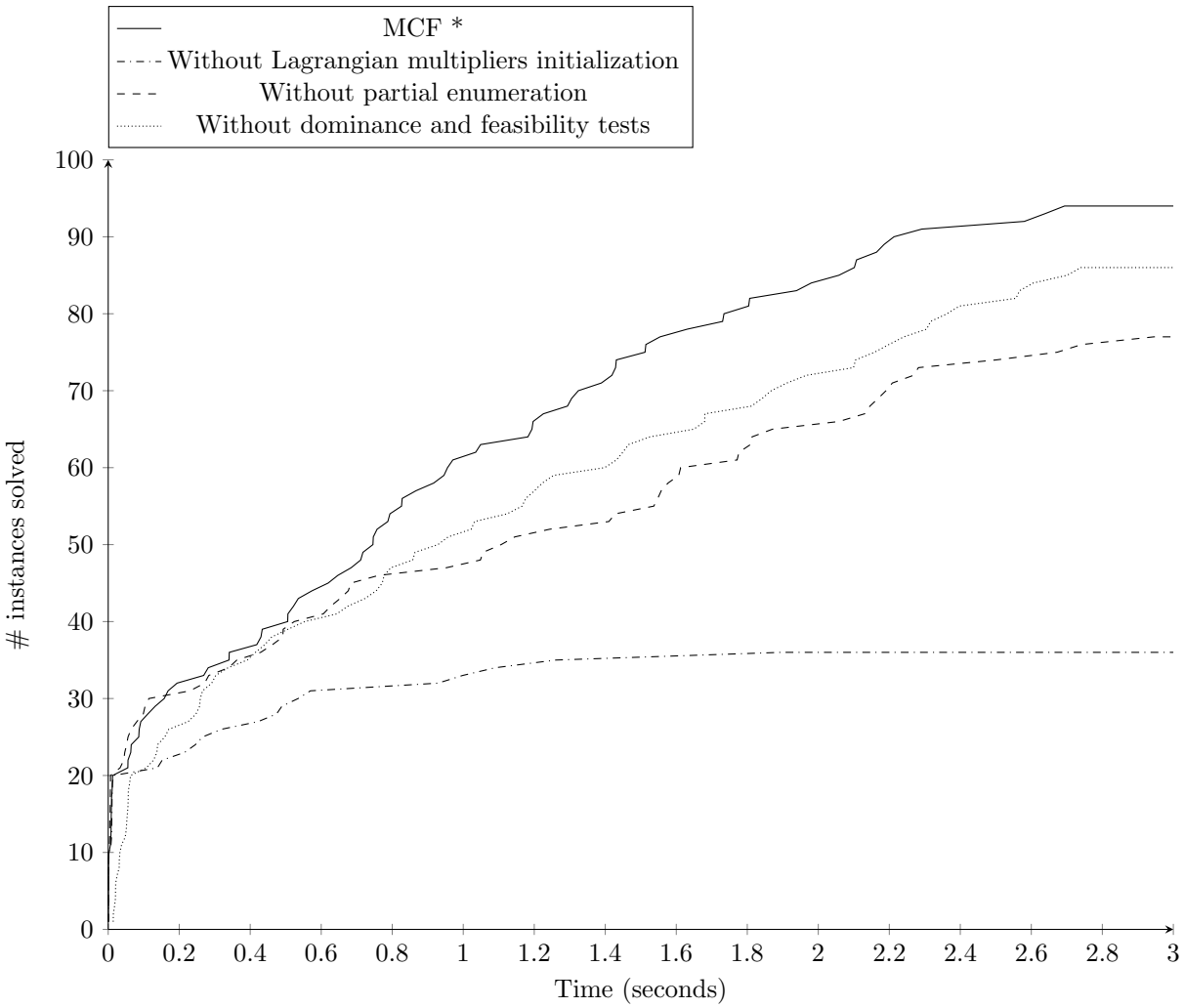

Figure 2: Number of instances of TKP from data set U ([CFMT16]) solved along time, for the best version of our algorithm, and three versions obtained by desactivating some techniques.

Initializing the Lagrangian multipliers at the first iteration with help of dual values of model (7)-(15) allows solving more than 2.5 times more instances compared to starting with null Lagrangian multipliers (94 against 36 out of 100 instances). There are two possible explanations for this fact: either warmstart helps Volume converging faster, or the dual bound obtained after convergence is better. To understand better the behavior of the algorithm, we report in Table 1 the average time needed by Volume to converge for the first network, the average gap with the best primal bound, and the average number of vertices and edges in the network after convergence (the two latter values indicate the strength of the filtering process).

Table 1: Impact of the initialization of the Lagrangian multipliers on the first relaxation solution and on the size of the network after the first filtering step. "k" stands for thousands.

	Average time (s.)	Average nodes	Average edges	Average gap
MCF* with initial LagMult	118	161 k	1,650 k	0.42%
MCF* without initial LagMult	191	188 k	2,136 k	1.53%

Table 1 indicates clearly that warmstart allows a faster and better convergence of Volume. Altough it takes almost 200 seconds on average to Volume to converge for an average gap of $1,53\%$, warmstart allows to converge in 118 seconds for an average gap more than three times smaller. The smaller gap leads to a better filtering phase, and a smaller network. This would hint that our implementation of Volume is able to converge toward good dual solution when it starts from a solution near the optimum, and that the dual values from the linear relaxation of (7)-(15) is actually of good estimation of the

optimal multipliers.

The number of instances solved optimally within 3 hours increases by about 20% when partial enumeration is used. This means that enumerating sequences of actions allows to include useful information that is used by the filtering algorithm. To illustrate the effect of partial enumeration, we report in Table 2 the size of the first network constructed, for different values of k^{enum}. As one can expect, increasing the number of consecutive actions considered reduces the number of nodes in the network but increases the number of arcs (since each combination of actions is replaced by one arc). Although it yields a higher memory consumption and a longer solving time of the relaxations, it can be advantageous to some extent: selecting one arc means deciding more actions simultaneously and more impact on the (Lagrangian) cost of the solution. Thus, such long sequences of decisions are more easily discarded by Lagrangian filtering. This also explains, along with the removal of short infeasible sequences, that the network with $k^{\text{enum}} = 2$ is smaller than that with $k^{\text{enum}} = 1$.

Table 2: Average size of first network for different value of k^{enum}, after the filtering step. "k" stands for thousands.

k^{enum}	1	2	3	4	5	6
Average nodes	703 k	384 k	264 k	202 k	162 k	135 k
Average transitions	1,392 k	1,344 k	1,639 k	2,269 k	3,155 k	4,658 k

Finally, the dominance property stated in Proposition 3 and the feasibility tests proposed in Propositions 4 to 8 have non-negligible impact on the performance of our algorithm, since they allow solving ten more instances in one hour-time limit, and eight more instances in three hour-time limit. These tests remove about 20% of the nodes and 32% of the arcs included in the first network when $k^{\text{enum}} = 4$.

5.2 Strategies for adding dimensions

As stated in most papers working on iterative state-space relaxations, the choice of the dimensions to add is the most critical component in the method. In what follows, we compare empirically our different strategies to determine the most effective.

In Table 3, we report how each configuration of our algorithm we tested is parameterized. For method MCF*, the weight associated with each constraint is a combination of the expected number of additional labels and the frequency of violation of this constraint in good relaxation solutions. When using Weighted stable set strategy, minimizing the expected number of added labels comes to selecting no new constraint. That is why we maximize the complement to the maximum number of expected additional labels. This value is weighted by the frequency of violation of the constraint. Configuration Hybrid aims at improving the dual bound as fast as possible by making the most often violated constraints feasible. Once the network is too large (we empirically fixed a limit at 4,000k nodes), adding fewer labels is preferred.

Table 3: Parameters of the tested methods.

Configuration	Strategy	Criterion ψ_i
MCF*	Cardinality constrained	$\psi_i^2\left(1-\psi_i^3\right)$
Stable	Weighted stable set	$\frac{-\psi_i^2+\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}{\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}\psi_i^3$
NbLabels	Cardinality constrained	ψ_i^2
Hybrid	Cardinality constrained	First ψ_i^3, then ψ_i^2
LagMult	Cardinality constrained	ψ_i^1
KColor	K-Color	$\frac{-\psi_i^2+\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}{\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}\psi_i^3$

In Figure 3, we compare empirically our different methods for choosing the dimensions to add. Similarly to Figure 2, we report the total number of instances solved along the time, with a maximum of three hours.

From Figure 3, we observe that $k-$coloring strategy is clearly not competitive compared to strategies based on stable sets. The fact that this strategy performs poorly shows that our method to evaluate the size of the network in the stable-set based strategies is useful, and that one cannot just rely on the interval structure of the constraints. All strategies based on stable sets have similar behaviour. Configuration

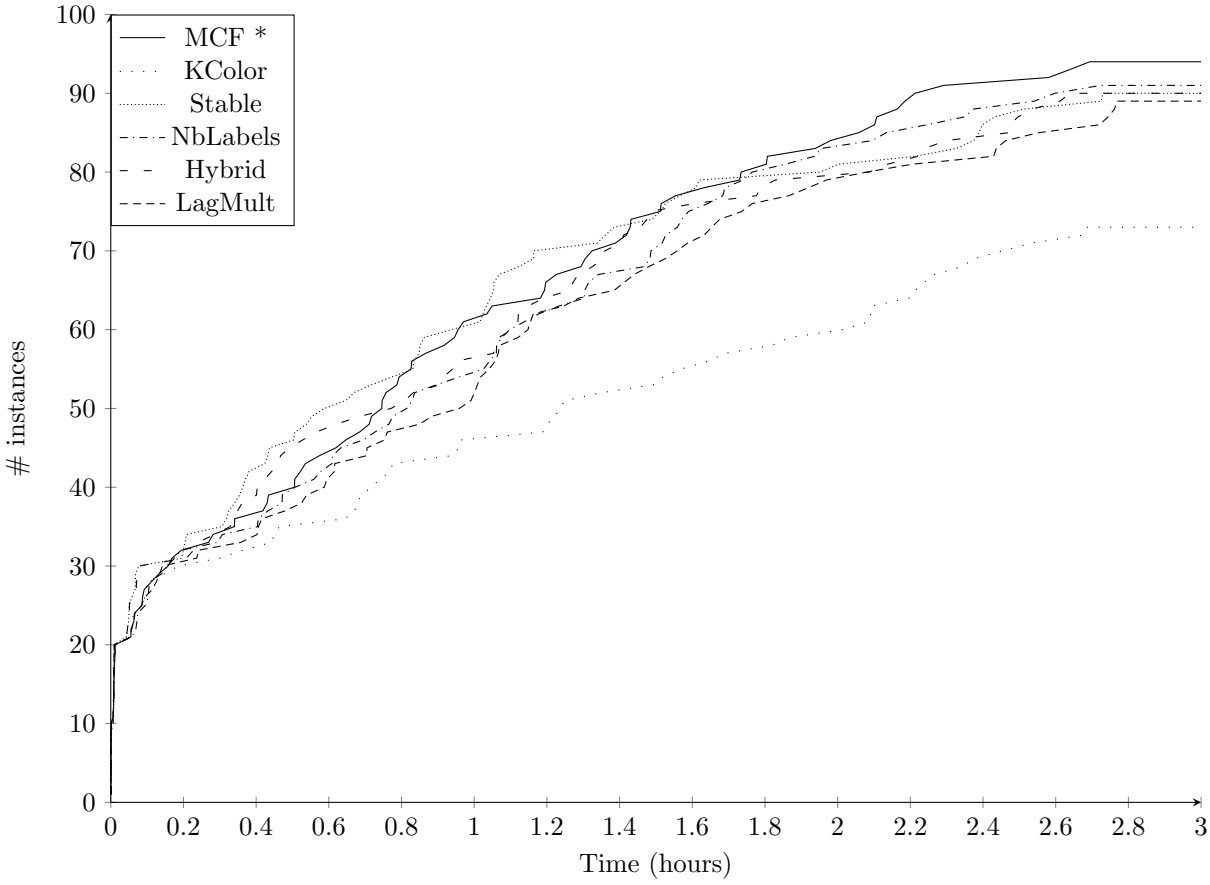

Figure 3: Number of instances solved along time for different methods used to determine the dimensions to add at each iteration.

Stable performs reasonably well within medium time limits. However, it does not seem to be more effective when more time is allocated. That can be explained by the fact that, the larger the network is, the less the number of new constraints added is controlled. Indeed, we empirically observed that only a few constraints are added in general by this method once a critical size is reached. The configurations based on Cardinality constrained stable set strategy do not suffer from that drawback. The performance of Hybrid configuration appears to be disappointing. This might be due to the difficulty of finding a good rule for switching between the basic criteria. Overall, an important conclusion is that taking into account the increase of the size of the network appears to be crucial for the method.

5.3 Comparison with the best results from the literature

We now compare our best results with those of [GI14] (denoted GI below). Note that [GI14] sent us the results obtained with a time limit equal to three hours. They implemented a pure branch-and-price without any primal heuristics and using best-first as node selection strategy. Their experiments were performed on a standard PC with an Intel(R) Core(TM) i7-2600 at 3.4 GHz with 16.0 GB main memory using a single thread only.

Figure 4 reports the performance of these methods compared to our best algorithm (MCF*), and a basic version of SSDP applied on TKP without the improvements we propose (MCF). It also shows how the ILP solver IBM Ilog Cplex performs on model (4)-(6) (CPLEX).

First, all instances but five were solved by at least one of the methods tested. In terms of number of instances solved after three hours, MCF* shows the best performance, followed by CPLEX applied to (4)–(6), GI, and MCF.

Applying SSDP in a straightforward fashion (MCF) turns out to be inefficient. Only 12 instances are solved after three hours of computing time. This confirms that all the improvements proposed in this document are needed to solve the problem efficiently using this methodology.

CPLEX applied to the first compact model outperforms all other methods for many instances, mostly

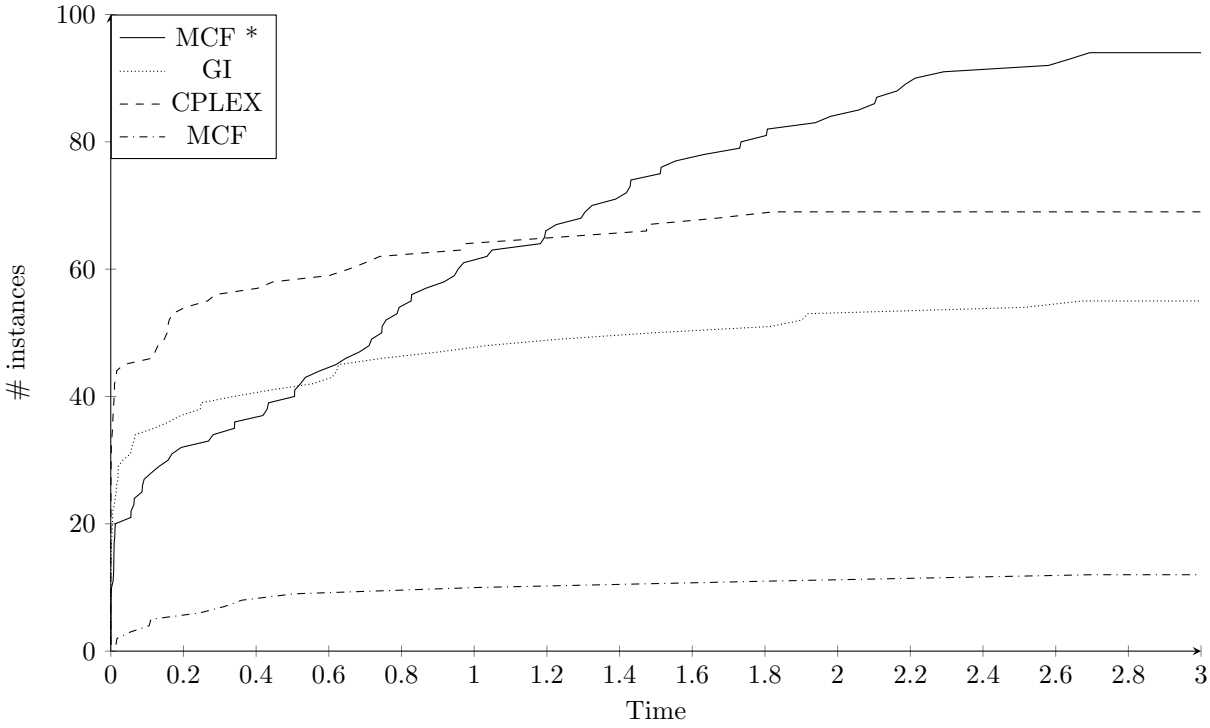

Figure 4: Number of instances solved along time for our best algorithm (MCF*), a basic version of SSDP (MCF), an ILP solver on model (4)-(6) (CPLEX) and the algorithm of [GI14] (GI).

instances numbered from 1 to 55. For most of these instances, it is able to solve the problem in a handful of seconds, while all other methods may need minutes or hours. That can be explained by the powerful procedures embedded in such solvers to deal with knapsack constraints (for example to derive cuts), as well as very good generic heuristics. From instance 55 to 99 however, CPLEX is only able to solve 16 instances. This can be explained by the structure of the instances. Each batch of ten consecutive instances has a similar structure, most notably the maximum number of items in a clique. This number increases with the index of the instances. It transpires from these experiments that linear-programming based methods are highly sensitive to this parameter.

Column generation is better than our method when the time is limited to 30 minutes. When one allows a large time, MCF* is able to solve much more instances than GI. The latter solves less instances than CPLEX applied to the compact model. Actually, after three hours of computation, the dual bound is still stricly less than the optimum.

After three hours, MCF* solved optimally 94 instances, which is 25 more than CPLEX. Only one instance is solved by CPLEX and not by MCF* (U69). On the contrary, MCF* is able to find 26 solutions when CPLEX does not reach convergence. Note that although the first twenty test cases of the data set are easier for our method, the computing time needed does not seem directly correlated to the size of the maximal cliques.

6 Conclusion

In this manuscript we have proposed a new algorithm for solving the temporal knapsack problem. It is based on an exponentially large dynamic program. We have shown that the latter can be solved efficiently using method SSDP. With the help of many refinements that we described, our method is able to solve significantly more instances than the methods of the literature. The strategies that we propose are subject to many parameters, and the numerical experiments suggest that better parameterization could yield better results (notably the Hybrid strategy). More generally, the most crucial ingredient is the choice of the constraints to add during each sublimation phase. Machine learning algorithms could be an option both for fine-tuning proposed approaches and guiding the choice of constraints in a more global way. Several techniques used in this manuscript could be easily adapted to other applications of SSDP. A generic library providing these features would be a useful tool for the community.

7 Acknowledgements.

We would like to thank Fabio Furini and Enrico Malaguti for sending us the detailed results of their experiments. We also would like to thank Timo Gschwind for running additional experiments for us.

This work was funded by Investments for the future Program IdEx Bordeaux, Cluster of Excellence SySNum.

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux, CNRS and ANR in accordance to the programme d'investissements d'Avenir (see `https://www.plafrim.fr/`).

References

[AS87] E. M. Arkin and E.B. Silverberg. Scheduling with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[BA00] F. Barahona and R. Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.

[BSW14] P.S. Bonsma, Jens Schulz, and Andreas Wiese. A constant-factor approximation algorithm for unsplittable flow on paths. *SIAM journal on computing*, 43(2):767–799, 2014.

[CCKR02] G. Calinescu, A. Chakrabarti, H.J. Karloff, and Y. Rabani. Improved approximation algorithms for resource allocation. In *Proceedings of the 9th International Conference on Integer Programming and Combinatorial Optimization, IPCO 2002*, page 401–414. Springer-Verlag, 2002.

[CFM13] A. Caprara, F. Furini, and E. Malaguti. Uncommon dantzig-wolfe reformulation for the temporal knapsack problem. *INFORMS Journal on Computing*, 25(3):560–571, 2013.

[CFMT16] A. Caprara, F. Furini, E. Malaguti, and E. Traversi. Solving the temporal knapsack problem via recursive dantzig–wolfe reformulation. *Information Processing Letters*, 116(5):379 – 386, 2016.

[CHT02] B. Chen, Refael Hassin, and Michal Tzur. Allocation of bandwidth and storage. *IIE Transactions*, 34(5):501–507, 2002.

[CSVV18] F. Clautiaux, R. Sadykov, F. Vanderbeck, and Q. Viaud. Combining dynamic programming with filtering to solve a four-stage two-dimensional guillotine-cut bounded knapsack problem. *Discrete Optimization*, 2018.

[Fra76] A. Frank. Some polynomial algorithms for certain graphs and hypergraphs. In *Proc. 5th Br. comb. Conf., Aberdeen 1975*, pages 211–226, 1976.

[GI14] T. Gschwind and S. Irnich. Stabilized column generation for the temporal knapsack problem using dual- optimal inequalities. Working Papers 1413, Gutenberg School of Management and Economics, Johannes Gutenberg-Universität Mainz, 2014.

[Iba87] Toshihide Ibaraki. Successive sublimation methods for dynamic programming computation. *Annals of Operations Research*, 11(1):397–439, December 1987.

[IN94] T. Ibaraki and Y. Nakamura. A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, 76(1):72 – 82, 1994.

[SV13] R. Sadykov and F. Vanderbeck. Bin packing with conflicts: a generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255, 2013.

[TFA09] Shunji Tanaka, Shuji Fujikuma, and Mituhiko Araki. An exact algorithm for single-machine scheduling without machine idle time. *Journal of Scheduling*, 12(6):575–593, December 2009.