An iterative dynamic programming approach for the temporal knapsack problem

François Clautiaux, Boris Detienne, Gaël Guillot

An iterative dynamic programming approach for the temporal knapsack problem

F. Clautiaux[1], B. Detienne[2], G.Guillot[3]

Université de Bordeaux, UMR CNRS 5251, Inria Bordeaux Sud-Ouest

Abstract

In this paper, we address the temporal knapsack problem (TKP). In this generalization of the classical knapsack problem, selected items enter and leave the knapsack at fixed dates. We solve the TKP with a dynamic program of exponential size, which is solved using a method called *Successive Sublimation Dynamic Programming* (SSDP). 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 effective method, and that several improvements are needed to compete with the best results from the literature.

Keywords: Temporal knapsack, Exact algorithm, Lagrangian Relaxation, Successive Sublimation Dynamic Programming method

1. Introduction

In this paper, we address the Temporal Knapsack Problem (TKP). The TKP is a generalization of the well-known knapsack problem, where the capacity constraint 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. Figure 1 represents an instance of TKP with three items. The name Temporal Knapsack was introduced in [1], although the problem had already been studied in [2] as a bandwidth allocation problem. Formally, the TKP can be stated as follows.

Problem 1 (Temporal Knapsack Problem). *Let $\mathcal{I} = \{1, \dots, n\}$ be a set of items. Each item $i \in \mathcal{I}$ has a profit $p_i \in \mathbb{R}^+$, a size $w_i \in \mathbb{N}$, and time interval $[s_i, f_i)$, where $s_i, f_i \in \mathbb{N}$ and $s_i < f_i$. Moreover, let $W \in \mathbb{N}$ be the weight of the knapsack. A feasible solution comprises a subset $\mathcal{J}$ of $\mathcal{I}$ such that for any value of $t \in \mathbb{N}$, the sum of the sizes of the items in $\mathcal{J}$ whose time interval contains t is less than W. The* Temporal Knapsack Problem *is the problem of finding a feasible subset $\mathcal{J}$ of $\mathcal{I}$ with maximum profit.*

In its general version, the TKP is NP-hard in the strong sense [3]. The first results proposed for TKP were focused on a theoretical characterization: a polynomially solvable case [4], and approximation results [2, 5]. Two dynamic programs were proposed by [2] and [6], which are described more precisely in the next section. The most recent works propose branch-and-price algorithms based on the Dantzig-Wolfe reformulation from [6]. The idea is to partition the time horizon into consecutive time periods (blocks). For each block, the variables related to the items whose time intervals intersect the corresponding time period are duplicated. Each subproblem is a smaller TKP, while the master problem makes sure that the duplicated variables related to the same item have the same value. These results were improved in [7] by using an innovative stabilization technique to improve the branch-and-price of [6]. This method relies on so-called dual-optimal inequalities, and uses dominance relations between (pairs of) items to add additional effective dual cuts that are satisfied by at least one optimal dual solution. Finally [8] proposed a method based on the previous

[1]francois.clautiaux@u-bordeaux.fr
[2]boris.detienne@u-bordeaux.fr
[3]gael.guillot@u-bordeaux.fr

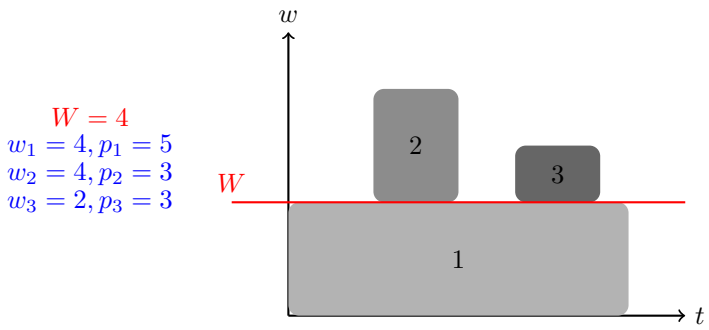

Figure 1: An instance of TKP with three items

Dantzig-Wolfe reformulation, where each subproblem is itself decomposed into a master problem and several smaller TKP, and solved by branch-and-price.

In this manuscript, we propose a new exact algorithm for TKP. It is based on an exponential size dynamic program, where the size of the state-space depends exponentially on the number of items n. Several methods in the literature have been proposed to tackle such large dynamic programs [9, 10]. All of them are based on the concept of *state-space relaxation* [11]. Among them, we selected *Successive Sublimation Dynamic Programming* (SSDP) method, originally proposed by [10], which has been successfully adapted to several one-machine scheduling problems (see [12] for example).

SSDP consists in solving a relaxation of the original dynamic program, removing some transitions that cannot belong to an optimal solution, 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).

As is the case with many generic methods, obtaining an effective version of SSDP for a new problem 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.

We implemented our algorithms and compared them empirically against a state-of-the-art MIP solver, using instances proposed in [6]. We also report results obtained by [7] on these instances. These experiments show that our algorithm is competitive compared to the best methods of the literature.

In Section 2, we formally discuss integer programming formulations and a recursive formulation for TKP. In Section 3, we describe an application of SSDP to TKP. Section 4 exposes the various refinements of the method that are 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 discuss compact MIP formulations for the problem. Then we describe the recursive formulation that we use.

2.1. Integer programming formulations

We first recall the commonly used integer programming formulation (see *e.g.* [6, 8]) 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. As suggested in [6], the capacity constraint needs to be satisfied only at starting time s_i of each item i.

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

$$\text{s.t.} \sum_{i \in \mathcal{I}: s_i \leq s_j < f_i} w_i x_i \leq W, \quad j \in \mathcal{I} \tag{2}$$

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

We now propose an alternative MIP formulation for TKP, which is not meant to be used directly to solve the problem, but simplifies the presentation of our dynamic program. In this model, we see the problem as a succession of events where decisions have to be taken (adding the item, or 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 index $i(e) \in \mathcal{I}$. We distinguish the events related to the beginning of the time window of an item (set $\mathcal{E}^{\text{in}}$) and those related to the end of the time window of an item ($\mathcal{E}^{\text{out}}$). Let also $\hat{t}(e)$ be the time slot related to event e, *i.e.* respectively $f_{i(e)}$ if $e \in \mathcal{E}^{\text{out}}$ and $s_{i(e)}$ 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.

The decisions of the new MIP model are related to these events. 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 $i(e)$ to the current solution. If $e \in \mathcal{E}^{\text{out}}$, the decision corresponds with removing $i(e)$ 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 total size of the selected items at the end of event e.

$$\max \sum_{e \in \mathcal{E}} \frac{1}{2} p_{i(e)} y_e \tag{4}$$

$$\phi_1 = w_{i(1)} y_1 \tag{5}$$

$$\phi_e = \phi_{e-1} + w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{in}} \setminus \{1\} \tag{6}$$

$$\phi_e = \phi_{e-1} - w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{out}} \tag{7}$$

$$\phi_e \leq W \quad e = 1, \ldots, 2n \tag{8}$$

$$\phi_{2n} = 0 \tag{9}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e') \tag{10}$$

$$y_e \in \{0, 1\}, \quad e = 1, \ldots, 2n \tag{11}$$

$$\phi_e \in \mathbb{R}_+, \quad e = 1, \ldots, 2n \tag{12}$$

The objective function is similar to that of model (1). 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 (5)–(7) ensure that the capacity consumption at the end of each event is consistent with the contents of the knapsack. Constraints (8) and (9) guarantee that the capacity constraints are satisfied. Constraints (10) state that if an item enters the knapsack, it has to leave it. We call constraints (10) *consistency constraints*. Note that constraint (9) is redundant when no other constraint is relaxed.

2.2. Dynamic programming formulations

To our knowledge, two dynamic programs for TKP were proposed in the literature. In [2], the authors proposed a dynamic programing algorithm where states are arranged in levels corresponding with events described above. A state $(e, \mathbf{d})$ is characterized by the current event $e \in \mathcal{E}$ and $\mathbf{d} \in \{0, 1\}^n$ the characteristic vector of the set of items currently in the knapsack. In [6], another dynamic program is proposed. It is based on a reformulation of the problem as a maximum profit path problem in an exponentially large graph. This graph has one layer for each knapsack constraint (2). In each layer, a vertex is created for each feasible subset of items that can be in the knapsack. Then, there is an arc between two nodes from two consecutive layers if and only if their contents are compatible (*i.e.* if an item belongs to the first subset, it has to belong to the second, and vice-versa). The cost of the arc is equal to the sum of the profits of the items that are added to obtain the new configuration. This method allows to solve the smallest instances of the literature, but according to the results presented by the authors, it cannot be applied when too many items can be packed at the same time instant, since the number of states in the dynamic program depends exponentially on this feature.

Our dynamic program is also based on the concept of *event*. It is an adaptation of [2] where we add a redundant information to the states (the capacity consumption), which will be useful in our relaxations. The model works similarly to model (4)–(12) 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.

We now describe formally the dynamic program using *states* and *transitions*. We define a *state* as a tuple $(e, w, \mathbf{d})$ where $e \in \mathcal{E}$ is the current event, $w \in \mathbb{Z}_+$ the current occupation of the knapsack, and $\mathbf{d} \in \{0,1\}^n$ the characteristic vector of the set of items currently in the knapsack. Note that w is redundant, since it can be deduced from vector $\mathbf{d}$. We call *transition* the possibility to pass from one state to another by taking a decision. A transition is defined by a tuple $(\Delta_e, \Delta_w, \Delta_{\mathbf{d}}, p)$ where $\Delta_e \in \mathbb{Z}_+$ describes the increase in the current event index, $\Delta_w \in \mathbb{Z}$ is the capacity consumed/released when the decision is taken, $\Delta_{\mathbf{d}} \in \{-1, 0, 1\}^n$ a vector that updates the content of the knapsack, and $p \in \mathbb{R}_+$ the profit obtained when the decision is taken.

The possible decisions that can be taken are defined by ψ, the function that associates to each state a set of feasible transitions. Let $\mathbf{0}$ be the null vector of size n, and $\boldsymbol{\varepsilon}_k \in \{0,1\}^n$ be the characteristic vector of set $\{k\}$, for $k \in \mathcal{I}$. For any feasible state $(e, w, \mathbf{d})$, function $\psi((e, w, \mathbf{d}))$ is computed as follows.

$$\psi((e,w,\mathbf{d})) = \begin{cases} \left\{(1,0,\mathbf{0},0),(1,w_{i(e)},\boldsymbol{\varepsilon}_{i(e)},\frac{1}{2}p_{i(e)})\right\} & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_{i(e)} \leq W \\ \{(1,0,\mathbf{0},0)\} & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_{i(e)} > W \\ \left\{(1,-w_{i(e)},-\boldsymbol{\varepsilon}_{i(e)},\frac{1}{2}p_{i(e)})\right\} & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 1 \\ \{(1,0,\mathbf{0},0)\} & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 0 \end{cases} \tag{13}$$

When an event $e \in \mathcal{E}^{\text{in}}$ is considered, two transitions are possible: one corresponding with selecting $i(e)$, the other with not selecting $i(e)$ (the former exists only if the remaining capacity is large enough). When an event $e \in \mathcal{E}^{\text{out}}$ is considered, only one transition is possible, depending on value $\mathbf{d}_{i(e)}$.

The cost function α from each state $(e, w, \mathbf{d})$ is then expressed in a backward recursive fashion. In the remainder of the paper, when two vectors are considered, symbols + and − respectively stand for the component-wise addition and substraction.

$$\alpha((e,w,\mathbf{d})) = \begin{cases} \max\limits_{(\Delta_e,\Delta_w,\Delta_{\mathbf{d}},p)\in\psi((e,w,\mathbf{d}))} \{p + \alpha((e+\Delta_e, w+\Delta_w, \mathbf{d}+\Delta_{\mathbf{d}}))\} & \text{if } e \in \{1,\ldots,2n\} \\ 0 & \text{if } e = 2n+1, w = 0, \boldsymbol{d} = \mathbf{0} \end{cases} \tag{14}$$

The optimal value of the TKP is $\alpha((1,0,\mathbf{0}))$.

3. Specializing Successive Sublimation Dynamic Programming to TKP

In this section, we explain how the generic method called SSDP can be used 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. Graph representation of the dynamic program

We first describe a graph representation of dynamic program (13), where states are represented by vertices, and transitions by arcs. The graph representation $G = (V, A)$ of the DP is obtained by creating a vertex for each possible reachable state of (13), and an arc for each possible transition. Each arc has the profit p of the corresponding transition. Starting from initial state $(1, 0, \mathbf{0})$, the nodes of the graph are created by computing recursively function $\psi(s)$ and creating the corresponding transitions to obtain the arcs and the vertices. Figure 2 illustrates the graph representation of DP (13) applied to the instance of Figure 1.

Once the graph representation of the DP is built, the problem can be solved by finding the path of largest profit between the vertex associated with $(1, 0, \mathbf{0})$ and the vertex associated with the final state $(n+1, 0, \mathbf{0})$. Since the graph has no circuits, Bellman's algorithm can used.

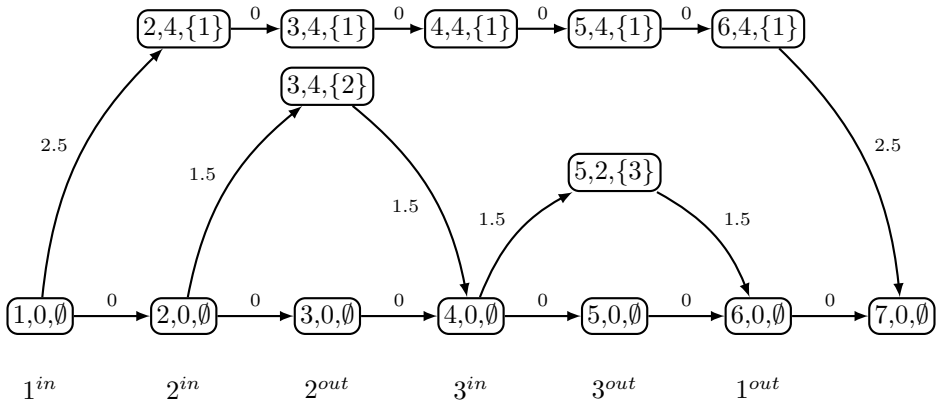

Figure 2: Graph representation of dynamic program (13) applied to instance of Figure 1.

3.2. Iterative state-space relaxation

Building the graph representation of DP (13) and using Bellman's algorithm cannot lead to any practical method. The size of the state-space in (13) 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 $O(n \times 2^n)$.

Several methods are used in the literature to solve dynamic programs with a large number of dimensions [9, 10]. All of them are based on the concept of state-space relaxation, introduced by [11]. The idea is to project the initial state-space onto lower dimensional space in such a way that a dual bound is obtained. Then, one computes several successive increasingly refined state-space relaxations until a stopping criterion is reached.

To our knowledge, the first method to use such a relaxation in an iterative algorithm was proposed by [10]. It is called *Successive Sublimation Dynamic Programming* (SSDP in the remainder). At each step of the algorithm, SSDP builds explicitly the graph representation of the DP expressed in the current relaxed state-space (called extended graph below). This extended graph, which can be exponentially large, is used to fix the value of some variables based on an evaluation of the cost of any path through the corresponding arcs. More details about this method are given in the next section.

Another iterative algorithm using iterative state-space was proposed by [9]. It is called *Decremental State Space Relaxation* (DSSR). To our knowledge, this method differs from SSDP in the way each relaxation is solved. The known implementations of DSSR do not build explicitly the extended graph, but use labelling algorithms instead. Several labels are kept for each vertex of the graph representation of the initial relaxation. DSSR has been applied successfully to solve resource-constraint shortest-path. A strength of this method is its ability to deal effectively with elementary constraints in routing problems, making use of strong dominance checks.

These two versions of iterative state-space relaxations have different advantages. On the one hand, DSSR allows more dominance checking than SDDP, since advanced label-setting/correcting techniques can be used. On the other hand, filtering techniques based on costs apply only to the initial graph, whereas SSDP filters arcs from the extended graphs corresponding to all intermediate relaxations.

For TKP, we decided to use SSDP for several reasons. First, our preliminary experiments have shown that the gap between the dual and primal bounds was good enough to filter a good percentage of arcs on many instances, and thus the extended graph does not grow too fast. Second, the dominance relations between two labels are weak for TKP, since one has to take into account the residual capacity that is freed by the items in the knapsack over the time.

3.3. Presentation of the generic algorithm

SSDP is a dual method that iteratively solves problems obtained by applying state-space relaxation to a dynamic program. An initial relaxation is obtained by relaxing constraints that cause the exponential

Algorithm 1: SSDP

Compute the graph related to the first relaxation.
Build graph G^0, the graph representation of the initial relaxation ;
$\ell \leftarrow 0$;
Solving the relaxation and filtering.
Solve the relaxation corresponding with graph G^ℓ to obtain a solution `sol` ;
if `sol` *is feasible and has a cost equal to some primal bound* **then return** `sol`;
Remove non-optimal states and transitions, obtaining graph $\hat{G}^\ell$;
Sublimation.
Construct the new graph $G^{\ell+1}$ from $\hat{G}^\ell$ by reintroducing new constraints ;
$\ell \leftarrow \ell + 1$;
go back to *step 4* ;

size of the state-space. A first dual bound is obtained by solving the relaxation. This dual bound is improved by refining the relaxation (*i.e.* reintroducing constraints), until the duality gap to a known primal bound is closed. The bound obtained at each step is possibly reinforced using a Lagrangian relaxation of the constraints. At each step of the algorithm, some unnecessary states and transitions are identified and removed from subsequently relaxations.

An important feature of the algorithm is that it constructs explicitly at each step the graph representation of the current dynamic program. This graph is used to record variable fixing information from a relaxation to the next. The main steps of the method are summarized in Algorithm 1.

Similarly to many generic frameworks, several ad-hoc key ingredients have to be designed for each new problem. 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.

3.4. Relaxation used for TKP

Our relaxation consists in not considering consistency constraints for some items. This is equivalent to consider only a subset of the values in $\mathbf{d}$ in the states, which reduces the size of the state-space. In this case, an item can enter the knapsack and not leave it, or the opposite.

Observation 1. *Projecting out vector* $\mathbf{d}$ *in* (13) *is equivalent to relaxing consistency constraints* (10) *in* (4)– (12).

We use a modified graph representation to apply the relaxation. At a given iteration of the algorithm, the relaxation is based on a set $\mathcal{J}$ of items that have to satisfy constraint (10). Let $G_\mathcal{J} = (V_\mathcal{J}, A_\mathcal{J})$ be the graph representation of the relaxed DP associated with $\mathcal{J}$. A vertex is now identified by a tuple $(e, w, \mathcal{C})$, where $\mathcal{C} \subseteq \mathcal{J}$ is the subset of items from $\mathcal{J}$ that are in the knapsack. Each vertex v represents a set of states $\mathcal{S}_\mathcal{J}(v)$ defined as follows: $\mathcal{S}_\mathcal{J}((e, w, \mathcal{C})) = \{(e', w', \mathbf{d}) : e' = e, w' = w, \forall i \in \mathcal{J}, d_i = 1 \leftrightarrow i \in \mathcal{C}\}$. For a given state $s = (e, w, \mathbf{d})$, we denote by $\hat{v}_\mathcal{J}(s)$ the unique vertex v such that $s \in \mathcal{S}_\mathcal{J}(v)$.

Given the modified graph representation, the relaxed dynamic program is obtained by the following recursive formulation that applies to the vertices of the graph.

$$\hat{\alpha}_\mathcal{J}((e, w, \mathcal{C})) = \max_{(\Delta_e, \Delta_w, \Delta_\mathbf{d}, p) \in \cup_{s \in \mathcal{S}_\mathcal{J}((e,w,\mathcal{C}))} \psi(s)} \{p + \hat{\alpha}_\mathcal{J}(\hat{v}_\mathcal{J}(e + \Delta_e, w + \Delta_w, \mathbf{d} + \Delta_\mathbf{d}))\} \tag{15}$$

For any arc a in $A_\mathcal{J}$, we denote by $\mu(a)$ the transition associated with arc a. An arc $a \in A_\mathcal{J}$ connects its tail $\tau(a)$ to its head $h(a)$. For a vertex v, let $\Gamma^+(v)$ be the set of out-going arcs, and $\Gamma^-(v)$ the set of in-going arcs.

Figure 3 depicts the graph representation of the relaxed version of the dynamic program when $\mathcal{J} = \emptyset$. The vertex represented by $(2, 4, \emptyset)$ represents two possibles states in the original dynamic program (with item 1 or with item 2). Consequently, two outgoing arcs have been built, each one related to the feasible transition that can be used from one of the two original DP states.

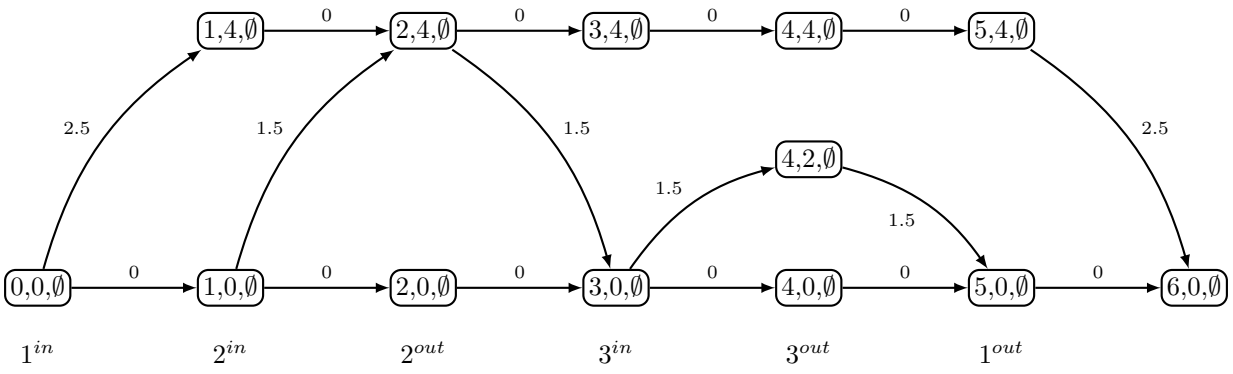

Figure 3: Graph representation of the relaxed DP with $\mathcal{J} = \emptyset$. Vertex labelled $(2, 4, \emptyset)$ represents the states of the original DP.

3.5. Solving the relaxation and filtering

We use Lagrangian relaxation to produce stronger bounds than combinatorial relaxation, while keeping the problem tractable. This method is used to compute a dual bound, and to remove some arcs that cannot belong to an optimal solution.

Let $\boldsymbol{\pi} \in \mathbb{R}^n$ be the vector of Lagrangian multipliers associated with Constraints (10) for indices $\mathcal{I} \setminus \mathcal{J}$. 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:

$$L_{\mathcal{J}}(\boldsymbol{\pi}) = \max \sum_{e \in \mathcal{E}^{in}} \left(\frac{1}{2} p_{i(e)} + \boldsymbol{\pi}_{i(e)} \right) y_e + \sum_{e \in \mathcal{E}^{out}} \left(\frac{1}{2} p_{i(e)} - \boldsymbol{\pi}_{i(e)} \right) y_e \tag{16}$$

$$(6) - (9), (11), (12) \tag{17}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e'), i(e) \in \mathcal{J} \tag{18}$$

For fixed $\mathcal{J}$ and any vector $\boldsymbol{\pi}$, $L_{\mathcal{J}}(\boldsymbol{\pi})$ is an upper bound on the optimum of (4)-(12). To compute a good bound using this relaxation, we need to solve 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, function $L_{\mathcal{J}}(\boldsymbol{\pi})$ is known to be convex, which implies that minimizing this function can be done using a subgradient algorithm, or one of its refinements).

We solve the Lagrangian dual problem using Volume algorithm proposed in [13]. This approximate method builds a sequence of solutions $\boldsymbol{\pi}$ that converges to the optimum. For each value of $\boldsymbol{\pi}$, $L_{\mathcal{J}}(\boldsymbol{\pi})$ is computed by applying Bellman's algorithm on graph $(V_{\mathcal{J}}, A_{\mathcal{J}})$, where the profits of the arcs are modified to take into account the penalization of the relaxed constraints. More precisely, for each arc a such that $\mu(a) = (\Delta_e, \Delta_w, \Delta_{\mathbf{d}}, p)$, the profit of a is now $p+ < \boldsymbol{\pi}, \Delta_{\mathbf{d}} >$. In what follows, we call $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ the graph $G_{\mathcal{J}}$ with the costs modified by the Lagrangian multipliers $\boldsymbol{\pi}$. We denote by $v^0 = (1, 0, \emptyset)$ the vertex representing the initial state, and by $v^{\Omega} = (2n+1, 0, \emptyset)$ the vertex representing the terminal state. We denote by $\alpha^{\boldsymbol{\pi}}_{\mathcal{J}}((e, w, \mathcal{C}))$ the value of Bellman's function for vertex $(e, w, \mathcal{C})$. The value of $L_{\mathcal{J}}(\boldsymbol{\pi})$ is the maximum cost of a path from v^0 to v^{Ω}. Solving the Lagrangian subproblem has a complexity in $O(|V_{\mathcal{J}}| + |A_{\mathcal{J}}|)$ using Bellman's algorithm.

Figure 4 illustrates how Lagrangian multipliers are added to the profits of the arcs to account for the Lagrangian cost (we omit π_2, whose value does not impact the value of any $v^0 - v^{\Omega}$ path).

Observation 2. *Problem* (4)-(12) *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* (4)-(12) *with the same cost since the contributions of Lagrangian multipliers cancel out.*

Now, we recall a result used in [10, 14] to remove unnecessary vertices and arcs from $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ (and thus the corresponding states and transitions). For this purpose, let us remark that for any node $(e, w, \mathcal{C}) \in V_{\mathcal{J}}$, Bellman function value $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}((e, w, \mathcal{C}))$ is equal to the maximum cost of a path in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ from $(e, w, \mathcal{C})$ to v^{Ω}.

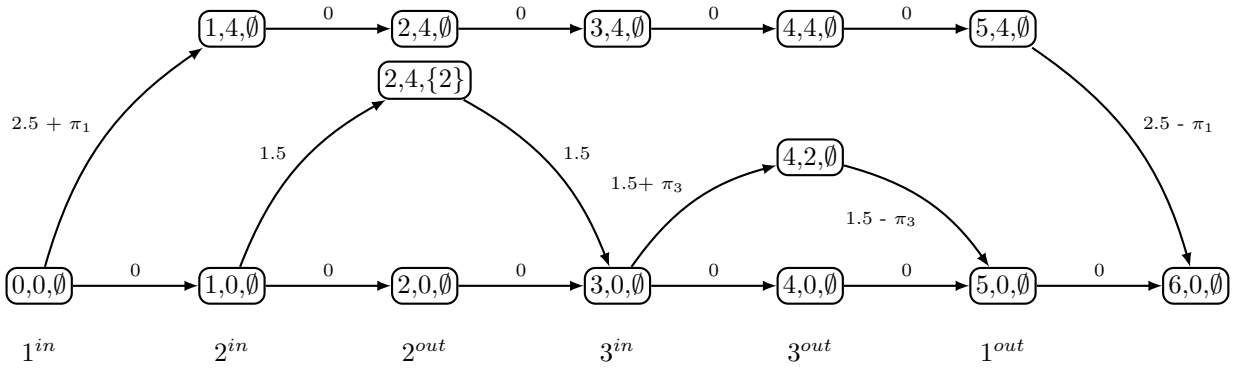

Figure 4: Graph representation of the relaxed dynamic program with $\mathcal{J} = \{2\}$, including Lagrangian costs.

Likewise, we define $\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}}((e, w, \mathcal{C}))$ as the maximum cost of a path from v^0 to $(e, w, \mathcal{C})$, which can be computed in a similar way.

Proposition 1 ([10]). *For $\mathcal{J} \subseteq \mathcal{I}$, let $a \in A_{\mathcal{J}}$, such that $\mu(a) = (\Delta_e, \Delta_w, \Delta_{\mathbf{d}}, p)$, between two vertices $(e^1, w^1, \mathcal{C}^1)$ and $(e^2, w^2, \mathcal{C}^2)$, and $\boldsymbol{\pi} \in \mathbb{R}^n$. The following value is an upper bound on the cost of any path in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ that uses arc a:*

$$\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}}((e^1, w^1, \mathcal{C}^1)) + p + < \boldsymbol{\pi}, \Delta_{\mathbf{d}} > + \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}((e^2, w^2, \mathcal{C}^2))$$

This result allows us to remove unnecessary transitions from graph $G_{\mathcal{J}}$: if the upper bound for arc a is lower than a known lower bound for the problem, then arc a and the related transition cannot be in an optimal solution of the relaxation defined by $\mathcal{J}$. Moreover, the efficiency of SSDP lies in the fact that the corresponding arcs in subsequent, more precise, relaxations can be removed as well. However, to our knowledge, the validity of this permanent removal is only implicitly assumed in the literature. We give a formal proof of this feature in our specific context, in appendix (Proposition 8).

Values $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}((e, w, \mathcal{C}))$ and $\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}}((e, w, \mathcal{C}))$ can be computed for all nodes $(e, w, \mathcal{C}) \in V_{\mathcal{J}}$ in two passes using respectively Bellman's forward and backward dynamic programming algorithm.

If an arc is filtered from a graph $G^{\boldsymbol{\pi}}_{\mathcal{J}}$, it is filtered in the graph representation $(V_{\mathcal{J}}, A_{\mathcal{J}})$, and all vertices with no predecessors or no successors are removed from $V_{\mathcal{J}}$. The corresponding states and transitions will not be considered in subsequent iterations. For any state $s = (e, w, \mathbf{d})$, we define $\rho_{\mathcal{J}}$ the function that associates to s the set of transitions that were not filtered up to the iteration related to set $\mathcal{J}$.

$$\rho_{\mathcal{J}}((e, w, \mathbf{d})) = \begin{cases} \emptyset & \text{if } \hat{v}((e, w, \mathbf{d})) \notin V_{\mathcal{J}} \\ \{\mu(a) : a \in \Gamma^+(\hat{v}(e, w, \mathbf{d}))\} & \text{otherwise} \end{cases}$$

3.6. Sublimation and convergence

In SSDP, the sublimation phase consists in strengthening the current relaxation by enforcing some constraints that are violated in the current solution. In our application, the set $\mathcal{J}$ of consistency constraints taken into account in the DP is extended by adding new ones, defining $\mathcal{K} \supset \mathcal{J}$. The sublimation phase builds a new graph $G_{\mathcal{K}}$ from filtered graph $G_{\mathcal{J}}$ using the following modified transition function:

$$\hat{\psi}^-_{\mathcal{K}}((e, w, \mathbf{d})) = \psi((e, w, \mathbf{d})) \cap \rho_{\mathcal{J}}((e, w, \mathbf{d}))$$

The maximum number of iterations of the algorithm is n, since at least one item index is added to $\mathcal{J}$ at each sublimation step and when $\mathcal{J} = \mathcal{I}$, the relaxation obtained is actually equivalent to (4)-(12) (Observation 2). 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 (16)-(18) is equal to its cost in (4)-(12), 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 effectively 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 by much when only one constraint at a time is reintroduced in the sublimation phase. We now propose several techniques 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 indices of constraints that are currently taken into account in the dynamic program. For a given vertex $v = (e, w, \mathcal{C})$ such that $e \in \mathcal{E}^{\text{out}}$, if $i(e) \notin \mathcal{J}$, two transitions are possible: $(+1, -w_{i(e)}, -\boldsymbol{\varepsilon}_{i(e)}, \frac{1}{2}p_{i(e)})$ that removes $i(e)$ from the knapsack, and $(+1, 0, \mathbf{0}, 0)$ that just goes to the next event. However, in some cases, in any path $(a_1, a_2, \ldots, a_\ell)$ from v^0 to v, there is no arc a_j such that $\mu(a_j) = (+1, +w_{i(e)}, +\boldsymbol{\varepsilon}_{i(e)}, \frac{1}{2}p_{i(e)})$ (*i.e.* adding item $i(e)$). The opposite case also happens (item $i(e)$ is never added). In both cases, only one transition should be created.

We attach to each vertex v an additional vector $\mathbf{d}^\eta(s) \in \{0, 1, \emptyset\}^n$. If $\mathbf{d}^\eta(v)_i = \emptyset$, v represents states where i is in the knapsack, and some where i is not. If $\mathbf{d}^\eta(v)_i = 0$, v represents only states where i is not in the knapsack, while $\mathbf{d}^\eta(v)_i = 1$ means that v represents only states where i is in the knapsack. For $i \notin \mathcal{J}$, we set $\mathbf{d}^\eta(v^0)_i = 0$, and we compute $\mathbf{d}^\eta(v)_i$ recursively for each other vertex v as follows:

$$\mathbf{d}^\eta(v)_i = \begin{cases} 1 & \text{if } \forall a \in \Gamma^-(v), \quad \mathbf{d}^\eta(\tau(a))_i = 1 \text{ or } \mu(a) = (+1, \Delta_w, +\boldsymbol{\varepsilon}_i, p) \\ 0 & \text{if } \forall a \in \Gamma^-(v), \quad \mathbf{d}^\eta(\tau(a))_i = 0 \text{ or } \mu(a) = (+1, \Delta_w, -\boldsymbol{\varepsilon}_i, p) \\ \emptyset & \text{otherwise} \end{cases}$$

Practically speaking, vector $\mathbf{d}^\eta(v)$ is computed on the fly while $(V_\mathcal{J}, A_\mathcal{J})$ is created. We attach another information to each vertex v, which corresponds with redundant constraints. For each vertex $v = (e, w, \mathcal{C})$, we define $q^\eta_{\min}(v)$ (resp. $q^\eta_{\max}(v)$) as a lower (resp. upper) bound on the number of items that can be in the knapsack in the states of $\mathcal{S}_\mathcal{J}(v)$. These values can be computed recursively, similarly to vector $\mathbf{d}^\eta(v)$.

Let $\mathcal{I}(e) = \{i \in \mathcal{I} : s_i \leq \hat{t}(e) < f_i\}$ be the set of items that may belong to the knapsack when e occurs. 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 vertex $v = (e, w, \mathcal{C})$, the number of items in any valid state represented by v is in $[0, Q^{\max}(e)]$.

4.2. 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 transitions following recurrence equations (13). We denote by $\mathcal{S}^+$ the set of feasible states. We recall that $V_\mathcal{J}$ is the set of vertices considered while solving the relaxation related to $\mathcal{J}$. A sufficient condition for the global solving process to be valid is that for all $\mathcal{J} \subseteq \mathcal{I}$, there is a path in $G_\mathcal{J}$ whose arcs form an optimal sequence of transitions. Thus, any state in $V_\mathcal{J}$ that is not related to a feasible state in S can be removed from the graph 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 2. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $v = (e, w, \mathcal{C}) \in V_\mathcal{J}$. If card$(\{i \in \mathcal{I} : \mathbf{d}^\eta(v)_i \neq 0\}) < q^\eta_{\min}(v)$ or card$(\{i \in \mathcal{I} : \mathbf{d}^\eta(v)_i = 1\}) > q^\eta_{\max}(v)$ then $\mathcal{S}(v) \cap \mathcal{S}^+ = \emptyset$.*

Another feasibility test is based on the set of possible weights of subsets of items that can belong to the 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 (3) follows from the definition of $\mathcal{F}$.

Proposition 3. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $v = (e, w, \mathcal{C}) \in V_\mathcal{J}$. If $w \notin \mathcal{F}(e)$ then $\mathcal{S}(v) \cap \mathcal{S}^+ = \emptyset$.*

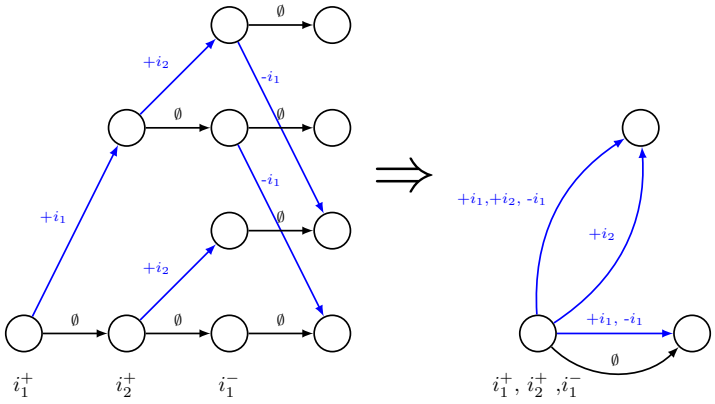

Figure 5: Partial enumeration of three events. Four sequences out of eight are feasible.

This rule can be improved by considering additional information gathered from $\mathbf{d}^\eta(s)$. We precompute, for each event e and each item $i \in \mathcal{I}(e)$, $\mathcal{F}^+(e,i)$ 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 $\mathcal{F}^+(e,i) = \{\sum_{j\in S\cup\{i\}} w_j : S \subseteq \mathcal{I}(e)\setminus\{i\}, \sum_{j\in S} w_j \leq W - w_i\}$ and $\mathcal{F}^-(e,i) = \{\sum_{j\in S} w_j : S \subseteq \mathcal{I}(e)\setminus\{i\}, \sum_{j\in S} w_j \leq W\}$.

Proposition 4. *Let $\mathcal{J} \subseteq \mathcal{I}$, $v = (e,w,\mathcal{C}) \in V_\mathcal{J}$ and $i \in \mathcal{I}(e)$. If $\mathbf{d}^\eta(v)_i = 1$ and $w \notin \mathcal{F}^+(e,i)$ then $\mathcal{S}(v)\cap\mathcal{S}^+ = \emptyset$. Likewise, if $\mathbf{d}^\eta(v)_i = 0$ and $w \notin \mathcal{F}^-(e,i)$ $\mathcal{S}(v)\cap\mathcal{S}^+ = \emptyset$.*

The following result allows detecting nodes related to states that can be generated only by 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 5. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $v = (e,w,\mathcal{C}) \in V_\mathcal{J}$. If $\sum_{i\in\mathcal{I}:\mathbf{d}^\eta(v)_i=1} w_i > w$ then $\mathcal{S}(v)\cap\mathcal{S}^+ = \emptyset$.*

For a vertex $(e,w,\mathcal{C})$, the following feasibility test integrates the bounds on the number of items in the knapsack to ensure the consistency of set $\mathcal{C}$ with respect to value w.

Proposition 6. *Let $\mathcal{J} \subseteq \mathcal{I}$ and $v = (e,w,\mathcal{C}) \in V_\mathcal{J}$. Let $S^1 = \{i \in \mathcal{I}(e) : \mathbf{d}^\eta(v)_i \neq 0\}$ be the set of items potentially in the knapsack. If $\max\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \leq q^\eta_{\max}(v)\right\} < w$ or $\min\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \geq q^\eta_{\min}(v)\right\} > w$ then $\mathcal{S}(v)\cap\mathcal{S}^+ = \emptyset$.*

4.3. Partial enumeration of transitions

Combining several consecutive transitions into single compound transitions 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 transitions. More precisely, from a given state, instead of computing the two possible transitions, we compute the $O(2^{k^{\text{enum}}})$ possible sequences of k^{enum} successive transitions. Sequences that violate consistency constraints are not generated. Figure 5 illustrates a case where three consecutive events are considered.

4.4. Criteria for selecting constraints to reintroduce

At each sublimation step, one has to select the dimensions related to violated constraints that are integrated into the state-space. Let us denote $\boldsymbol{\pi}^q$ the vector of Lagrangian multipliers that achieves the q^{th} best dual bound during the current relaxation solution step, and let $\boldsymbol{y}^q$ be the solution found when solving (16)-(18) to compute $L_\mathcal{J}(\boldsymbol{\pi}^q)$ (expressed with the variables of (16)-(18)). 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 $\mathcal{J}$.

Note that, since the magnitude of the violation of constraints is always one in our relaxation, it is not significant when it comes to selecting a relaxed constraint to be reintroduced, contrary to applications of SSDP in the context of scheduling. We thus describe three criteria for estimating the computational attractiveness of adding a specific constraint.

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: $\psi_i^1 = |\boldsymbol{\pi}_i^1|$.

The second criterion (**Network Size**) aims at controling the number of states in the network after the sublimation step. For each constraint, we compute an estimation of the growth of the vertex set if the corresponding constraint – and only that one – is included into the state-space. Remark that, when adding a single constraint related to item 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: $\psi_i^2 = -|v \in V_\mathcal{J} : \mathbf{d}^\eta(v)_i = \emptyset|$.

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: $\psi_i^3 = \frac{1}{k^{\text{nbsol}}} \sum_{q=1}^{k^{\text{nbsol}}} \left| y^q_{e^{in}(i)} - y^q_{e^{out}(i)} \right|$.

4.5. Reintroducing batches of constraints

Adding several violated constraints at once is generally a good strategy for TKP. The sublimation step is a time-consuming procedure, and preliminary experiments have shown that in many cases, adding only one constraint is not sufficient to ensure a significant decrease in the dual bound. However, adding too many constraints increases dramatically the size of the network. Therefore, we have to find a good trade-off between the quality of the bound and the size of the network.

A first issue is to compute a reliable estimation of the size of the network when a new constraint is introduced. 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 7. *Let $\mathcal{J}$ and $\mathcal{K}$ such that $\mathcal{J} \subseteq \mathcal{K} \subseteq \mathcal{I}$. It holds that $|V_\mathcal{K}| \leq \sum_{e \in \mathcal{E}} \left(2^{|(\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e)|} \, card(\{(e', w, \mathcal{C}) \in V_\mathcal{J} : e' = e\})\right)$.*

Proof. The set of vertices created in $V_\mathcal{K}$ from a given vertex $(e, w, \mathcal{C}) \in V_\mathcal{J}$ is included in the set

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

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 indicates that adding constraints related to items with pairwise disjoint time windows is particularly attractive: in such cases, for all events $e \in \mathcal{E}$ we have $\text{card}((\mathcal{K} \setminus \mathcal{J}) \cap \mathcal{I}(e)) \leq 1$. It follows that the network grows by a constant factor only, as stated formally in the next corollary. Let $G^{\text{int}} = (\mathcal{I}, E^{\text{int}})$ be the interval graph related to intervals $[s_i, f_i), i \in \mathcal{I}$, and $G^{\text{int}}_{\neq}$ the subgraph of $G^{\text{int}} = (\mathcal{J}^{\neq}, E^{\text{int}}_{\neq})$ induced by $\mathcal{J}^{\neq}$ (which is also an interval graph).

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 corollary guided our strategies to select the set of constraints that are added at each sublimation step. We propose four strategy that aim at finding a good tradeoff between the approximate growth of the network and the quality of the relaxation.

The first strategy, that we call **Weighted stable set**, consists in adding a set of constraints related to 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 MAXG). 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). In the model below, a binary variable x_i is created for all $i \in \mathcal{J}^{\neq}$, indicating whether i is selected or not. The problem of constraint selection can be cast as follows.

$$\max\{\sum_{i \in \mathcal{J}^{\neq}} \psi_i x_i : \sum_{i \in \mathcal{J}^{\neq}} \psi_i^2 x_i \leq \text{MAXG}, x_i + x_j \leq 1 \ \forall (i,j) \in E^{\text{int}}_{\neq}, x_i \in \{0,1\} \ \forall i \in \mathcal{J}^{\neq}\}$$

This problem is a disjunctive knapsack with conflicts, which is NP-complete, but can be solved in pseudo-polynomial time through dynamic programming for interval graphs (see [15]). For the instances we considered, the time needed to solve this subproblem is negligible compared to the overall time of the algorithm.

Our second strategy, called **Cardinality Constrained Stable Set**, aims at circumventing a major drawback of the Weighted stable set strategy, which sometimes adds too few new constraints, leading to a slow convergence of the overall algorithm. Our strategy 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. Using the same variables as the model above, one obtains the following model.

$$\max\{\sum_{i\in\mathcal{J}^{\neq}} \psi_i x_i : \sum_{i\in\mathcal{J}^{\neq}} x_i \geq C_{\max} * k^{\text{rstable}}, x_i + x_j \leq 1 \ \forall (i,j) \in E^{\text{int}}_{\neq}, x_i \in \{0,1\} \ \forall i \in \mathcal{J}^{\neq}\}$$

This problem is also NP-complete, but is solved effectively by modern MILP solvers (*i.e.* the time needed to solve it is also negligible compared to the overall time of the algorithm).

The third strategy, called k**-coloring**, not only considers stable sets in $G^{\text{int}}_{\neq}$, but in a larger graph representing also constraints that were added in the previous iterations (set $\mathcal{J}$ below). Thus we consider the subgraph $G^{\text{int}}_{+} = (\mathcal{J}^{\neq} \cup \mathcal{J}, E^{\text{int}}_{+})$ of G^{int} induced by $\mathcal{J}^{\neq} \cup \mathcal{J}$. Let k^{color} be equal to the number of colors that we allow at the current step. At each iteration, we seek a k^{color}-colorable subgraph of G^{int}_{+} of maximum weight. 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. Initially, k^{color} is set to one. We solve repeatedly the following model, where z_{ij} are binary variables indicating that item i is assigned to color j.

$$\begin{aligned}
\max \sum_{i\in\mathcal{J}^{\neq}} \sum_{j=1,\ldots,k^{\text{color}}} & \psi_i z_{ij} \\
z_{ij} + z_{\ell j} &\leq 1, \forall (i,\ell) \in E^{\text{int}}_{+}, j \in \{1,\ldots,k^{\text{color}}\} \\
\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} &\leq 1, \forall i \in \mathcal{J}^{\neq} \\
\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} &= 1, \forall i \in \mathcal{J}^{+} \\
z_{ij} &\in \{0,1\}, \forall i \in \mathcal{J}^{\neq} \cup \mathcal{J}^{+}, j \in \{1,\ldots,k^{\text{color}}\}
\end{aligned}$$

We also solve this subproblem with a general purpose MILP solver. Again, the time is negligible compared to the time needed to build the new graph at each iteration.

Finally, we consider a strategy called **Hybrid** that favours a strong improvement in the gap in the first iterations, and then favours a network of manageable size when its size reaches a threshold. This is based on the following observation: in the first iterations, one would like to close the gap between the primal and dual bounds a fast as possible to allow a better performance of filtering procedure, but 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, the hybrid strategy uses one of the strategies above in the first iterations, and when the size of the network is larger than a given threshold, the choice is only based on the expected size of the network.

4.6. Implementation issues

The effectiveness of the filtering step heavily depends 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 heuristically solve model (1)–(3) by giving a small amount of time to a general purpose integer linear programming solver.

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

Table 1: 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 arcs	1,392 k	1,344 k	1,639 k	2,269 k	3,155 k	4,658 k

relaxation of (4)–(10), and use the optimal dual values of constraints (10). Solving the LP relaxation is fast and provides a good starting point for Volume.

We implemented a parallel version of Bellman's algorithm. We first compute the longest path (in terms 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 [7] and to the results obtained using 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 [6], composed of two groups. For instances in the first group (I), results are not reported since [7] 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 algorithm, and the impact of the different improvements that we have proposed. We also want to assess their effectiveness 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

We first 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 partial enumeration technique (subsection 4.3) and dominance and feasibility tests (subsection 4.2). For each method, we report in Figure 6 the number of instances solved along the time, with a limit of three hours.

The number of instances solved optimally within 3 hours increases by about 20% when partial enumeration is used. This means that enumerating sequences of transitions allows to include useful information that is used by the filtering algorithm. To illustrate the effect of partial enumeration, we report in Table 1 the size of the first network constructed, for different values of k^{enum}. As one can expect, increasing the number of consecutive transitions considered reduces the number of nodes in the network but increases the number of arcs (since combinations of arcs are replaced by single arcs). 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 arcs 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 $k^{\text{enum}} = 2$ has fewer arcs than that with $k^{\text{enum}} = 1$.

The feasibility tests proposed in Propositions 2 to 6 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 $k^{\text{enum}} = 4$.

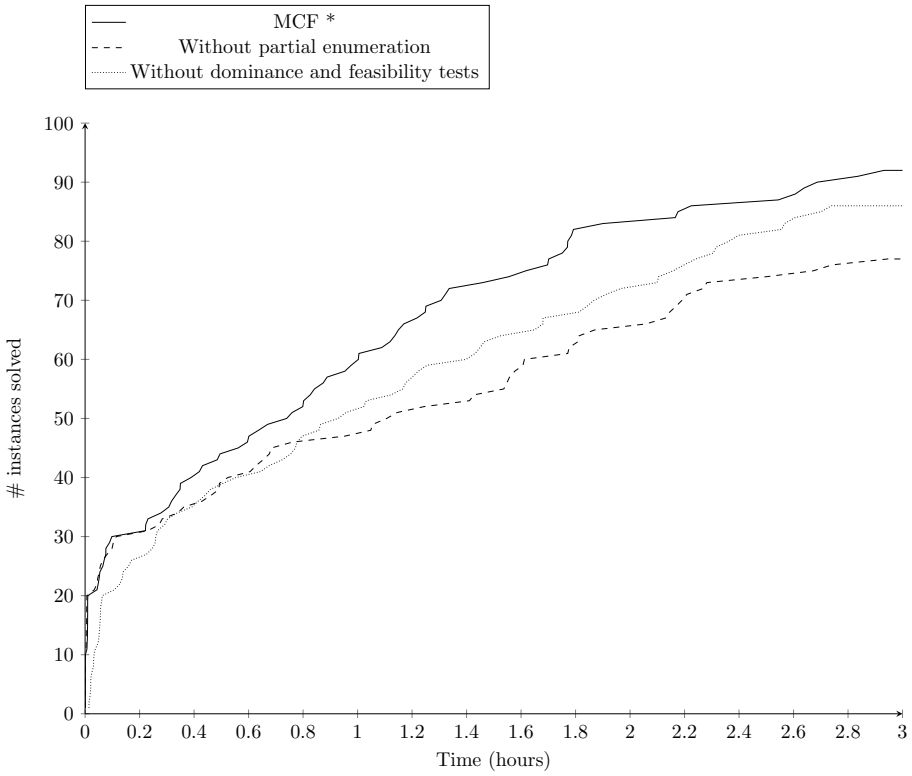

Figure 6: Number of instances of TKP from data set U ([8]) solved along time, for the best version of our algorithm, and two versions obtained by deactivating some techniques.

5.2. Strategies for reintroducing constraints

As stated in most papers working on iterative state-space relaxations, the selection of the constraints to reintroduce is the most critical component in the method. In what follows, we empirically compare our different strategies to determine the most effective. In Table 2, we report how each configuration of our algorithm 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.

Figure 7 numerically compares our different methods for selecting the constraints to add during the sublimation phase. Similarly to Figure 6, we report the total number of instances solved along the time, with a limit of three hours. 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 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 is disappointing. This might be due to the difficulty of finding a good rule for switching between the two criteria. Overall, an important conclusion is that taking into account the increase of the size of the network

Table 2: 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	$\frac{-\psi_i^2+\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}{\max_{j\in\mathcal{J}^{\neq}}\psi_j^2}\psi_i^3$

appears to be crucial for the method.

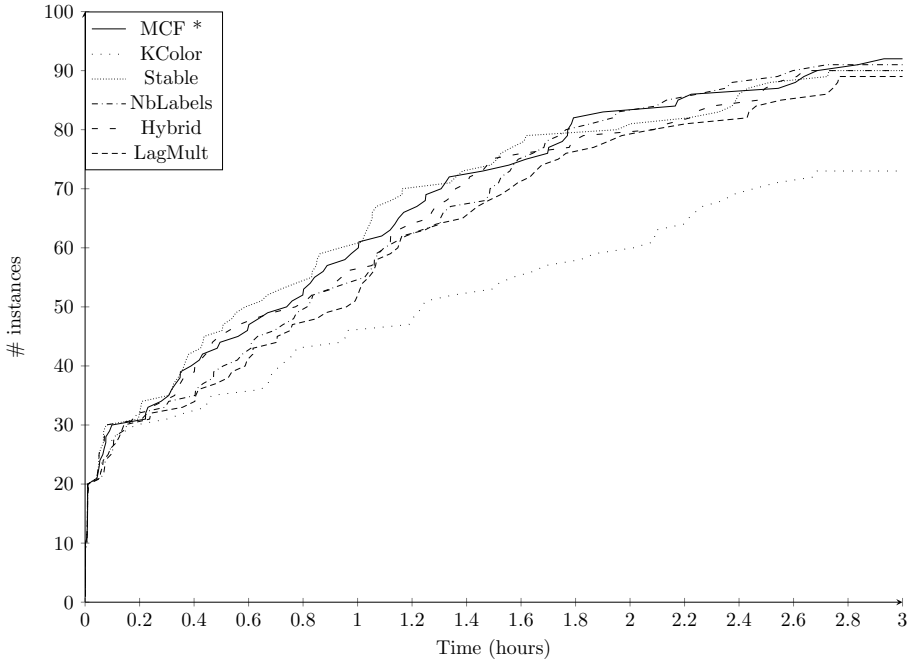

Figure 7: Number of instances solved along time for different methods used to determine the constraints to add at each iteration.

5.3. Comparison with the branch-and-price of [7]

We now compare our method with the algorithm of [7]. The authors have kindly provided us the results obtained with their algorithm within a three-hour time limit. 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 8 reports the performance of our best algorithm (MCF*) and those obtained by [7](GI) using a similar computer (same processor, same amount of RAM), using a single thread only. The processor speeds in this setting and in the one described at the beginning of Section 5 are roughly comparable. However, the limited amount of memory on this machine is not always enough for our method (the algorithm ran out of memory for eight instances that are solved on the other machine).

The approach of [7] is more efficient within short computing time: it solves 41 instances in 30 minutes, when the best version of our algorithm, when restricted to a single thread on the same machine, solves only 35 instances. Both algorithms perform similarly within a one-hour time limit (47 instances solved versus 49). However, only a few more instances are solved by the branch-and-price approach within 3 hours of

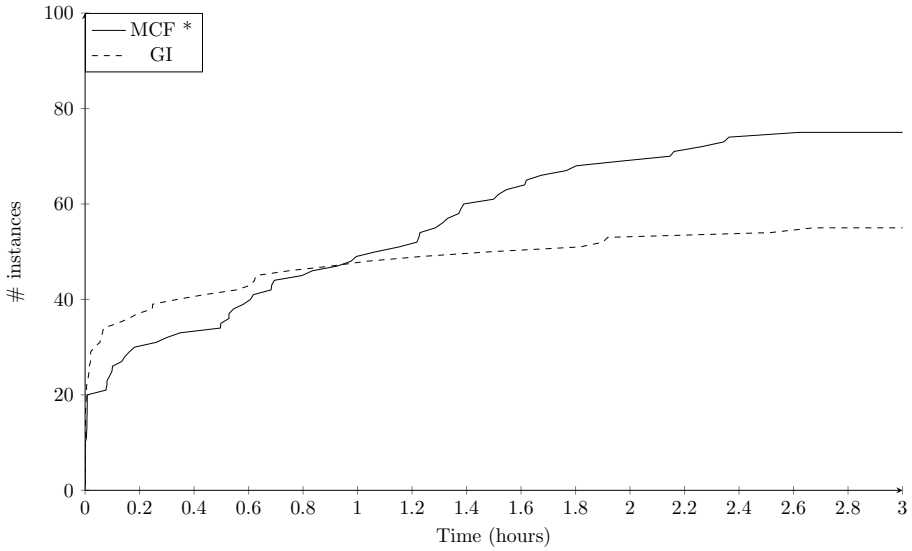

Figure 8: Number of instances solved along time for our best algorithm (MCF*) and the algorithm of [7] on Intel(R) Core(TM) i7-2600 at 3.4 GHz with 16.0 GB main memory using a single thread only.

computing time (55 instances in total), while our approach solves 50 percent of the still unsolved instances between 1 and 3 hours (75 instances in total).

5.4. Comparison with a general-purpose MILP solver

Figure 9 reports the performance of our best algorithm (MCF*), and those obtained by ILP solver IBM Ilog Cplex when solving model (1)-(3) (CPLEX). We limited both methods to a single thread.

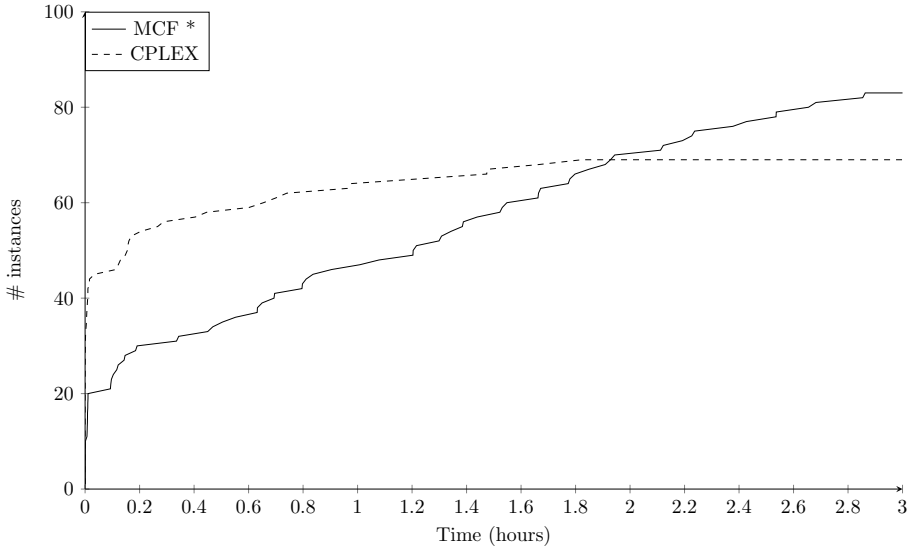

Figure 9: Number of instances solved along time for our best algorithm (MCF*) and an ILP solver solving model (1)-(3) (CPLEX).

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 (1)–(3) and MCF. After three hours, MCF* solves optimally 83 instances, which is 14 more than CPLEX.

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

We now report the performances of CPLEX and our method in their multiple thread setting. We are not aware of any parallel implementation of the branch-and-price procedure of [7]. Figure 10 reports the results with six threads for MCF* and CPLEX. Using more cores is useful for both methods. After three hours, MCF* with six threads solved optimally 94 instances, which is 12 more than CPLEX. Our method is not six times faster, since only the Lagrangian problem solver is parallelized. The time needed to construct and update the graph representation of the dynamic program represents a large percentage of the total running time, and this part of the algorithm does not benefit from a multi-core architecture. Only two instances are solved by CPLEX and not by MCF* (U69 and U78). On the contrary, MCF* is able to find 14 solutions when CPLEX does not reach convergence.

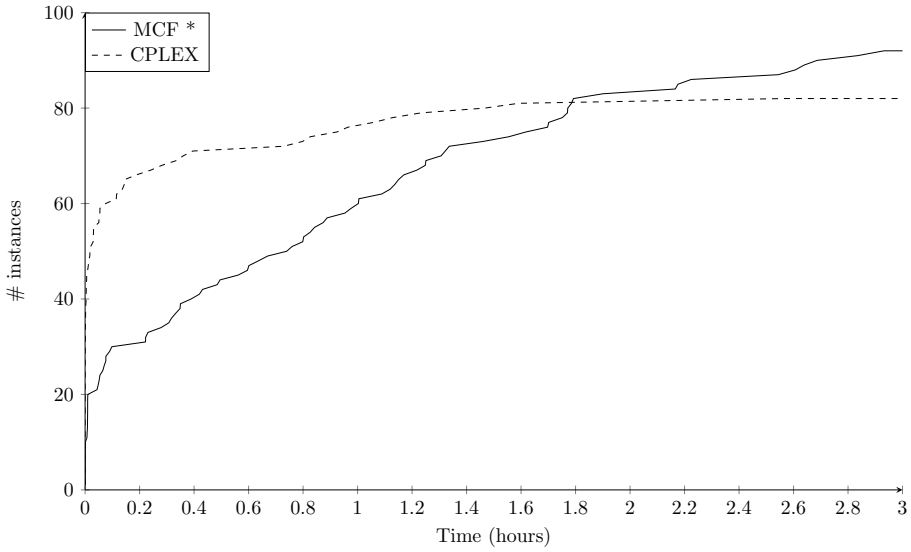

Figure 10: Number of instances solved along time for our best algorithm (MCF*) using 6 threads and an ILP solver on model (1)-(3) (CPLEX) using 6 threads.

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 effectively using method SSDP. With the help of several refinements that we described, our method is able to obtain results that are competitive with those 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). 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 selection of constraints in a more global way. Several techniques used in this manuscript could be easily adapted to other applications of SSDP. We plan to develop a generic library providing these features 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

[1] M. Bartlett, A. M. Frisch, Y. Hamadi, I. Miguel, S. A. Tarim, C. Unsworth, The Temporal Knapsack Problem and Its Solution, in: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Vol. 3524, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 34–48. doi:10.1007/11493853_5.
URL http://link.springer.com/10.1007/11493853_5

[2] B. Chen, R. Hassin, M. Tzur, Allocation of bandwidth and storage, IIE Transactions 34 (5) (2002) 501–507. doi:10.1023/A:1013535723204.
URL http://dx.doi.org/10.1023/A:1013535723204

[3] P. Bonsma, J. Schulz, A. Wiese, A constant-factor approximation algorithm for unsplittable flow on paths, SIAM journal on computing 43 (2) (2014) 767–799. doi:10.1137/120868360.

[4] E. M. Arkin, E. Silverberg, Scheduling with fixed start and end times, Discrete Applied Mathematics 18 (1987) 1–8.

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

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

[7] T. Gschwind, S. Irnich, Stabilized column generation for the temporal knapsack problem using dual-optimal inequalities, OR Spectrum 39 (2) (2017) 541–556. doi:10.1007/s00291-016-0463-x.
URL https://doi.org/10.1007/s00291-016-0463-x

[8] A. Caprara, F. Furini, E. Malaguti, E. Traversi, Solving the temporal knapsack problem via recursive dantzig–wolfe reformulation, Information Processing Letters 116 (5) (2016) 379 – 386. doi:http://dx.doi.org/10.1016/j.ipl.2016.01.008.
URL http://www.sciencedirect.com/science/article/pii/S0020019016000107

[9] G. Righini, M. Salani, New dynamic programming algorithms for the resource constrained elementary shortest path problem, Networks 51 (2008) 155 – 170. doi:10.1002/net.20212.

[10] T. Ibaraki, Successive sublimation methods for dynamic programming computation, Annals of Operations Research 11 (1) (1987) 397–439. doi:10.1007/BF02188549.
URL https://doi.org/10.1007/BF02188549

[11] N. Christofides, A. Mingozzi, P. Toth, State-space relaxation procedures for the computation of bounds to routing problems, Networks 11 (2) (1981) 145–164. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230110207, doi:10.1002/net.3230110207.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230110207

[12] S. Tanaka, S. Fujikuma, M. Araki, An exact algorithm for single-machine scheduling without machine idle time, Journal of Scheduling 12 (6) (2009) 575–593. doi:10.1007/s10951-008-0093-5.
URL http://link.springer.com/10.1007/s10951-008-0093-5

[13] F. Barahona, R. Anbil, The volume algorithm: producing primal solutions with a subgradient method, Mathematical Programming 87 (3) (2000) 385–399. doi:10.1007/s101070050002.
URL https://doi.org/10.1007/s101070050002

[14] T. Ibaraki, Y. Nakamura, A dynamic programming method for single machine scheduling, European Journal of Operational Research 76 (1) (1994) 72 – 82. doi:http://dx.doi.org/10.1016/0377-2217(94)90007-8.
URL http://www.sciencedirect.com/science/article/pii/0377221794900078

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

Appendix

The following lemma shows that the set of arcs going out of each vertex of a refined relaxation related to $\mathcal{J}$' is a subset of the arcs going out of the vertex, in the relaxation related to $\mathcal{J}$, it comes from through sublimation.

Lemma 1. *Let us consider $e \in \mathcal{E}$, $w \in \{0,\dots,W\}$, $\mathcal{J}$ and $\mathcal{J}$' such that $\mathcal{J} \subseteq \mathcal{J}' \subseteq \mathcal{I}$, $v = (e,w,\mathcal{C}) \in V_{\mathcal{J}}$ and $v' = (e,w,\mathcal{C}') \in V_{\mathcal{J}'}$ such that $\mathcal{C} \cap \mathcal{J} = \mathcal{C}' \cap \mathcal{J}$ (i.e. vertex v' comes from the sublimation of vertex v). Then $\cup_{s \in \mathcal{S}_{\mathcal{J}}((e,w,\mathcal{C}))}\psi(s) \supseteq \cup_{s \in \mathcal{S}_{\mathcal{J}'}((e,w,\mathcal{C}'))}\psi(s)$.*

Proof. Let $(e,w,\mathbf{d}) \in S_{\mathcal{J}'}(e,w,\mathcal{C}')$. Then by definition of $S_{\mathcal{J}}$, for all $i \in \mathcal{J}'$, $d_i = 1 \leftrightarrow i \in \mathcal{C}'$. Since $\mathcal{J} \subset \mathcal{J}'$, for all $i \in \mathcal{J}$, $d_i = 1 \leftrightarrow i \in \mathcal{C}'$, and so $d_i = 1 \leftrightarrow i \in \mathcal{C}$ because $\mathcal{C} \cap \mathcal{J} = \mathcal{C}' \cap \mathcal{J}$. Thus $(e,w,\mathbf{d}) \in S_{\mathcal{J}'}(e,w,\mathcal{C})$, from which the result follows. □

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

Lemma 2. *Let us consider $e \in \mathcal{E}$, $w \in \{0,\dots,W\}$, $\mathcal{J}$ and $\mathcal{J}$' such that $\mathcal{J} \subseteq \mathcal{J}' \subseteq \mathcal{I}$, $v = (e,w,\mathcal{C}) \in V_{\mathcal{J}}$ and $v' = (e,w,\mathcal{C}') \in V_{\mathcal{J}'}$ such that $\mathcal{C} \cap \mathcal{J} = \mathcal{C}' \cap \mathcal{J}$ (i.e. vertex v' comes from the sublimation of vertex v) and $\boldsymbol{\pi} \in \mathbb{R}^n$ a vector of Lagrangian multipliers. Then $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e,w,\mathcal{C}) \geq \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e,w,\mathcal{C}')$ and $\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}}(e,w,\mathcal{C}) \geq \hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e,w,\mathcal{C}')$.*

Proof. We proceed by induction on e to prove the part of the proposition involving $\hat{\alpha}$. A straightforward adaptation of the proof yields the result for $\hat{\gamma}$. At rank $e = 2n+1$, the property is satisfied since we have $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e,0,\mathcal{C}) = \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e,0,\mathcal{C}') = 0$ and $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e,w,\mathcal{C}) = \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e,w,\mathcal{C}') = -\infty$ if $w \neq 0$, $\mathcal{C} \neq \emptyset$ or $\mathcal{C}' \neq \emptyset$.

At rank $e \in \{1,\dots,2n\}$, assume that $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e+1,\bar{w},\bar{\mathcal{C}}) \geq \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e+1,\bar{w},\bar{\mathcal{C}}')$ for all $(e+1,\bar{w},\bar{\mathcal{C}}) \in V_{\mathcal{J}}$ and $(e+1,\bar{w},\bar{\mathcal{C}}') \in V_{\mathcal{J}'}$ such that $\bar{w} \in \{0,\dots,W\}$ and $\bar{\mathcal{C}} \cap \mathcal{J} = \bar{\mathcal{C}}' \cap \mathcal{J}$. Then for all $(e,w,\mathcal{C}) \in V_{\mathcal{J}}$ and $(e,w,\mathcal{C}') \in V_{\mathcal{J}'}$ such that $w \in \{0,\dots,W\}$ and $\mathcal{C} \cap \mathcal{J} = \mathcal{C}' \cap \mathcal{J}$, and for all $(\Delta_e,\Delta_w,\Delta_{\mathbf{d}},p) \in \cup_{s \in \mathcal{S}_{\mathcal{J}}((e,w,\mathcal{C}))}\psi(s)$, we have

$$\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e+\Delta_e, w+\Delta_w, \mathcal{C}_+) \geq \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e+\Delta_e, w+\Delta_w, \mathcal{C}'_+)$$

with $\mathcal{C}_+ = \mathcal{C} \cup \{i \in \mathcal{I} : (\Delta_d)_i = 1\} \setminus \{i \in \mathcal{I} : (\Delta_d)_i = -1\}$ and $\mathcal{C}'_+ = \mathcal{C}' \cup \{i \in \mathcal{I} : (\Delta_d)_i = 1\} \setminus \{i \in \mathcal{I} : (\Delta_d)_i = -1\}$. Indeed, $\mathcal{C}_+ \cap \mathcal{J} = \mathcal{C}'_+ \cap \mathcal{J}$. From (15) and Lemma 1, we have that $\hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e,w,\mathcal{C}) \geq \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e,w,\mathcal{C}')$. □

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 8. *Let LB be a valid lower bound for the problem, $\mathcal{J} \subseteq \mathcal{I}$, $a \in A_{\mathcal{J}}$ such that $\tau(a) = (e^1,w^1,\mathcal{C}^1)$, $h(a) = (e^2,w^2,\mathcal{C}^2)$, $\mu(a) = (\Delta_e,\Delta_w,\Delta_{\mathbf{d}},p)$, and $\boldsymbol{\pi} \in \mathbb{R}^n$. If $\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}}(e^1,w^1,\mathcal{C}^1) + < \Delta_{\mathbf{d}}, \boldsymbol{\pi} > + \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}}(e^2,w^2,\mathcal{C}^2) < LB$, then the shortest path problem in $G^{\boldsymbol{\pi}}_{\mathcal{J}}$ without arc a is a relaxation of (4)-(12). Moreover, for any $\mathcal{J}' \supseteq \mathcal{J}$ and $\boldsymbol{\pi}' \in \mathbb{R}^n$, the shortest path problem $G^{\boldsymbol{\pi}'}_{\mathcal{J}'}$ without any arc related to transition $\mu(a)$ from states $(e^1,w^1,\mathcal{C}^{1'})$ such that $\mathcal{C}^{1'} \cap \mathcal{J} = \mathcal{C}^1 \cap \mathcal{J}$ is a relaxation of (4)-(12) as well.*

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 $b \in \mathcal{A}_{\mathcal{J}'}$, such that $\mu(a) = \mu(b)$, $\tau(b) = (e^1, w^1, \mathcal{C}^{1'})$ such that $\mathcal{C}^{1'} \cap \mathcal{J} = \mathcal{C}^1 \cap \mathcal{J}$. Then $h(b) = (e^2, w^2, \mathcal{C}^{2'})$, such that $\mathcal{C}^{2'} \cap \mathcal{J} = \mathcal{C}^2 \cap \mathcal{J}$. Indeed, $\mathcal{C}^{2'} \cap \mathcal{J} = (\mathcal{C}^{1'} \cap \mathcal{J}) \cup (\{i \in \mathcal{I} : (\Delta_d)_i = 1\} \cap \mathcal{J}) \setminus \{i \in \mathcal{I} : (\Delta_d)_i = -1\} = \mathcal{C}^1 \cup (\{i \in \mathcal{I} : (\Delta_d)_i = 1\} \cap \mathcal{J}) \setminus \{i \in \mathcal{I} : (\Delta_d)_i = -1\} = \mathcal{C}^2 \cap \mathcal{J}$. Hence Lemma 2 implies that $\hat{\gamma}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e^1, w^1, \mathcal{C}^{1'}) + < \Delta_{\mathbf{d}}, \boldsymbol{\pi} > + \hat{\alpha}^{\boldsymbol{\pi}}_{\mathcal{J}'}(e^2, w^2, \mathcal{C}^{2'}) < LB$. Arc b 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 b can be removed from $G^{\boldsymbol{\pi}}_{\mathcal{J}'}$ that shall still define a relaxation of (4)-(12).

Second, we prove the validity of the proposition for a fixed set of consistency constraints $\mathcal{J}$ and a different vector of multipliers $\boldsymbol{\pi}'$. Lemma 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 a in $G^{\boldsymbol{\pi}'}_{\mathcal{J}}$, $\boldsymbol{\pi}' \in \mathbb{R}^n$ can be optimal for the problem, and removing a does not remove optimal solutions of the problem from those relaxations. □