RAIRO-Oper. Res. 57 (2023) 1–16
https://doi.org/10.1051/ro/2022022

RAIRO Operations Research
www.rairo-ro.org

FULLY POLYNOMIAL TIME APPROXIMATION SCHEME FOR THE PAGINATION PROBLEM WITH HIERARCHICAL STRUCTURE OF TILES

Aristide Grange, Imed Kacem*, Sébastien Martin and Sarah Minich

Abstract. The pagination problem is described as follows. We have a set of symbols, a collection of subsets over these symbols (we call these subsets the tiles) and an integer capacity C. The objective is to find the minimal number of pages (a type of container) on which we can schedule all the tiles while following two fundamental rules. We cannot assign more than C symbols on each page and a tile cannot be broken into several pieces, all of its symbols must be assigned to at least one of the pages. The difference from the Bin Packing Problem is that tiles can merge. If two tiles share a subset of symbols and if they are assigned to the same page, this subset will be assigned only once to the page (and not several times). In this paper, as this problem is NP-complete, we will consider a particular case of the dual problem, where we have exactly two pages for which the capacity must be minimized. We will present a fully polynomial time approximation scheme (FPTAS) to solve it. This approximation scheme is based on the simplification of a dynamic programming algorithm and it has a strongly polynomial time complexity. The conducted numerical experiments show its practical effectiveness.

Mathematics Subject Classification. 68W25.

Received November 11, 2021. Accepted February 5, 2022.

1. General description of the subproblem

The considered problem in this paper is based on the one introduced in the work by Sindelar, Sitaraman and Shenoy in [14]. In their paper, the fundamental problem is to assign a set of virtual machines (VM) to physical servers. A virtual machine is composed of different memory pages (of different sizes) describing (among other information) its operating system (iOS, Windows or Linux), the OS version run on the VM (Windows XP, Windows 7, Ubuntu, ...), the architectural version of the libraries the VM are going to use (32 or 64 bits), the different software programs that will be installed in the VM and so on until the set of memory pages describes a real-functioning-virtual machine. A VM is therefore a set of memory pages and the size of a VM is the sum of the sizes of its memory pages. Now, two VM can have a subset of memory pages in common. If these VM are assigned to the same physical server, then they can share these pages. It implies that the common pages do not have to be repeated: they are assigned only once on the server. In this case, we say that the VMs can merge. This is the particularity that separates the pagination problem from the famous Bin Packing Problem.

However, the general description of the pagination problem is less VM-oriented, this is why the vocabulary we use for it is more general. It can be described as follows. We are given a set of symbols $\mathcal{S}$ (corresponding to

Keywords. Approximation algorithms, scheduling, FPTAS, dynamic programming.

Université de Lorraine, LCOMS, Metz, France.
*Corresponding author: imed.kacem@univ-lorraine.fr

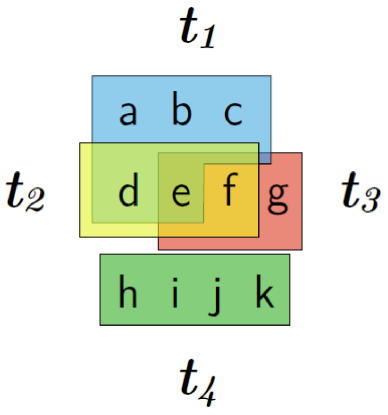

FIGURE 1. An example of an input for the Pagination Problem – extracted from [4].

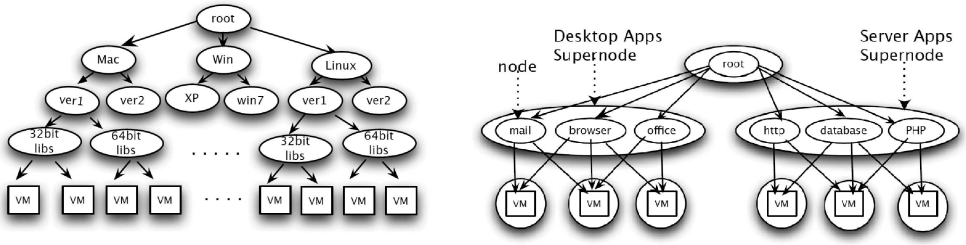

FIGURE 2. Two examples of models, the tree model (*left*) and the cluster-tree model (*right*) – extracted from [14].

the former memory pages). All the symbols have the same size which is equal to 1. We are also given a collection of subsets of these symbols called the tiles $\mathcal{T}$ (former virtual machines) and an integer capacity (C). The size of a tile is the number of symbols it contains. We have to find the minimal number of pages (former physical servers), which can contain all tiles without exceeding a maximal number of C symbols per page. The other fundamental and inherent rule of the problem is that a tile cannot be broken: all its symbols must be assigned to at least one page to be able to claim that the tile has been allocated to the page. The NP-hardness proof of this problem is developed in [5].

In Figure 1 (extracted from [4]), we can see an example of an input for the Pagination Problem. There are four different tiles and three of them partially merge. For example, tile $t_1 = \{a, b, c, d, e\}$ fuses with tile $t_2 = \{d, e, f\}$ as the intersection between t_1 and t_2 is not empty: $t_1 \cap t_2 = \{d, e\}$.

It is worth-mentioning again that we exclude the case where a tile would be entirely contained in another one as this situation does not present any interest (it can be easily detected and avoided).

In [14], the authors claimed that *"[. . .] The inter-VM sharing occurs largely in hierarchical fashion"* and they proposed two hierarchical sharing models to represent this organisation: a model based on a tree and a more general model based on a cluster-tree. We can see two illustrative examples in Figure 2 (both illustrations are from the article of [14]).

In this paper, we consider that the tiles are organised according to the tree model. In the tree representation of Figure 2, each node contains one piece of information, *i.e.*, one symbol. However, by generalising having multiple symbols per node, we create the most generic inputs for our subproblem. That is why from now on, we will present the tree model representation with numbers in the nodes. We label the nodes using a breadth-first

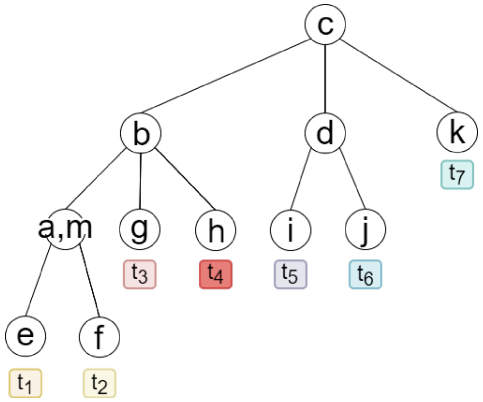

FIGURE 3. Example of an input for the 2 $|\,\text{merging}, \text{tree}\,|\ C_{\max}$ where one node contains two symbols.

algorithm, which starts with the root. Each number $i \in [1, 2, \dots N]$ (with N the number of nodes in the tree) will represent the name of a node and $p_i \in \mathbf{R}$ will be equal to the number of symbols contained by a node, also called the size of the node. For example, in Figure 3, the node containing symbols a and m will be labelled *node* 5 and its size p_5 will be equal to 2.

There will be as much leaves as there are tiles in the input. We denote by m the number of leaves in the tree (the size of a tile is equal to the sum of the corresponding nodes' sizes). We can also verify that $|\mathcal{S}| = \sum_{i=1}^{N} p_i$ and here again, each symbol is contained by only one node.

It is easy to observe that the well-known 2-PARTITION problem can be reduced to our problem $2\ |\,\text{merging}, \text{tree}\,|\ C_{\max}$. Recall that the 2-PARTITION problem consists in determining if a set of N given integers $(a_1, a_2, \dots, a_N)$ can be partitioned into two disjoint subsets S_1 and S_2 such that $\sum_{i \in S_1} a_i = \sum_{i \in S_2} a_i$ (for further information, see [11]. Indeed, we can consider an input where we have a tree with one root and $m = N$ leaves (*i.e.*, m tiles). Every tile i is associated to the root ($p_1 = 0$) and to its specific leaf (containing its proper $p_{i+1} = a_i$ symbols, with $i = 1, 2, \dots, m$). These hypotheses have several consequences. First, the only symbol shared by all tiles is put in the root of the tree. Then, as the tiles do not share any other symbol, all the vertices apart from the root will have only one child.

This example illustrates how one can transform an input of 2-PARTITION problem to an equivalent input of our problem and thus perform the polynomial reduction 2-PARTITION $\prec_p 2\ |\,\text{merging}, \text{tree}\,|\ C_{\max}$. Such a reduction proves that $2\ |\,\text{merging}, \text{tree}\,|\ C_{\max}$ is NP-complete.

2. Dynamic programming algorithm

2.1. Introduction

In order to use the dynamic programming (DP) principle, we have to define what a state of this algorithm is and then we have to establish the recursive relationship we will use in it. A state will have four integer components $[a, b, j, k]$:

- a: number of symbols assigned to page $P1$
- b: number of symbols assigned to page $P2$
- j: index of last tile on $P1$
- k: index of last tile on $P2$

The second step in the design of a dynamic programming (DP) algorithm is to devise the recursive relationship. In our case, it is simple: the relationship shows the choice we have to make when we schedule a tile. This

choice can be presented as a question: shall we assign the current tile to page $P1$ or to page $P2$? In the DP algorithm displayed in Algorithm 1, both choices are explored (see lines 6 and 7).

2.2. Pre-process

As the main idea in our algorithm is based on the tree representing how tiles share symbols, it is natural to compute the tiles following an order coming from that tree. We will begin with the tile ending with the left-most leaf (when $i = 1$) and then we will continue with the tile ending with the leaf just to the right of the previous one ($i = 2$) and so on until the tiles are all handled (when we arrive at the right-most tile with $i = m$). Of course, we can process the tiles in the opposite order (starting with the right-most tile and going toward the left-most one) with the same algorithm.

Before actually presenting the pseudocode of the DP algorithm, we need to introduce one more thing we use in it: a matrix M. A cell in the matrix represents the number of symbols we need to add to a page when we want to assign a tile to it. Of course, this number depends on the tile we are considering but it also depends on the tiles already scheduled on this page.

Thanks to the hierarchy into the tiles and if the tiles are processed in a correct order (which is described above), it is easy to see that we only need to know the last tile assigned to a page in order to know how many symbols we will have to add to this page to schedule the new tile.

Let us study a quick example. Let $\mathcal{P}$ be one of our two pages and let us say tile j is the last tile assigned to page $\mathcal{P}$. The value $\mathsf{M}(i, j)$ represents the number of symbols we have to add to page $\mathcal{P}$ as a consequence of scheduling tile i.

The matrix is filled in a pre-process before the beginning of the dynamic programming algorithm that is why this step is ignored in Algorithm 1. This is also the reason why this computation is not taken into account later in the time complexity of the DP algorithm.

2.3. The algorithm

Algorithm 1: The dynamic programming algorithm.

Input: The set of tiles $\mathcal{T}$ and the set of symbols $\mathcal{S}$
Output: The optimal value

```
1  begin
2      χ_0 ← {[0, 0, 0, 0]}
3      Fill matrix M
4      for i from 1 to m do
5          foreach [a, b, j, k] in χ_{i−1} do
6              χ_i ← χ_i ∪ {[a + M(j, i), b, i, k]}
7              χ_i ← χ_i ∪ {[a, b + M(k, i), j, i]}
8          foreach [a, b, j, k] in χ_i do
9              if Several quadruplets have the same values for a, j, k then
10                 Keep only the one having the smallest b
11         Delete set χ_{i−1}
12 return min_{[a,b,j,k]∈χ_m} {max{a, b}}
```

Complexity. Recall that P is $\sum_{i=1}^{\mathcal{N}} p_i$. Thanks to the dominating rule, there is only one possible value for b while there are $P + 1$ possible different values for a and at most m different values for j and k. From this, we know that we generate at most $\mathcal{O}(m^2.P)$ different states during one iteration. As there are m iterations in total, this gives us a time complexity of $\mathcal{O}(m^3.P)$ in a worst-case scenario analysis.

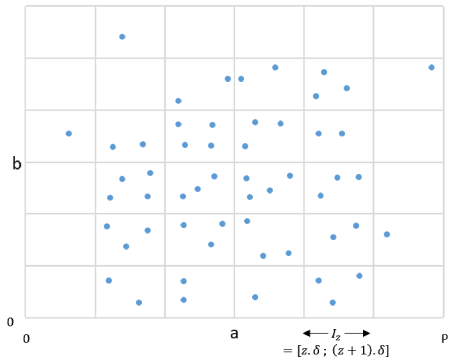

FIGURE 4. All the states in χ_i displayed in a chequered 2-dimension space.

3. FPTAS BASED ON DP

3.1. Principe

Observation. We already used this property earlier but both a and b have P as maximum value such as: $a \leq P$ and $b \leq P$. Furthermore, the best (but maybe infeasible) relaxed solution we can obtain is splitting the set of symbols $\mathcal{S}$ into two subsets having exactly the same size and scheduling the first subset on $P1$ and the other one on $P2$. This implies that $\text{OPT} \geq \frac{P}{2}$.

The principle of the FPTAS we are going to present is to decrease the time complexity of the dynamic programming algorithm thanks to the reduction of the number of states we keep at each generation while assuring a certain quality in the solution we find at the end.

Steps. For one iteration i, we can represent all the states generated in χ_i in a 2-dimension coordinate system. The x-axis (resp. y-axis) represents the different values that a (resp. b) can take.

First, we divide both axes (each going from 0 to P) into subintervals defined as follows: $I_z = [(z-1)\cdot\delta;\ z\cdot\delta\ ]$ with $z = 1, 2, \ldots, \frac{2m}{\epsilon}$. It implies a division of the space in squares. Then, we apply a modified algorithm based on the previous DP. For each iteration, the states of χ_i are ranked according to their value a. For each interval I_z, we will keep the state with the smallest a-value as a representative for all states being the same interval. If two representatives are possible (if two states have the same a-value), we will keep the one with the smallest b-value. Such a representative is denoted by $[a^\sharp, b^\sharp, j, k]$ and stored in the set $\chi_i^\#$.

This simplification technique is inspired from [8], in which the state space is divided into rectangular sub-spaces (based on upper and lower bounds) and a specific representative state for each sub-space is kept at each iteration of the FPTAS. Another successful example can be found in [9]. Other geometrical forms, not rectangular, can be used for these sub-spaces (see for instance [10]) but generally are less efficient, since they do not guarantee usually a strongly polynomial time complexity.

→ For the need of the performance guarantee of the algorithm, we set $\delta = \frac{\epsilon P}{2m}$. In Figures 4–8, we can see an illustration of the two steps described above for an arbitrary iteration i.

3.2. The algorithm

These modifications lead us to the modified algorithm showed in Algorithm 2. The next subsection of this paper is dedicated to the proof that this modified dynamic programming algorithm is an FPTAS.

4. THE PROOF

Before proving that the modified algorithm is an FPTAS for the problem, we need to show a lemma that will be useful in the second part of the proof.

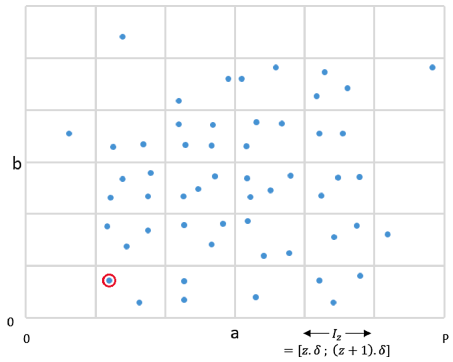

FIGURE 5. We select the state with the smallest a in the 1st "square".

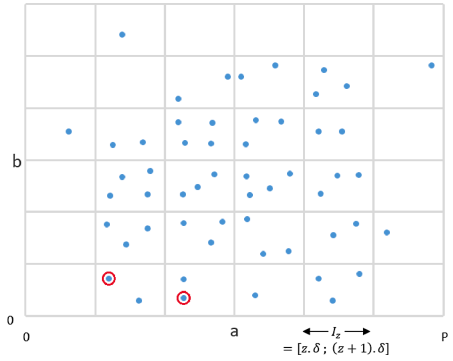

FIGURE 6. When two states have the same a, we take the one with the smallest b.

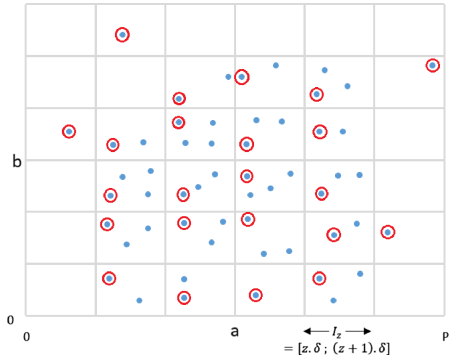

FIGURE 7. We keep only one state for each delimited zone.

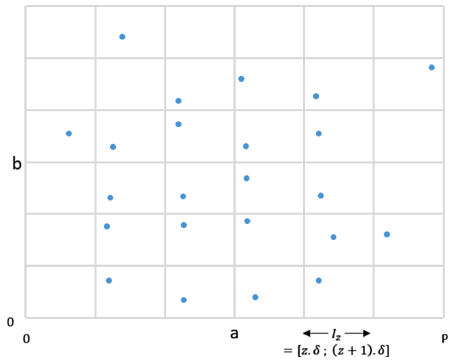

FIGURE 8. $\chi_i^{\#}$ is composed of the kept state of each interval.

Algorithm 2: FPTAS pseudo-code: reducing the number of states kept at each iteration.

Input: The set of tiles $\mathcal{T}$ and the set of symbols $\mathcal{S}$
Output: The optimal value

begin
  $\chi_0^{\#} \leftarrow \{[0,0,0,0]\}$
  Fill matrix M
  for *i from* 1 *to* m **do**
    $\chi_i^{\#} \leftarrow \emptyset$ **foreach** $[a^{\#}, b^{\#}, j, k]$ *in* $\chi_{i-1}^{\#}$ **do**
      $\chi_i^{\#} \leftarrow \chi_i^{\#} \cup \{[a^{\#} + \mathsf{M}(i,j), b^{\#}, i, k]\}$ $\chi_i^{\#} \leftarrow \chi_i^{\#} \cup \{[a^{\#}, b^{\#} + \mathsf{M}(i,k), j, i]\}$
    For every $[I_z, I_{z'}, j, k]$, keep only one state $[a^{\#}, b^{\#}, j, k]$ such as $a^{\#} \in I_z$ and $b^{\#} \in I_{z'}$. $(z, z' \in \{0, 1, 2, \ldots, z_0\})$ If several states can be chosen, keep the one with the smallest a.
    Delete set $\chi_{i-1}^{\#}$
return $\min_{[a^{\#},b^{\#},j,k] \in \chi_m^{\#}} \{\max\{a^{\#}, b^{\#}\}\}$

4.1. Lemma and proof

Lemma 1. *$\forall\ [a,b,j,k] \in \chi_i (i \in \{0,\ldots,m\})$, the modified algorithm generates at least one state $[a^{\#}, b^{\#}, j, k] \in \chi_i^{\#}$ such as:*

$$a^{\#} \leq a \tag{1}$$

$$b^{\#} \leq b + i\delta. \tag{2}$$

Proof by Induction.

Base case. For $i = 0$, we have:

$\chi_0 = \chi_0^{\#} = \{[0,0,0,0]\}$.
$\hookrightarrow$ Then, the lemma is verified for $i = 0$.

Inductive step. We assume that the property is verified until step $i-1$ (with $i \geq 1$). We want to prove that the lemma holds for step i.

Let us consider an arbitrary state $[a,b,j,k] \in \chi_i$ produced in step i. This state can have been created from two different ways: the ith job was scheduled on $P1$ and so we can replace j by i (represented in Eq. (3)) or it was scheduled on $P2$ and we replace k by i (see Eq. (4)).

$$[a,b,j,k] = [a,b,i,k] = [a' + \mathsf{M}(i,j'), b', i, k] \text{ with } [a', b', j', k] \in \chi_{i-1} \tag{3}$$

$$[a, b, j, k] = [a, b, j, i] = [a', b' + \mathsf{M}(i, k'), j, i] \text{ with } [a', b', j, k'] \in \chi_{i-1}. \tag{4}$$

Case 1. $[a, b, j, k] = [a' + \mathsf{M}(i, j'), b', i, k]$ with $[a', b', j', k] \in \chi_{i-1}$.
The induction hypothesis says that there exists a state $[a'^{\#}, b'^{\#}, j', k]$ in $\chi_{i-1}^{\#}$ such that:

$$a'^{\#} \leq a' \tag{5}$$
$$b'^{\#} \leq b' + (i-1)\delta. \tag{6}$$

The modified algorithm will thus create the state $[a'^{\#} + \mathsf{M}(i, j'), b'^{\#}, i, k] \in \chi_i^{\#}$. This state may be kept or it can be represented by another state $[\alpha, \beta, i, k]$ from the same zone in $\chi_i^{\#}$such that:

$$\alpha \leq a'^{\#} + \mathsf{M}(i, j') \tag{7}$$
$$\beta \leq b'^{\#} + \delta \tag{8}$$
$$(7) \implies \alpha \leq a' + \mathsf{M}(i, j') \text{ (using the lemma)} \tag{9}$$
$$(8) \implies \beta \leq b'^{\#} + \delta \leq b' + (i-1)\delta + \delta = b' + i\delta \tag{10}$$
$$\hookrightarrow \quad \beta \leq b + i\delta \tag{11}$$

$\hookrightarrow$ The lemma is verified for the first case.

Case 2. $[a, b, j, k] = [a', b' + \mathsf{M}(i, k'), j, i]$ with $[a', b', j, k'] \in \chi_{i-1}$.
The induction hypothesis says that there exists a state $[a'^{\#}, b'^{\#}, j, k']$ in $\chi_{i-1}^{\#}$ such that:

$$a'^{\#} \leq a' \tag{12}$$
$$b'^{\#} \leq b' + (i-1)\delta. \tag{13}$$

The modified algorithm will thus create the state $[a'^{\#}, b'^{\#} + \mathsf{M}(i, k'), j, i]$. This state may be kept or it can be dominated by another state $[\alpha, \beta, j, i]$ in $\chi_i^{\#}$ with:

$$\alpha \leq a'^{\#} \tag{14}$$
$$\beta \leq b'^{\#} + \mathsf{M}(i, k') + \delta \tag{15}$$
$$(14) \implies \alpha \leq a' \text{ (using the lemma)} \tag{16}$$
$$\alpha \leq a \text{ (as } a' = a) \tag{17}$$
$$(15) \implies \beta \leq b'^{\#} + \mathsf{M}(i, k') + \delta \leq b' + \mathsf{M}(i, k') + (i-1)\delta + \delta \tag{18}$$
$$\beta \leq b + i\delta \tag{19}$$

$\hookrightarrow$ The lemma is verified for the second case.

Conclusion. Since both the base case and the inductive step have been performed, by induction the lemma holds for all natural numbers. □

Theorem 1. *The problem admits an FPTAS.*

Proof. In order to prove the modified algorithm is an FPTAS, we need to show two results:

- First, we must prove that the algorithm respects the performance quality required to belong to this family of algorithms;
- Then, we have to prove that the time complexity is bounded by a polynomial in the input size and $1/\epsilon$.

Performance quality. Consequence of the Lemma 1: Let $[a^*, b^*, l, s] \in \chi_m$ be a state associated with an optimal solution.

The lemma claims there exists a state $[a^\#, b^\#, l, s]$ in $\chi_m^\#$ such as:

$$a^\# \leq a^* \tag{20}$$
$$b^\# \leq b^* + m\delta. \tag{21}$$

First, the equations (20) and (21) imply that:

$$(20) \implies a^\# \leq a^* \tag{22}$$
$$(21) \implies b^\# \leq \text{OPT} + m * \frac{\epsilon P}{2m} \tag{23}$$
$$\leq \text{OPT} + \frac{\epsilon P}{2} \tag{24}$$
$$\leq \text{OPT} + \epsilon * \text{OPT} \tag{25}$$
$$\leq (1 + \epsilon)\ \text{OPT} \tag{26}$$

$\hookrightarrow$ The solution found by the modified algorithm respects the condition quality.

Time complexity. As we keep only one representative per box and as each interval length is equal to δ, there are in fact $\frac{P}{\delta}$ different possible intervals respectively for $a^\#$ and $b^\#$. In every box, at most one state will remain. Moreover, by using the dominance between boxes, the number of non-dominated states $[a^\#, b^\#, j, k]$ for a given couple (j, k) can be reduced to $\mathcal{O}(\frac{P}{\delta})$.

Furthermore, there are m tiles in total, which means that for one iteration, there are at most $m^2.(\frac{P}{\delta})$ generated non-dominated states. As there are m iterations, we generate at most $\mathcal{O}(\frac{m^4}{\epsilon})$ non-dominated states.

$\rightarrow$ The time complexity of the modified algorithm can be reduced to $\mathcal{O}(\frac{m^4}{\epsilon})$.

Conclusion. Both conditions are respected, which implies that the modified algorithm is an FPTAS. Moreover, the complexity time is strongly polynomial.

Conclusion. The problem admits an FTPAS with a strongly time complexity. □

Both algorithms we designed to address $2 \mid \text{merging, tree} \mid C_{\max}$ being presented, let us move on the experimental study we conducted.

5. Experiments

Before going further, let us first recall a central element to our algorithms: the triangular matrix M (also called cost matrix). Recall that a cell in M represents the number of symbols we have to add to a page p in order to be able to assign tile i to it, knowing that the last tile assigned to p was tile j. Since this matrix is fundamental for our algorithms, we decided to try to measure the impact of the average of values in the matrix might have on the performance of the algorithms. However, we have to mention that there is an input of the problem directly impacted by these values: the larger the mean values in the matrix, the larger the weights of the symbols and therefore the larger the $\sum_{i=1}^{n} p_i$ will be. The only control we have chosen to keep over the p_i is the range of values in which they are allowed to evolve. We have chosen three: i1: $p_i \in [1; 50]$, i2: $p_i \in [51; 100]$ and finally i3: $p_i \in [101; 200]$. The number of tiles in an instance and the ϵ used in the FPTAS are logical parameters to study. We have chosen to use five values for ϵ: 0.1, 0.3, 0.5, 0.7 and 0.9. The instances created have a number of tiles between 6 and 48. The last parameter we varied is the height of the tree representing the hierarchy in the tiles. We have results for three heights ($h = 6$, $h = 7$ and $h = 8$). We tried for greater heights but the computation times became too long to test all the instances we wanted. In summary, for a height, we created three subgroups according to the range of values allowed for p_i (i1, i2 or i3). In each subgroup, we generated three groups of one hundred instances with a certain mean in the M matrix.

cost mean = 22

tile count	instance count	Number of DP run success	% of instance that the DP
13	15	15	100
14	5	5	100
15	12	12	100
16	8	8	100
17	13	13	100
18	3	3	100
19	11	11	100
20	7	7	100
21	3	3	100
22	4	4	100
24	4	4	100
25	4	4	100
26	4	4	100
28	3	0	0
31	1	0	0
34	2	0	0
38	1	0	0

cost mean = 71

tile count	instance count	Number of DP run success	% of instance that the DP
15	12	12	100
16	8	8	100
17	13	13	100
18	3	3	100
19	11	11	100
20	7	7	100
21	3	3	100
22	4	4	100
24	4	4	100
25	4	4	100
26	4	4	100
28	3	0	0
31	1	0	0
34	2	0	0
38	1	0	0

cost mean = 109

tile count	instance count	Number of DP run success	% of instance that the DP
15	12	12	100
16	8	8	100
17	13	13	100
18	3	3	100
19	11	11	100
20	7	7	100
21	3	3	100
22	4	4	100
24	4	4	100
25	4	4	100
26	4	4	100
28	3	0	0
31	1	0	0
34	2	0	0
38	1	0	0

FIGURE 9. Simple statistics over the number of successful execution of the DP algorithm.

TABLE 1. Presentation of simple statistics on the percentage of successful DP executions as a function of the number of tiles in the instances.

Tile count	Instance count	Number of successful DP runs	% of successful DP runs
26	16	16	100
27	8	16	100
28	18	0	0
29	18	0	0
31	5	0	0
32	10	0	0

5.1. Impact of the number of tiles on the hardness of an instance

As the theoretical time complexities of both algorithms depend on the number of tiles m in an input, we suppose there exists a close link between m and the difficulty of an instance. Figure 9 displays three tables presenting simple statics on executions of the dynamic programming algorithm. Each table works in the same way: it regroups all inputs having the same cost mean value c. Then, the instances are separated according to the number of tiles m its contain. Finally, we ran the DP on each group defined by (c, m) and we added up the number of successful runs. For example, we see there is a 100% rate of success for inputs having 13 tiles and an average cost value of 22. Conversely, the success rate drops to 0% for inputs having more than 28 tiles whatever the average cost value.

We deduce that if an instance has 26 tiles or less, then we will be able to find a solution for it using our dynamic programming algorithm. However, if the input has 28 tiles or more, the combinatorial explosion becomes too important (the memory of the computer used to perform the tests was saturated).

There remains the case of instances having 27 tiles for which the data presented in Figure 1 do not allow us to draw a reliable conclusion. However, in another group (with larger processing time: $p_i \in [51, 100]$) for which a small number of dynamic programming runs were successful, the results seem to indicate that 27-tile instances are still solvable by DP algorithm (see Tab. 1).

We do not present the totality of the results for $h = 7$, $p_i \in [51, 100]]$ and a cost mean equal to 406 because we have only tried dynamic programming on a part of the instances (the first 70). As the other instances were not processed, we should not try to interpret the results as a whole.

In order to validate our results, we tried to show a positive correlation between the execution time and the number of tiles in an instance. For this, we used the Principal Component Analysis (PCA) method.

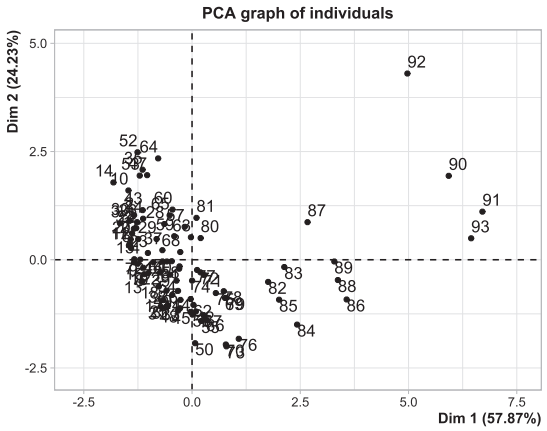

FIGURE 10. Example of a cloud of individuals (cloud obtained by PCA on the DP results).

We used an implementation of PCA provided by the R library *FactoMineR* and ran this algorithm on several result files. We studied [7, 12, 13] to help us draw our conclusions.

The data files selected for the PCA are the results of running the DP algorithm and the FPTAS (for three values of ϵ: 0.1, 0.5 and 0.9) on a group of 100 instances with characteristics $h = 7$, $p_i \in [1; 50]$, cost mean = 109. The number of tiles in this set of instances is between 13 and 38.

The first step when applying PCA is to check that there is not a factorial axis impacted by only a small number of individuals (and therefore that the cloud of individuals along the different factorial axes is roughly regular). In fact, if an axis was predominantly impacted by a small number of individuals, we would have to start by interpreting the results in terms of individuals and not in terms of variables first. This is not an issue here: most of the inputs are close to the axe's origin but not along a single axis (an example of a cloud of individuals is given in Fig. 10).

The numbers that appear in the figures generated by the PCA represent an instance: we could not use the names of the instances directly as they were too long. The correspondence is available, but it is sufficient to know that the larger a number is, the larger the number of tiles in the instance. We did not put all the clouds of individuals we had to generate and then check, that would have made too many different graphs.

Let us now study the graph of variables obtained when we used the PCA over the data of the runs of DP algotihm. An example is available in Figure 11. Some graphs of variables of the FPTAS are available in the appendix (see Figs. A.1–A.3).

It can be seen that systematically, the arrows representing the time and the number of tiles are separated by acute angles. Moreover, The length of both arrows are very similar.

We can conclude that the time and the number of tiles are positively and closely correlated

5.2. Impact of the average value in the cost matrix

To see if the average in the cost matrix has a direct impact on the execution times of the algorithms, we grouped in a single file all the results for the instances containing twenty tiles. Then, for each value of the average in the matrix M, we computed the average execution time. The results for $h = 7$ are presented in Table 2.

In Figure 12 are presented several histograms summarising in a more visual way the data in the Table 2.

It is apparent that the overall look of the histograms presented in Figure 12 match what we expected. We will focus on the impact on the average value in the cost matrix M for now and we will move on to the impact of ϵ in the next paragraphs.

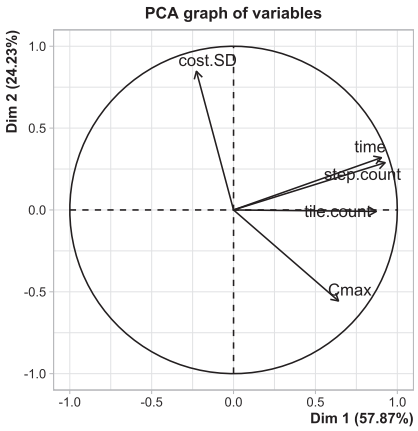

FIGURE 11. Graph of variables when PCA was fed data from DP executions.

TABLE 2. Table of execution times of the FPTAS as a function of the average value in the cost matrix.

	Cost mean values:		22	71	109	250	323	529	606	694
		$\epsilon = 0.1$	0,90	0,94	0,95	0,67	0,78	0,76	0,69	0,72
$h = 7$	Average time	$\epsilon = 0.3$	0,23	0,24	0,23	0,22	0,22	0,24	0,23	0,22
	(in seconds)	$\epsilon = 0.5$	0,12	0,12	0,11	0,11	0,22	0,11	0,13	0,11
		$\epsilon = 0.7$	0,08	0,08	0,07	0,08	0,12	0,07	0,08	0,07
		$\epsilon = 0.9$	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,05

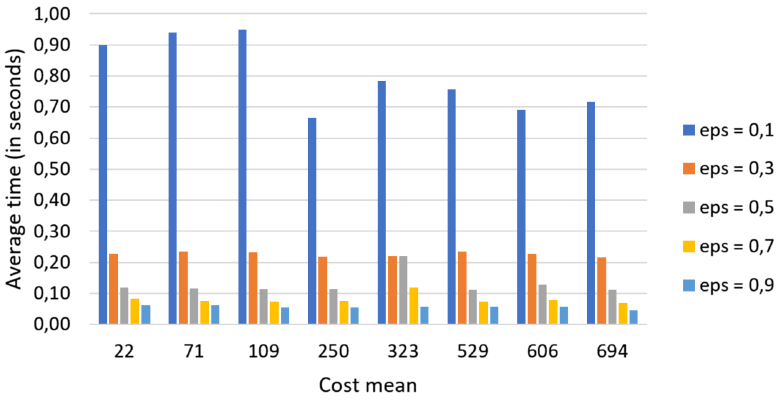

FIGURE 12. Presentation of average execution times of the FPTAS on instances having 20 tiles.

It seems that for different cost mean values but fixed ϵ values, the executions times are quite steady. Therefore, it seems that the value of the average cost in matrix M does not have a direct impact on the execution times of the FPTAS.

This hypothesis is confirmed by the PCA results presented in Figure 13. In order to obtain them, we ran the PCA on all the execution times of the FPTAS whatever the ϵ value.

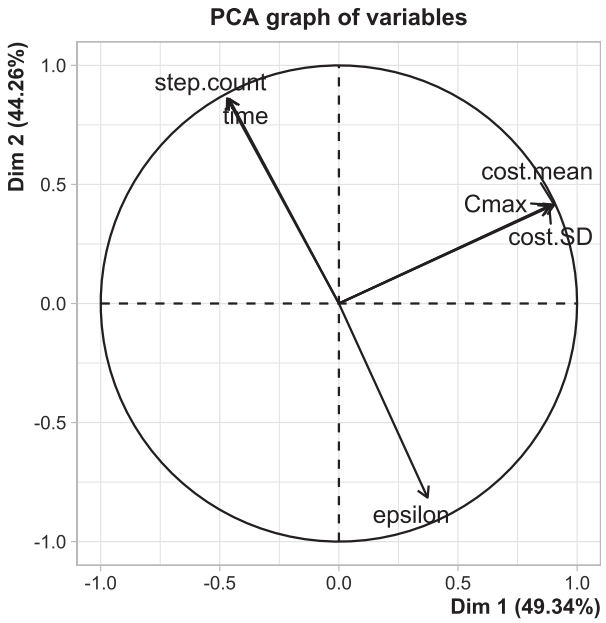

FIGURE 13. Graph of variable when we ran the PCA over execution times of the FPTAS.

We notice that the PCA shows that the arrows representing the time and the average in the cost matrix are almost orthogonal. It means that both variables seem to be very slightly correlated. But as their length are similar and nearly equal the radius of the circle, they both convey a lot of information in regard of the dimensions we chose for the graph of variables.

5.3. Impact of ϵ

Regarding the value of ϵ, it seems that she smaller the ϵ, the longer the execution times. Indeed, the arrows representing these values are almost on the same straight line, of nearly the same length but of opposite directions. This means that the execution time and the value of ϵ are negatively correlated: the bigger the epsilon, the smaller the execution times.

We can therefore state that the execution time of the FPTAS is inversely proportional to the value of ϵ.

6. Conclusion and perspectives

In this paper, we presented the work we conducted on a first scenario of the pagination problem. We establish that this scenario is NP-hard in the ordinary sense since a pseudo-polynomial dynamic programming algorithm is proposed to solve optimally the problem. In addition, we propose an FPTAS with a strongly polynomial time complexity. This scheme is based on the simplification of the dynamic programming algorithm by removing a part of the generated states at every iteration of the algorithm, without deteriorating the solution quality too much. The exact algorithm is compared to the FPTAS in terms of effectiveness and fastness. The numerical results are conducted and analysed, which allows us to understand more on the correlation of some parameters by applying PCA approach.

As perspectives, the study of other scenarios seems to be very interesting (the case of 2 symbols per tile is in especially very challenging). Moreover, we will try to find another reliable alternative to devise different FPTAS as it is done in other references [2,3,9]. Finally, the study of differential approximation seems of a great interest since the general pagination problem is difficult to approximate [1].

Appendix A. PCA graphs of variables

Here follow the graphs of variables obtained when PCA was fed data from FPTAS executions with different values of ϵ.

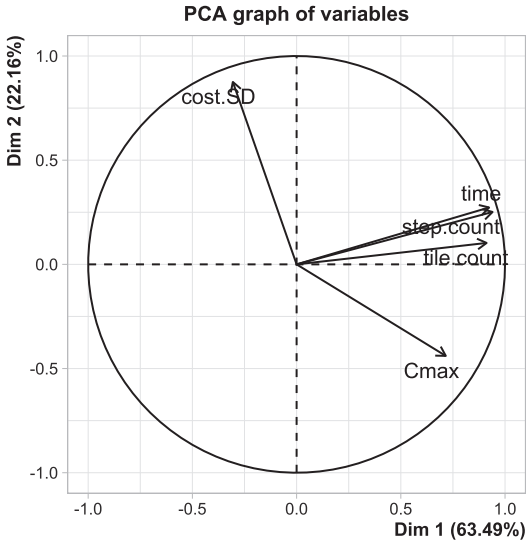

Figure A.1. Graph of variables when PCA was fed data from FPTAS executions where $\epsilon = 0.1$.

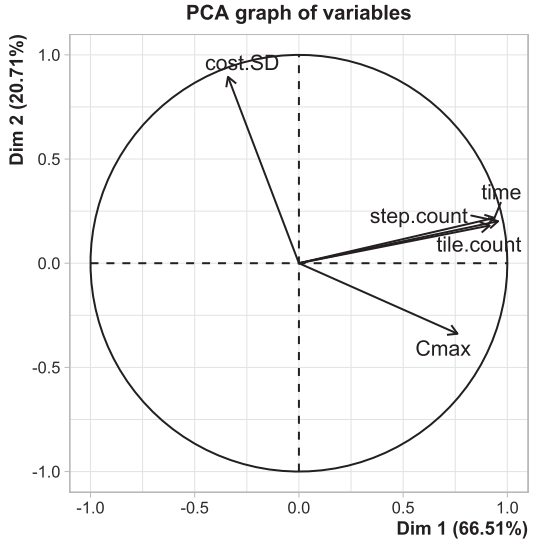

Figure A.2. Graph of variables when PCA was fed data from FPTAS executions where $\epsilon = 0.5$.

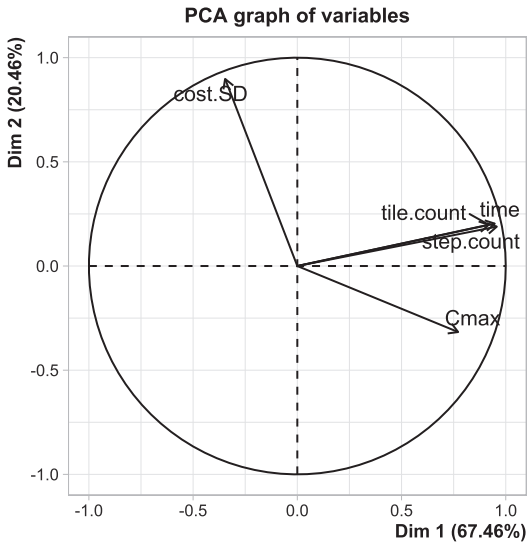

FIGURE A.3. Graph of variables when PCA was fed data from FPTAS executions where $\epsilon = 0.9$.

Acknowledgements. This work was supported by the French MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE, by the CONSEIL REGIONAL GRAD-EST, by the EUROPEAN UNION, through the Cyber-entreprises project. The preliminary version of this work has been published at CIE49 conference [6].

REFERENCES

[1] G. Ausiello and V. Paschos, Differential approximation (September 2005), Chapter 13 in Approximation Algorithms and Metaheuristics, edited by T. Gongalez. Taylor and Francis (2005).

[2] A. Elalouf, An FPTAS for just-in-time scheduling of a flow shop manufacturing process with different service quality levels. *RAIRO Oper. Res.* **55** (2021) S727–S740.

[3] A. Elalouf and E. Levner, Improving the solution complexity of the scheduling problem with deadlines: a general technique. *RAIRO Oper. Res.* **50** (2016) 681–687.

[4] A. Grange, I. Kacem and S. Martin, Algorithms for the bin packing problem with overlapping items. *Comput. Ind. Eng.* **115** (2018) 331–341.

[5] A. Grange, I. Kacem, K. Laurent and S. Martin, On the knapsack problem under merging objects' constraints, in Proc. of 45th International Conference on Computers and Industrial Engineering 2015 (CIE45). Vol. 2. Metz, France, Curran Associates, Inc. (2018) 1359–1368.

[6] A. Grange, I. Kacem, S. Martin and S. Minich, Fully polynomial-time approximation scheme for the pagination problem, in Proc. of 49th International Conference on Computers and Industrial Engineering 2019 (CIE49). Beijing, China (2019) 400.

[7] I.T. Jolliffe, Principal component analysis for special types of data, in Principal Component Analysis. *Springer Series in Statistics*. Springer, New York, NY (1986).

[8] I. Kacem, Approximation algorithms for the makespan minimization with positive tails on a single machine with a fixed non-availability interval. *J. Comb. Optim.* **17** (2009) 117–133.

[9] I. Kacem and H. Kellerer, Fast approximation algorithms to minimize a special weighted flow-time criterion on a single machine with a non-availability interval and release dates. *J. Sched.* **14** (2011) 257–265.

[10] I. Kacem and E. Levner, An improved approximation scheme for scheduling a maintenance and proportional deteriorating jobs. *J. Ind. Manage. Optim.* **12** (2016) 811.

[11] M. Kovalyov and E. Pesch, A generic approach to proving NP-hardness of partition type problems. *Discrete Appl. Math.* **158** (2010) 1908–1912.

[12] M. Rouaud, Probabilités, statistiques et analyses multicritères (2012).

[13] J. Pagès, Analyse Factorielle Multiple Avec R. Vol. 158. EDP Sciences (2013) 1908–1912.

[14] M. Sindelar, R.K. Sitaraman and P. Shenoy, Sharing-aware algorithms for virtual machine colocation, in Proc. of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA'11. San Jose, California, USA (2011) 367–378.