FPT Approximation Algorithm for Scheduling with Memory Constraints

Eric Angel[1], Cédric Chevalier[2], Franck Ledoux[2], Sébastien Morais[1,2(✉)], and Damien Regnault[1]

[1] IBISC, Université d'Evry, Evry, France
{Eric.Angel,Sebastien.Morais,Damien.Regnault}@ibisc.univ-evry.fr
[2] CEA, DAM, DIF, 91297 Arpajon, France
{Cedric.Chevalier,Franck.Ledoux,Sebastien.Morais}@cea.fr

Abstract. In this paper we study a scheduling problem motivated by performing intensive numerical simulations on large meshes. In order to run the simulation as fast as possible, we must allocate computations on different processors such that the makespan is minimized, but also take care of the limited memory on each processor. We present a dynamic programming based algorithm that ensures that both of these objectives are satisfied, within a ratio of $1 + \varepsilon$. Our algorithm is fixed-parameter tractable (FPT) with respect to the path-width of the graph. For sake of readability, the algorithm is presented for two identical machines, but it can be generalized for a fixed number of unrelated processors.

Keywords: Scheduling · Approximation algorithm · Dynamic programming · Fixed-parameter tractable

1 Introduction

In this paper, we study a specific scheduling problem involving two types of memory constraints: each processing unit has a bounded memory capacity; the tasks to be scheduled depend on each others in a complex way, which we model using a graph structure. A motivation for this problem comes from distributed numerical simulations where most numerical schemes are based on finite elements or volume methods (FEM or VEM) [3,10]. Such approaches require the geometric domain of study Ω to be discretized into basic elements, called cells, which form a mesh. Then, each cell j is assigned a computation valued by a computation cost p_j, and data (like density, pressure, ...) valued by a memory weight m_j. Moreover, performing the computation of a cell j requires, in addition to its data, data located in its neighborhood[1], denoted $\mathcal{N}(j)$. For a distributed simulation, the problem is so to assign all the computations to processing units

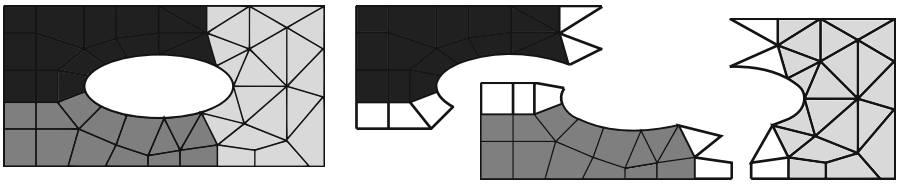
[1] The neighborhood is most of the time topologically defined (cells sharing an edge or a face) and its depth depends on the numerical scheme used for performing the numerical simulation.

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 196–208, 2016.
DOI: 10.1007/978-3-319-43659-3_15

with bounded memory capacities, while minimizing the makespan[2] and ensuring that the following constraints are satisfied:

- the computation of each cell j is scheduled to a processing unit;
- if a processing unit performs the computation of a cell j, then it needs to locally access data from cell j and cells in $\mathcal{N}(j)$;
- the amount of data stored by a processing unit cannot exceed its capacity.

To illustrate such an assignment, let us consider Fig. 1(a), where a mesh M_Ω and its associated computations are assigned onto 3 processing units. This assignment puts each computation onto a processing unit according to the cell color. Due to the neighborhood constraint, the total amount of memory needed for each processing unit is not limited to those colored cells but extends to some adjacent cells. For an edge-based adjacency relationship, we get the configuration presented in Fig. 1(b), where the memory needed for each processing unit is equal to the memory of both white and colored cells. The white cells contain additional data needed by a processing unit to process all its assigned computation.



(a) Compact view of the computation assignment of M_Ω

(b) Exploded-view of M_Ω, where the white cells are cells whose data are locally known by the processing unit and the computation is not performed by the latter.

Fig. 1. Computation assignment of a mesh M_Ω onto 3 processing units in (a) and its exploded-view with neighbor memory needed (b).

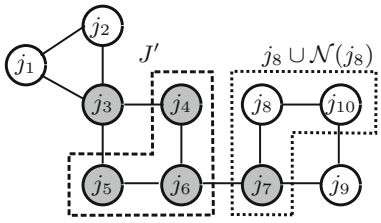
In the present work, we model this problem as a scheduling problem using a graph $G(J, E)$, which we refer to as the *neighborhood graph*[3]. Computations assigned to cells correspond to *jobs* (one per cell in the mesh), modeled by the set J. Throughout this paper we will denote by n the number of jobs, i.e. $n := |J|$. Job $j \in J$ requires $p_j \in \mathbb{N}$ *units of time* to be executed (computation time) and an amount $m_j \in \mathbb{N}$ of memory. Jobs have to be assigned among k identical *machines* (i.e. processing unit), each machine l having a memory capacity M_l, for $l = 1, \ldots, k$. Moreover, each job j requires data from some *adjacent jobs*, denoted by $\mathcal{N}(j) \subseteq J$. We say that jobs $j \in J$ and $j' \in J$ are adjacent if there

[2] Recall that the makespan is the maximum computation time among the processing units.

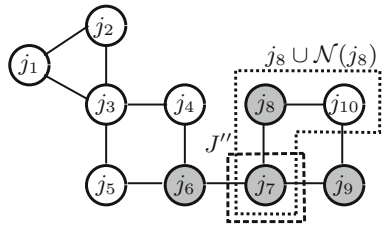
[3] We draw the reader's attention on the fact that this graph is not a precedence graph, as our problem has no precedence relation between the jobs.

is an edge $(j, j') \in E$. We assume the graph is not directed, i.e. $j' \in \mathcal{N}(j)$ if and only if $j \in \mathcal{N}(j')$. For a subset of jobs $J' \subseteq J$, $\mathcal{N}(J') := \cup_{j \in J'} \mathcal{N}(j)$. When a subset of jobs $J' \subseteq J$ is scheduled on a machine, this machine needs to allocate an amount of memory equal to $\sum_{j \in J' \cup \mathcal{N}(J')} m_j$, while its processing time is $\sum_{j \in J'} p_j$. The objective is then to assign all jobs of J onto a machine, while minimizing the makespan and ensuring strong memory constraints: the amount of memory stored by each machine is smaller than or equal to its memory capacity. In the following we assume that there exists at least one feasible solution, i.e. an assignment of all the jobs such that the memory constraint on each machine is satisfied. Using the notation introduced by Graham et al. [5] we refer to our problem as $Pk|G, mem|C_{max}$. Notice that the second field doesn't contain the term *prec* as there are no precedence constraints among the jobs.

The neighborhood graph $G(J, E)$ is the main feature of $Pk|G, mem|C_{max}$ and dealing with it is the most challenging part as dynamically assigning the jobs may lead to different amount of memory needed to be allocated. As an illustration, let us consider an instance with 2 machines with the neighborhood graph depicted on Fig. 2. Suppose that the subset $J' := \{j_4, j_5, j_6\}$ is assigned to machine 1 while the subset $J'' := \{j_7\}$ is assigned to machine 2. Then the assignment of j_8 to machine 1 or 2 has a different impact in terms of memory allocation: assigning j_8 to machine 1 makes this machine to allocate an additional amount of memory equal to $m_{j_8} + m_{j_{10}}$ (see Fig. 2(a)) whereas assigning it to machine 2 makes this machine to allocate an additional amount of memory equal to $m_{j_{10}}$ only (see Fig. 2(b)).



(a) The assignment of $J' = \{j_4, j_5, j_6\}$ constraints the machine to allocate an amount of memory for each (colored) job $j \in J' \cup \mathcal{N}(J') = \{j_3, j_4, j_5, j_6, j_7\}$. Assigning j_8 to this machine induces an additional amount of $m_8 + m_{10}$.



(b) The assignment of $J'' = \{j_7\}$ constraints the machine to allocate an amount of memory for each (colored) job $j \in J'' \cup \mathcal{N}(J'') = \{j_6, j_7, j_8, j_9\}$. Assigning j_8 to this machine induces an additional amount of m_{10}.

Fig. 2. A neighborhood graph $G(J, E)$ and the memory allocation induced by $J' = \{j_4, j_5, j_6\} \in J$ in (a) and $J'' = \{j_7\} \in J$ in (b).

1.1 Related Problems

When $m_j = 0$ for each job j, the problem $Rk|G, mem|C_{max}$ becomes the well-known **NP**- scheduling problem denoted by $Rk||C_{max}$. Lenstra et al. [9] gave a

2-approximation algorithm when the number of machines k is not part of the input and proved that no polynomial algorithm can achieve an approximation ratio less than 3/2 unless $P = NP$. Their algorithm computes an optimal fractional solution to a natural LP-relaxation and then uses rounding to obtain a schedule for the discrete problem. In [4], Gairing et al. gave a faster algorithm that matches the 2-approximation quality and which is based on unsplittable flow techniques. If the number of machines is fixed, there exist a fully polynomial-time approximation scheme [14].

When the neighborhood graph has no edges, and the memory is bounded on each machine, and $m_j = 1$ for each job j, we get the so-called Scheduling Machines with Capacity Constraints problem (SMCC). In this problem, each machine k can process at most a fixed number of jobs. Saha and Srinivasan [12] gave a 2-approximation in a more general scheduling setting, i.e. Scheduling Unrelated Machines with Capacity Constraints. For the special case of two machines, Woeginger designed a FPTAS for this problem [15].

1.2 Main Contribution

As $Pk|G, mem|C_{max}$ is a generalization of those well-known scheduling problems, a reasonable question is to know whether we can get approximation algorithms, which could possibly depend on some parameters of the neighborhood graph[4]. We answer this question by providing a dynamic programming based algorithm that, assuming that there exists at least one feasible solution to our problem, returns a solution within a ratio of $(1 + \varepsilon)$ for both the optimum makespan and the memory capacity constraints. This algorithm is Fixed-Parameter Tractable (FPT) with respect to the path-width of the neighborhood graph. Notice that there cannot exist an exact FPT algorithm with respect to the path-width parameter, since when the graph has no edge (and therefore a path-width equal to 0), the problem is NP-hard (see Sect. 1.1).

1.3 Outline of the Paper

We start by briefly recalling in Sect. 2 the definitions of different notions useful for our proof. We then provide in Sect. 3 a dynamic programming based algorithm that computes all the solutions to this problem. This task is not trivial as dynamically assigning the jobs may lead to differents amounts of memory needed to be allocated. Since the time complexity of this algorithm is not polynomial in the input size, we apply the Trimming-of-the-State-Space technique [6] in Sect. 4 obtaining an approximation algorithm that is FPT with respect to the path-width of the graph. Finally, we give some concluding remarks in Sect. 5.

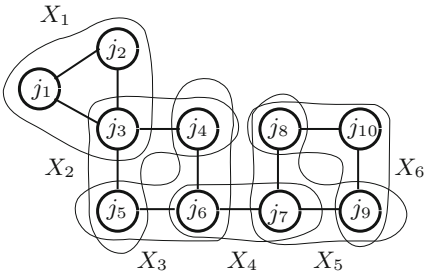
[4] Recall that an algorithm is *fixed-parameter tractable* (FPT) with respect to h if its running time is bounded by $f(h).|I|^{O(1)}$ where $|I|$ is the size of the instance and f is an arbitrary function depending only on the parameter h.

2 Definitions

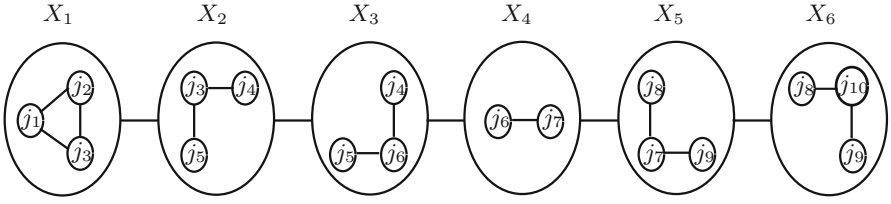
Throughout this paper we consider simple, finite, undirected graphs. Let us start by defining the notions of path decomposition and path-width. They were initially introduced in the framework of graph minor theory [11]. A *path decomposition* of a graph $G(J, E)$ is a pair (P, X), where $P := (J(P), E(P))$ is a path, and $X := (X_i)_{i \in J(P)}$ is a family of subsets of J satisfying:

1. $\cup_{i \in J(P)} X_i = J$;
2. $\forall (j, j') \in E$, there exists an $i \in J(P)$ such that $\{j, j'\} \subseteq X_i$;
3. $\forall j, j', j'' \in J(P)$, if j' lies on the path from j to j'' then $X_j \cap X_{j''} \subseteq X_{j'}$.

The *width* of a path decomposition is $\max(|X_i| - 1 : i \in J(P))$ and the *path-width* of G is the minimum width of a path decomposition of G. The construction of such a path decomposition is illustrated on Fig. 3(a), and the result is presented on Fig. 3(b). When P is required to be a tree instead of being a path, previous definitions straightforwardly extend to the definitions *tree decomposition* and *tree-width* of a graph.



(a) Construction of $X := (X_i)_{i \in J(P)}$, a family of subsets of J satisfying the three properties of a path decomposition.



(b) Path decomposition (P, X) of the neighborhood graph $G(J, E)$.

Fig. 3. Example of a path decomposition (P, X) of the neighborhood graph $G(J, E)$, where X is composed by the subsets $X_1 = \{j_1, j_2, j_3\}$, $X_2 = \{j_3, j_4, j_5\}$, $X_3 = \{j_4, j_5, j_6\}$, $X_4 = \{j_6, j_7\}$, $X_5 = \{j_7, j_8, j_9\}$, $X_6 = \{j_8, j_9, j_{10}\}$ in (a) and (P, X) is presented in (b). This path decomposition is optimal with respect to the path-width.

In order to define the notion of vertex separation number, which is equivalent to path-width [7], let us introduce the notion of *(linear) layout* of a graph $G(J, E)$, which is simply a one-to-one mapping $L : J \to \{1, 2, \ldots, |J|\}$. For any layout L, we define

$$V_L(i) := \{j \in J \mid L(j) \leq i \text{ and } \exists j' \in J \text{ such that } (j, j') \in E \text{ and } L(j') > i\}.$$

Thus $V_L(i)$ is the set of vertices of G mapped to integers less than or equal to i and that are adjacent to vertices mapped to integers greater than i. Then the *vertex separation* of G with respect to L, $vs_L(G)$, is the maximum number of vertices in any $V_L(i)$. And eventually, the *vertex separation number* of G is the minimum, over all possible layouts L of G, of $vs_L(G)$. Formally,

$$vs_L(G) := \max_{1 \leq i \leq |J|} \{|V_L(i)|\}$$

and

$$vs(G) := min\{vs_L(G) \mid L \text{ is a linear layout of } G\}.$$

We note $L^* := \arg\min_L vs_L(G)$, i.e. L^* is a linear layout associated with the vertex separation number. Such an optimal numbering is used on Figs. 2 and 3. When G has a path-width bounded by a constant integer value h, such a layout can be obtained in polynomial time by constructing a linear decomposition (P, X) of G with width h [1], and using an algorithm presented in [7], which (by using a suitable data structure) transforms a given optimal path-decomposition into an optimal layout L^*. It has been proved that $pw(G) = O(\log(n)tw(G))$ for any graph G on n vertices [8], and therefore $vs(G) = O(\log(n)tw(G))$, where pw and tw mean path-width and tree-width respectively.

3 An Exact Algorithm Using Dynamic Programming

For sake of readability, the presentation of the algorithm is done for two machines. But it can be generalized to a constant number k of machines, with $k > 2$, as we will see in Sect. 5. In the following, we assume that the jobs have been numbered such that $L^*(j_i) = i$, for $1 \leq i \leq n$, i.e. the layout is optimal with respect to the vertex separation number.

The dynamic programming goes through n phases. Each phase i, with $i = 1, \ldots, n$, processes the job j_i and produces a set $\mathcal{S}_i$ of states. Each state in the state space $\mathcal{S}_i$ is a vector $S = [s_1, s_2, s_3, s_4, C_i] \in \mathcal{S}_i$, which encodes a partial solution for the first i jobs, i.e. an assignment of the first i jobs to the machines, and where:

1. s_1 (resp. s_2) is the total processing time on the first (resp. second) machine in the partial schedule,
2. s_3 (resp. s_4) is the total amount of memory required by the first (resp. second) machine in the partial schedule,

3. C_i is an additional structure, called *combinatorial frontier*. For a given partial solution from j_1 to j_i, it is defined as $C_i := (V_L(i), \sigma_i, \sigma'_i)$ with $\sigma_i : V_L(i) \to \{1, 2\}$ and $\sigma'_i : V_L(i) \to \{0, 1\}$, such that $\sigma_i(j)$ is the machine on which j has been assigned, and $\sigma'_i(j) := 1$ if the machine on which j has not been assigned, i.e. the machine $3 - \sigma_i(j)$, has already memorized the data of j.

Notice that in the definition of C_i we use only the set of vertices $V_L(i)$, instead of the whole set $\{1, \ldots, i\}$, since these vertices are the only ones needed to compute additional amounts of memory induced by the future tasks's assignment. Moreover, the number of distinct combinatorial frontiers C_i is equal to $4^{|V_L(i)|} \leq 4^{vs(G)}$.

The algorithm main structure is summarized in Algorithm 1.

Algorithm 1. Summary of the exact dynamic programming algorithm.

input : A graph $G(J, E)$ where $L^*(j_i) = i$, for $1 \leq i \leq n$
output: A solution vector s

1 $\mathcal{S}_1 := \{[p_1, 0, m_1 + \sum_{j \in \mathcal{N}(j_1)} m_j, 0, C_1^1], [0, p_1, 0, m_1 + \sum_{j \in \mathcal{N}(j_1)} m_j, C_1^2]\}$;
2 **foreach** $i \leftarrow 2, n$ **do**
3 **foreach** $[s_1, s_2, s_3, s_4, C_{i-1}] \in \mathcal{S}_{i-1}$ **do**
4 Compute α_i^1 and C_i^1;
5 $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup [s_1 + p_i, s_2, s_3 + \alpha_i^1, s_4, C_i^1]$;
6 Compute α_i^2 and C_i^2;
7 $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup [s_1, s_2 + p_i, s_3, s_4 + \alpha_i^2, C_i^2]$;
8 **end**
9 **end**
10 $s \leftarrow [s_1, s_2, s_3, s_4, C_n] \in \mathcal{S}_n$ with $s_3 \leq M_1$ and $s_4 \leq M_2$ and such that $\max\{s_1, s_2\}$ is minimum;

Line 1 is the initialization phase. The set $\mathcal{S}_1$ contains two states, the first (resp. second) is when job 1 is assigned on machine 1 (resp. 2). $C_1^1 := (V_L(1), \sigma_1, \sigma'_1)$ is the combinatorial frontier when job j_1 is assigned on machine 1. Either $V_L(1) = \{j_1\}$ or $V_L(1) = \emptyset$. In case $V_L(1) = \{j_1\}$ we have $\sigma_1(j_1) := 1$ and $\sigma'_1(j_1) := 0$. Similarly, C_1^2 is the combinatorial frontier when job j_1 is assigned on machine 2. Then at each iteration in the lines 4–7, for each state in $\mathcal{S}_{i-1}$ we add two states in $\mathcal{S}_i$: The state in line 5 (resp. 7) corresponds to the case when job j_i is assigned on machine 1 (resp. 2) and α_i^1 (resp. α_i^2) is the memory induced by this assignment. In order to show how to compute α_i^1, used in Line 5, we define two sets of jobs, namely J_1 and J_2, such that

$$J_1 := \{j \in V_L(i-1) \cap \mathcal{N}(j_i) \ : \ \sigma_{i-1}(j) \neq 1 \wedge \sigma'_{i-1}(j) = 0\},$$
$$J_2 := \{j_k \in \mathcal{N}(j_i) : k > i \ \wedge \forall j_l \in V_L(i-1) \cap \mathcal{N}(j_k) \ \sigma_{i-1}(j_l) \neq 1\}.$$

J_1 is the set of jobs that are in the neighborhood of j_i, have not been assigned to machine 1, and have not been memorized by machine 1 either. J_2 is the set of

jobs that are in the neighborhood of j_i, not already assigned, and such that they do not have a job in their neighborhood already assigned to machine 1. Then, we have

$$\alpha_i^1 := \sum_{j \in J_1 \cup J_2} m_j + \begin{cases} m_i \text{ if } \forall j_l \in V_L(i-1) \cap \mathcal{N}(j_i), \sigma_{i-1}(j_l) \neq 1 \\ 0 \text{ otherwise} \end{cases}$$

To illustrate the sets J_1 and J_2, let us consider Fig. 4 where we want to assign j_6 to machine 1, and where $J' = \{j_1, j_2\}$ is assigned to machine 1 and $J'' = \{j_3, j_4, j_5\}$ is assigned to machine 2. We have $V_L(5) = \{j_4, j_5\}$, $\mathcal{N}(j_6) = \{j_4, j_5, j_7\}$ so $V_L(5) \cap \mathcal{N}(j_6) = \{j_4, j_5\}$. As $J'' = \{j_3, j_4, j_5\}$ is assigned to machine 2, we have $\sigma_5(j_4) \neq 1$, $\sigma_5(j_5) \neq 1$ and $\sigma'_5(j_4) = \sigma'_5(j_5) = 0$. Therefore $J_1 = \{j_4, j_5\}$, i.e. assigning j_6 to machine 1 forces this machine to allocate an amount of memory for j_4 and j_5. Moreover, we have $j_7 \in \mathcal{N}(j_6)$ such that $\forall j_l \in V_L(5) \cap \mathcal{N}(j_7)$, $\sigma_5(j_l) \neq 1$. Therefore $J_2 = \{j_7\}$, i.e. assigning j_6 to machine 1 forces this machine to allocate an amount of memory for j_7.

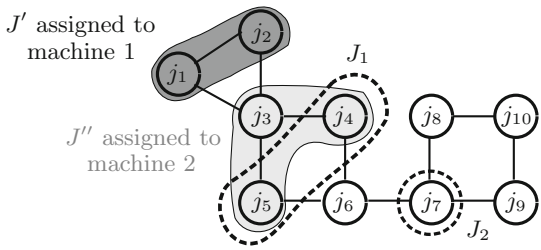


Fig. 4. An example illustrating the sets of jobs J_1 and J_2, when $i = 6$, $J' = \{j_1, j_2\}$ is assigned to machine 1 and $J'' = \{j_3, j_4, j_5\}$ is assigned to machine 2.

Value α_i^2, used in Line 7, is similarly computed. Eventually, let us show how to obtain the new combinatorial frontier $C_i := (V_L(i), \sigma_i, \sigma'_i)$ in Line 5 and 7, denoted by C_i^1 and C_i^2 respectively, from $C_{i-1} := (V_L(i-1), \sigma_{i-1}, \sigma'_{i-1})$. Let us consider the first case, i.e. the vertex j_i is assigned on the first machine, and let us show how to obtain C_i^1. If $j_i \in V_L(i)$, then $\sigma_i(j_i) := 1$, and $\sigma'_i(j_i) := 1$ if $\exists j \in V_L(i-1) \cap \mathcal{N}(j_i)$ such that $\sigma_{i-1}(j) = 2$, and $\sigma'_i(j_i) := 0$ otherwise. For $j \in (V_L(i) \backslash \{j_i\}) \cap \mathcal{N}(j_i)$ we have $\sigma_i(j) := \sigma_{i-1}(j)$ and $\sigma'_i(j) := 1$ if $\sigma'_{i-1}(j) = 1$ or $\sigma_i(j) = 2$, and 0 otherwise. For $j \in V_L(i) \backslash (\{j_i\} \cup \mathcal{N}(j_i))$ we have $\sigma_i(j) := \sigma_{i-1}(j)$ and $\sigma'_i(j) := \sigma'_{i-1}(j)$. The combinatorial frontier C_i^2 can be similarly computed. Notice that in the dynamic programming algorithm, if two states S and S' have the same components, including the same combinatorial frontier, then only one of them is kept in the state space. The time complexity to test whether two states S and S' are the same, is thus $O(vs(G))$.

Let $p_{sum} := \sum_{i=1}^{n} p_i$ and $m_{sum} := \sum_{i=1}^{n} m_i$, then for each vector $S = [s_1, s_2, s_3, s_4, C_i] \in \mathcal{S}_i$, s_1 and s_2 are integers between 0 and p_{sum}, s_3 and s_4 are integers between 0 and m_{sum}, and we have $|\mathcal{S}_i| = O(p_{sum}^2 \times m_{sum}^2 \times 4^{vs(G)})$.

The time complexity of this algorithm is proportional to $\sum_{i=1}^{n} |\mathcal{S}_i|$. Thus, the overall complexity is $O(n \times vs(G) \times p_{sum}^2 \times m_{sum}^2 \times 4^{vs(G)})$.

The time complexity of this algorithm being pseudo-polynomial, we are going to transform it into an approximation FPT algorithm with respect to the path-width of the neighborhood graph.

4 Getting an Approximated Algorithm via Trimming Techniques

In this section, we propose an approximated algorithm, derived from Algorithm 1 to get an FPT approximation algorithm. The main idea is to apply the trimming-the-state technique [14] and to withdraw, during the execution of the algorithm, states that are close to each other.

We define $\Delta := 1 + \varepsilon/2n$, with $\varepsilon > 0$ a fixed constant. Let us first consider the first two coordinates of a state $S = [s_1, s_2, s_3, s_4, C_i]$. We have $0 \leq s_1 \leq p_{sum}$ and $0 \leq s_2 \leq p_{sum}$. We divide each of those intervals into intervals of the form $[0]$ and $[\Delta^l, \Delta^{l+1}]$, with l an integer value getting from 0 to $L_1 := \lceil \log_\Delta(p_{sum}) \rceil = \lceil ln(p_{sum})/ln(\Delta) \rceil \leq \lceil (1 + \frac{2n}{\varepsilon}) ln(p_{sum}) \rceil$. In the same way, we divide the next two coordinates into intervals of the form $[0]$ and $[\Delta^l, \Delta^{l+1}]$, with l an integer value getting from 0 to $L_2 := \lceil \log_\Delta(m_{sum}) \rceil$. The union of those intervals defines a set of axis-aligned and non-overlapping boxes in a four dimensional space. If two states have the same combinatorial frontier and have their first four coordinates falling into the same box, then they encode similar solutions.

The approximation algorithm proceeds in the same way as the dynamic programming Algorithm 1, except that we add a trimming phase. The trimming phase works as follows. If in a box, there are more than one state with the same combinatorial frontier, then we keep only one of them (chosen arbitrarily). We will denote by $\mathcal{U}_i$ the (untrimmed) state space obtained before performing that trimming phase at the i-th phase of the algorithm, and $\mathcal{T}_i$ the (trimmed) state space obtained after thinning out and trimming $\mathcal{U}_i$. Algorithm 2 fully describes the approximated dynamic programming algorithm.

The worst time complexity of this algorithm is $O(n \times vs(G) \times (L_1)^2 \times (L_2)^2 \times 4^{vs(G)})$. Since the size of an instance I is $\Theta(n + |E| + ln(p_{sum} + m_{sum}))$, this algorithm is therefore FPT with respect to the path-width. Let us also notice that if the tree-width is a constant h, then the time complexity remains polynomial since, as mentioned in Sect. 2, $vs(G) = O(\log(n) tw(G))$. Moreover, the tree-width of G being a constant h, we can construct a layout L such that $vs_L(G) = O(\log(n)h)$ in polynomial time by constructing a tree decomposition (T, X) of G with width h [1] and using works in [7,13].

Theorem 1. *There exists an FPT algorithm with respect to the path-width, which returns a solution for the problem $Pk|G, mem|C_{max}$ within a ratio of $(1 + \varepsilon)$ for the optimum makespan, where the memory capacity M_i, $1 \leq i \leq k$, of each machine may be exceeded by at most a factor $(1 + \varepsilon)$.*

Algorithm 2. The approximated dynamic programming algorithm.

input : A graph $G(J, E)$ where $L^*(j_i) = i$, for $1 \leq i \leq n$
output: A solution vector s

$\mathcal{S}_1 := \{[p_1, 0, m_1 + \sum_{j \in \mathcal{N}(j_1)} m_j, 0, C_1^1], [0, p_1, 0, m_1 + \sum_{j \in \mathcal{N}(j_1)} m_j, C_1^2]\}$;
$\mathcal{T}_1 := \mathcal{S}_1$;
foreach $i \leftarrow 2, n$ **do**
  $\mathcal{U}_i := \emptyset$;
  foreach $[s_1, s_2, s_3, s_4, C_{i-1}] \in \mathcal{T}_{i-1}$ **do**
    Compute α_i^1 and C_i^1;
    $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup [s_1 + p_i, s_2, s_3 + \alpha_i^1, s_4, C_i^1]$;
    Compute α_i^2 and C_i^2;
    $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup [s_1, s_2 + p_i, s_3, s_4 + \alpha_i^2, C_i^2]$;
  end
  Compute a trimmed copy $\mathcal{T}_i$ of $\mathcal{U}_i$;
end
$s \leftarrow [s_1, s_2, s_3, s_4, C_n] \in \mathcal{T}_n$ with $s_3 \leq (1+\varepsilon)M_1$ and $s_4 \leq (1+\varepsilon)M_2$ and such that $\max\{s_1, s_2\}$ is minimum;

As stated before we present here the proof when $k = 2$. In the conclusion we mention the general case when k is any fixed constant. The proof of this theorem relies on the following lemma.

Lemma 1. *For each state $S = [s_1, s_2, s_3, s_4, C_i] \in \mathcal{S}_i$, there exists a state $T = [s_1^\#, s_2^\#, s_3^\#, s_4^\#, C_i] \in \mathcal{T}_i$ such that*

$$s_1^\# \leq \Delta^i s_1 \text{ and } s_2^\# \leq \Delta^i s_2 \text{ and } s_3^\# \leq \Delta^i s_3 \text{ and } s_4^\# \leq \Delta^i s_4. \quad (1)$$

Proof. The proof of this statement is by recurrence on i. By construction $\mathcal{T}_1 = \mathcal{S}_1$ so the statement is true for $i = 1$. Now, let us assume that inequalities (1) hold for some index $i - 1$, and consider an arbitrary state $S = [s_1, s_2, s_3, s_4, C_i] \in S_i$. Then, S is computed from a state $[w, x, y, z, C_{i-1}] \in S_{i-1}$ and either $[s_1, s_2, s_3, s_4, C_i] = [w + p_i, x, y + \alpha_i^1, z, C_i^1]$ or $[s_1, s_2, s_3, s_4, C_i] = [w, x + p_i, y, z + \alpha_i^2, C_i^2]$ must hold. We assume that $[s_1, s_2, s_3, s_4, C_i] = [w + p_i, x, y + \alpha_i^1, z, C_i^1]$ as, with similar arguments, the rest of the proof is also valid when $[s_1, s_2, s_3, s_4, C_i] = [w, x + p_i, y, z + \alpha_i^2, C_i^2]$. By the inductive assumption, there exists a vector $[w^\#, x^\#, y^\#, z^\#, C_{i-1}] \in T_{i-1}$ such that

$$w^\# \leq \Delta^{i-1} w \text{ and } x^\# \leq \Delta^{i-1} x \text{ and } y^\# \leq \Delta^{i-1} y \text{ and } z^\# \leq \Delta^{i-1} z. \quad (2)$$

The trimmed algorithm generates the vector $[w^\# + p_i, x^\#, y^\# + \alpha_i^1, z^\#, C_i^1] \in \mathcal{U}_i$ and may remove it during the trimming phase, but it must leave some vector $[s_1^\#, s_2^\#, s_3^\#, s_4^\#, C_i^1] \in \mathcal{T}_i$ that is in the same box as $[w^\# + p_i, x^\#, y^\# + \alpha_i^1, z^\#, C_i^1]$. This vector $[s_1^\#, s_2^\#, s_3^\#, s_4^\#, C_i^1] \in \mathcal{T}_i$ is an approximation of $S = [s_1, s_2, s_3, s_4, C_i] \in S_i$ in the sense of (2). Indeed, its first coordinate $s_1^\#$ satisfies

$$s_1^\# \leq \Delta(w^\# + p_i) \leq \Delta(\Delta^{i-1} w + p_i) \leq \Delta^i w + \Delta p_i \leq \Delta^i (w + p_i) = \Delta^i s_1, \quad (3)$$

its third coordinate $s_3^{\#}$ satisfies

$$s_3^{\#} \leq \Delta(y^{\#} + \alpha_i^1) \leq \Delta(\Delta^{i-1}y + \alpha_i^1) \leq \Delta^i y + \Delta\alpha_i^1 \leq \Delta^i(y + \alpha_i^1) = \Delta^i s_3, \quad (4)$$

and its last coordinate C_i^1 is equal to C_i. By analogous arguments, we can show that $s_2^{\#} \leq \Delta^i s_2$ and $s_4^{\#} \leq \Delta^i s_4$. Our assumption is valid during the transition from phase $i-1$ to i, which completes the inductive proof.

□

Let us now go back to the proof of Theorem 1. At the end of phase n, the untrimmed algorithm (Algorithm 1) outputs the vector $s = [s_1, s_2, s_3, s_4, C_n]$ that minimizes the value $\max\{s_1, s_2\}$ such that $s_3 \leq M_1$ and $s_4 \leq M_2$. By Lemma 1, there exists a vector $[s_1^{\#}, s_2^{\#}, s_3^{\#}, s_4^{\#}, C_n] \in \mathcal{T}_n$ whose coordinates are at most a factor of Δ^n above the corresponding coordinates of s. We conclude that our algorithm (Algorithm 2) returns a solution such that the makespan is at most Δ^n times the optimal solution and the amount of memory for each machine is at most Δ^n its capacity. Moreover $\Delta^n \leq 1+\varepsilon$. Indeed, if we consider functions $f(x) = (1 + x/n)^n$ and $g(x) = 1 + 2x$, with $0 \leq x \leq 1$ and $n \geq 1$, we have

$$(1 + x/n)^n \leq 1 + 2x \quad (5)$$

since f and g are respectively a convex and a linear function in x and the inequality holds true at $x = 0$ and $x = 1$.

So we have constructed an algorithm that returns a solution such that the makespan is at most $(1+\varepsilon)$ times the optimal solution and the amount of memory for each machine is at most $(1+\varepsilon)$ its capacity. It ends the proof of Theorem 1.

We provide a non-intuitive optimal solution to $P2|G, mem|C_{max}$ on Fig. 5. On that Figure the connected set of colored jobs is assigned to machine 1 while the non-connected set of white jobs is assigned to machine 2.

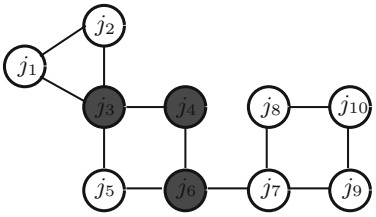


Fig. 5. Example of an optimal assignment to $P2|G, mem|C_{max}$ for the following instance : $M_l = 21, 1 \leq l \leq 2$; $p_{j_i} = m_{j_i}, 1 \leq i \leq 10$; $p_{j_1} = p_{j_2} = p_{j_3} = p_{j_5} = p_{j_7} = p_{j_9} = p_{j_{10}} = 1$; $p_{j_4} = p_{j_8} = 5$; $p_{j_6} = 9$. This optimal assignment induces a computation time of 11 to machine 1 (resp. 15 to machine 2) and a memory allocation of 21 to machine 1 (resp. 19 to machine 2). Therefore, the induced makespan is 15 and the memory capacity of each machine is satisfied.

5 Conclusion

Given 2 machines and a neighborhood graph of jobs with h-bounded linear-width or tree-width, we have constructed an algorithm that returns a solution if at least one solution exists for our scheduling problem with memory constraints. The output of this algorithm is generated in polynomial time and is such that the makespan is at most $(1+\varepsilon)$ times the optimal solution and the amount of memory for each machine is at most $(1+\varepsilon)$ its capacity. Moreover, we have constructed an algorithm that returns an optimal solution to our problem in pseudo-polynomial time.

This result can be extended to any constant number of machines as adding machines means increasing the number of dimensions of a state. It only requires to redefine the combinatorial frontier where $\sigma'_i(j)$ would express the machines on which j has not been assigned and which have memorized the data of j. This leads to a time complexity $O(n \times k \times vs(G) \times (L_1)^k \times (L_2)^k \times (k \times 2^k)^{vs(G)})$ where n is the number of phases; $k \times vs(G)$ is the time complexity to test whether two states S and S' are the same; $(L_1)^k \times (L_2)^k$ is the number of boxes induced by the algorithm; and $(k \times 2^k)^{vs(G)}$ is the number of distinct combinatorial frontiers. Notice that if the maximum degree of G is bounded by a constant d, we can lower the previous complexity as at most d machines can memorize the data of a task. Extending the result to unrelated machines can be easily carried over.

As the algorithm is FPT with respect to the path-width, it is particularly interesting for graphs with bounded path-width. Given the fact that such a graph does not occur naturally in large simulations on large meshes, we are wondering if there are FPT approximation algorithms with respect to more generic graph parameters such as the tree-width, and the local tree-width [2].

References

1. Bodlaender, H.L., Kloks, T.: Better algorithms for the pathwidth and treewidth of graphs. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 544–555. Springer, Heidelberg (1991)
2. Eppstein, D.: Diameter and treewidth in minor-closed graph families. Algorithmica **27**, 275–291 (2000)
3. Ern, A., Guermond, J.L.: Theory and Practice of Finite Elements. Applied Mathematical Sciences, vol. 159. Springer, New York (2004)
4. Gairing, M., Monien, B., Woclaw, A.: A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. Theor. Comput. Sci. **380**(1–2), 87–99 (2007)
5. Graham, R., Lawler, E., Lenstra, J., Rinnooy Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discrete Math. **5**, 287–326 (1979). Elsevier
6. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM **22**(4), 463–468 (1975)
7. Kinnersley, N.G.: The vertex separation number of a graph equals its path-width. Inf. Process. Lett. **42**(6), 345–350 (1992)

8. Korach, E., Solel, N.: Tree-width, path-width, and cutwidth. Discrete Appl. Math. **43**(1), 97–101 (1993)
9. Lenstra, J.K., Shmoys, D.B., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. Math. Program. **46**(3), 259–271 (1990)
10. LeVeque, R.J.: Finite Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge (2002)
11. Robertson, N., Seymour, P.: Graph minors. I. Excluding a forst. J. Comb. Theory Ser. B **35**(1), 39–61 (1983)
12. Saha, B., Srinivasan, A.: A new approximation technique for resource-allocation problems. In: Proceeding of Innovations in Computer Science (ICS), 342–357 (2010)
13. Skodinis, K.: Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. J. Algorithms **47**(1), 40–59 (2003)
14. Woeginger, G.J.: When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? INFORMS J. Comput. **12**(1), 57–74 (2000)
15. Woeginger, G.J.: A comment on scheduling two parallel machines with capacity constraints. Discret. Optim. **2**(3), 269–272 (2005)