Minimizing Memory in Parallel Task Graph Scheduling: Focusing on Average Consumption

Anne Benoit, Loris Marchal, Adrien Obrecht

Minimizing Memory in Parallel Task Graph Scheduling: Focusing on Average Consumption

Anne Benoit[a,c], Loris Marchal[b,a], Adrien Obrecht[a,*]

[a] *LIP – UMR 5668 – ENS Lyon, CNRS, Inria, UCBL,*
[b] *CNRS,*
[c] *Institut Universitaire de France,*

Abstract

The scheduling of applications represented as task graphs has been extensively studied in the literature, where the goal is often to minimize the total execution time. When accounting for the memory needs of tasks, many algorithms that strive to minimize the peak memory have been proposed, so that the execution can fit on a processor with limited memory. However, minimizing the peak memory may lead to a high memory consumption on average, which would deteriorate the performance when several such applications must be executed in parallel. In this case, one would hence rather minimize the average memory consumption over time for each application, since peak usage is likely not to happen at the same time for all applications. In this work, we formalize the problem of minimizing the average memory consumption of a task graph, with two different models, and we present optimal algorithms for some particular task graphs (k-chains, pumpkins). Building on these algorithms, we design heuristics for general graphs. We evaluate them through simulations, using both synthetic task graphs and task graphs coming from real applications, such as QR eliminations. The results show that the heuristics often return solutions close to the optimal for a single task-graph application, and achieve in addition reasonably small peak memory usage. In a parallel setting, as we anticipated, minimizing the average memory of each application turns out to be more efficient than using a classical algorithm that focuses on minimizing the peak memory, hence demonstrating the usefulness of average memory consumption optimization.

Keywords: Scheduling, Task graphs, Memory, Average vs peak memory usage

1. Introduction

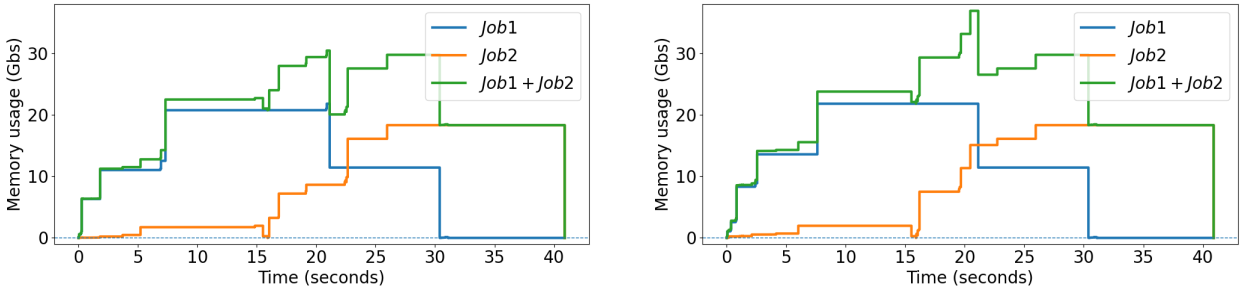
Many high-performance computing applications are characterized by substantial memory requirements. This challenge has become even more pronounced with the emergence of modern computational workloads, such as large data analytics or deep neural network training. As the scale of these workloads continues to grow, optimizing memory usage has become increasingly critical, motivating extensive research efforts aimed at reducing their memory footprint.

Among the approaches that have been explored, the scheduling of the operations within an application is an important leverage to reduce its memory demand without changing the total execution time. From the seminal work of Hong and Kung [1], several techniques have been derived, most of them trying to minimize the peak memory usage of a computation, that is, the maximum amount of memory required at any point during the computation. This works particularly well for applications modeled as task graphs, where vertices represent computation modules, or tasks, while edges represent the dependencies between these tasks in the form of input/output data [2]. Several optimal algorithms have been proposed to minimize peak memory on special classes of graphs for their sequential processing [3, 4, 5], while both theoretical and practical scheduling strategies have been proposed for this problem when dealing with the parallel processing of task graphs [6, 7, 8].

*Corresponding author.
Email address: adrien.obrecht@ens-lyon.fr (Adrien Obrecht)

The motivation behind this focus on the peak memory is to avoid costly input/output operations, by ensuring that a system is equipped with sufficient memory to handle the highest memory demand over time. However, many systems concurrently run applications with varying memory demands that must share a common memory. This is in particular the case for cloud-based applications, where vertical scaling dynamically reallocates memory among jobs based on their demand [9, 10]. In large-scale HPC systems, the use of disaggregated memory is increasingly advocated: a large amount of memory can be shared among the nodes of a cluster, instead of providing each of them with its own limited quantity of private memory [11, 12]. In this context, memory allocation and reallocation strategies have been proposed to optimize the execution of HPC applications with disaggregated memory [13, 14].

When multiple computations are processed concurrently on a shared-memory system, each computation is scheduled independently from all others and it is rare that their memory peaks coincide in time. Moreover, while reducing the peak memory of each application is essential, it does not necessarily lead to a good memory behavior for the parallel execution of several applications. Indeed, minimizing the peak memory for each application may cause each application to use a large amount of memory on average. We claim that minimizing the **average memory usage** of each application is a better way to reduce the total memory demand. We illustrate this claim with an experiment where QR eliminations are performed in parallel (see Figure 1 and more details and results in Section 6): targeting the peak of each individual task graph results in a higher overall memory usage than targeting the average.



(a) Memory profile where each job minimizes average memory consumption

(b) Memory profile where each job minimizes peak memory consumption

Figure 1: Comparison of memory profiles for the execution of QR elimination task graphs on two concurrent jobs. Each job schedules two QR task graphs in a row. The time axis is a representation that assumes that each process can perform a constant 1 Teraflop/s. The memory profiles for the individual jobs are in blue and orange, and the cumulated sum is in green. The maximum cumulated memory used when minimizing the peak memory for each job is 36.9Gb, while it is only 30.4Gb when minimizing the average.

The move to the average memory objective can also be motivated by an analogy with the caching problem, where recent studies suggest that optimizing the average cache usage for sequences of page accesses can significantly improve performance when multiple sequences are served in parallel [15].

In this paper, we make several contributions towards the efficient parallel execution of task graphs with limited shared memory. We first study the challenge of minimizing the average memory usage for the sequential processing of a single application modeled as a task graph. We then explore the effects of scheduling multiple task graphs in parallel on a shared-memory system, with the goal of optimizing the overall memory efficiency. We acknowledge that this is only a first step in the direction of efficient memory sharing, as we do not deal with the parallel execution of each task graph, the complex reality of memory hierarchies and interactions between resources. However, the proposed strategies and the claim that is it meaningful to consider average memory in practice could serve as a basis for further developments.

The main contributions are the following:

- We formally define the problem of optimizing the average memory of a task graph for two different memory models, depending on whether the data produced by a task is the same for all its successors or not (Section 2).
- We propose optimal algorithms for some specific classes of graphs (namely k-chains and pumpkin graphs) in Section 4.
- Based on the previous findings, we design heuristics to optimize the processing of general graphs (Section 5).
- We present an experimental evaluation of the heuristics through simulations on both synthetic and real-life task graphs (Sections 6.2 and 6.3).

- We demonstrate through simulations that minimizing the average memory of each application is indeed a better way to optimize the total memory when processing several applications in parallel than minimizing the peak memory of each application (Section 6.4).

Related work is discussed in Section 3. We conclude and discuss future work directions in Section 7.

2. Model

We focus on task graphs represented by a directed acyclic graph (DAG), G. We denote by $V = \{a_1, \ldots, a_n\}$ the set of vertices of G, where each vertex represents a task. Each task $a_i \in V$ takes a fixed time t_i to be executed. E is the set of edges of G, which represent data dependencies between tasks. We denote by $V^+(a_i)$ the set of successors of a_i in G. If $a_j \in V^+(a_i)$, then a_i produces some output data that is required by a_j as input data. This means that we need to execute task a_i before task a_j. We are looking for a sequential ordering of the tasks representing their order of computation.

Memory models. For the sake of completeness, we consider two distinct models for representing the output data: the *single data* model and the *multiple data* model, as both models are used in the literature.

In the *single data* model, each task a_i produces only one piece of data as output, with a size of w_i. This data is stored in memory as soon as task a_i begins its execution and remains there until all of its successor tasks have started. Once all successor tasks in $V^+(a_i)$ have begun their execution, the data produced by a_i can safely be removed from memory. This model, illustrated in Figure 2, has been used in previous scheduling work [5]. It is well suited to represent some linear algebra computations such as Cholesky or LU decompositions, where computational tasks require several input tiles but produce a single output tile.

In contrast, in the *multiple data* model, each task generates distinct output data for each of its successors. Each piece of data is removed from memory as soon as its corresponding successor starts execution. In this model, each output data can have a unique size, so we denote the weight of the edge $(a_i, a_j) \in E$ as $w_{i,j}$, representing the size of the data generated by task a_i for task a_j. This model is illustrated in Figure 3 and has also been used in several scheduling studies [4, 16]. It is required, for instance, when a fork task of a scientific workflow produces specific data for its successors.

For both models, we represent the associated weighted DAG as G. While the distinction in edge weights may seem subtle, it leads to different optimization challenges and algorithmic solutions. Note that the model assumes fixed and known data sizes and cannot cope with computations where data sizes depends on the input data. However, this is a common assumption in previous scheduling work. We now detail the associated optimization problems.

Optimization problems. For both memory models, the problem is to find a schedule that minimizes the average memory consumption of the processor, given a task graph G. All tasks have to be ordered for their sequential processing, while respecting the constraints imposed by the graph structure. Hence, the execution always takes a total time of $t_{\max} = \sum_{i=1}^{|V|} t_i$. We formally define a schedule:

Definition 1 (Schedule)**.** *Let G be a weighted DAG. A schedule is represented by a function ϕ that associates to each task its starting time ($\phi : V \to \{0, \ldots, t_{\max}\}$). It is a valid schedule if and only if:*

- $\forall a_i, a_j \in V, \phi(a_j) \geqslant \phi(a_i) + t_i$ *or* $\phi(a_i) \geqslant \phi(a_j) + t_j$ *(no task overlap)*
- $\forall a_i \in V, \phi(a_i) + t_i \leqslant t_{\max}$ *(no idle time)*
- $\forall (a_i, a_j) \in E, \phi(a_i) < \phi(a_j)$ *(data dependencies)*

Since we enforce that all tasks are finished at time $t_{\max}$, the schedule is totally defined by an ordering of the tasks L_ϕ, also called an arrangement. For instance, $L_\phi = (1, 2, 3, 4, 5, 6, 7, 8)$ corresponds to a valid schedule ϕ for the graph in Figure 2 with eight tasks.

Given a schedule ϕ, in the *single data* model, the data of weight w_i stays in memory for a duration $\max_{a_j \in V^+(a_i)} \phi(a_j) - \phi(a_i)$, i.e., from the start of task a_i until the last successor a_j has started its execution. In order to minimize the average memory usage, we aim at minimizing the weighted sum of the time each data stays in memory. This cost function has already been studied in the literature under the name of weighted sum cut [17].

Definition 2 (Weighted Sum Cut (*WSC*))**.** *Given a valid schedule ϕ for a DAG G, the weighted sum cut of ϕ is:*

$$WSC_G(\phi) = \sum_{a_i \in V} w_i \max_{a_j \in V^+(a_i)} (\phi(a_j) - \phi(a_i)).$$

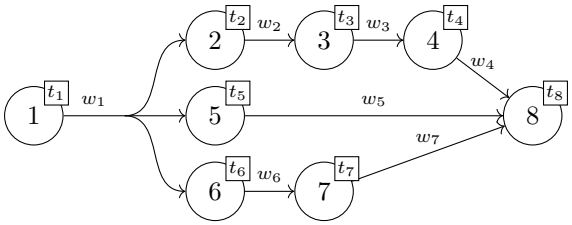


Figure 2: Pumpkin graph in the *single data* model.

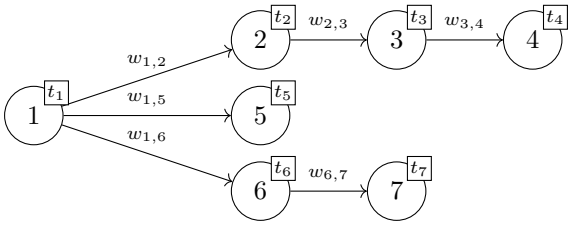


Figure 3: 3-chain graph in the *multiple data* model.

For the *multiple data* memory model, the data between a_i and a_j stays in memory for a duration $\phi(a_j) - \phi(a_i)$. This leads to another known function on graphs, called the Weighted Linear Arrangement. This quantity appears naturally on graphs and has already been studied fairly extensively, see for instance the work by Adolphson et al.[18, 19]

Definition 3 (Weighted Linear Arrangement (*WLA*))**.** *Given a valid schedule ϕ for a DAG G, the Weighted Linear Arrangement cost of ϕ is:*

$$\mathit{WLA}_G(\phi) = \sum_{(a_i,a_j)\in E} w_{i,j}\ (\phi(a_j) - \phi(a_i)).$$

For the definition of both cost functions, we consider that the output data of task a_i is produced as soon as the task starts. We made this choice for the sake of simplicity only, and it has no impact on the problem: considering that the data is produced at the end of a task would only change the total cost of a schedule by a constant.

Both objective functions are directly related to the average memory consumption of a schedule, as we can obtain the average memory used by dividing *WSC* or *WLA* by the makespan of the schedule, which is a constant. Hence, we focus on the study of minimizing *WSC* and *WLA*.

Finally, we give a simple definition of the peak memory, following previous work by Kayaaslan et al. [4]:

Definition 4 (Peak memory)**.** *For both memory models, the peak memory consumption of a schedule ϕ is the highest amount of memory consumed throughout the execution of ϕ. More formally, we define:*

$$\mathit{PEAK}_{\text{Mult. Data}}(\phi) = \max_{1\leqslant t\leqslant t_{\max}} \sum_{\phi(a_i)\leqslant t<\phi(a_j)} w_{i,j}.$$

$$\mathit{PEAK}_{\text{Sing. Data}}(\phi) = \max_{1\leqslant t\leqslant t_{\max}} \sum_{\substack{\exists a_j\in V^+(a_i)\\ \phi(a_i)\leqslant t<\phi(a_j)}} w_i.$$

Graph types. Minimizing the average or the peak memory on general directed graphs is NP-complete on both models (see Section 3), hence we consider some special classes of graphs to assess the problem complexity.

Definition 5 (In-tree, out-tree, k-chain)**.** *An in-tree (resp. out-tree) is a directed graph where all the vertices have at most one outgoing (resp. incoming) edge. An out-tree T is a k-chain if its root has out-degree k and it is the only node with multiple successors (see Figure 3 for an example of a 3-chain).*

Definition 6 (Pumpkin). *A graph G is a pumpkin graph if it has one source of out-degree k and in-degree 0, one target of in-degree k and out-degree 0, and all other nodes have in-degree and out-degree 1 (see Figure 2 for an example).*

Definition 7 (Series Parallel (SP)). *G is a SP graph if:*

- *G consists of two nodes and a single edge, $s \to t$; or*
- *G is the series composition of two SP graphs (merging the target of the first with the source of the second); or*
- *G is the parallel composition of two SP graphs (merging both sources and targets respectively).*

3. Related work

While most of the literature on task graph scheduling focuses on makespan minimization, there have been several studies on graph traversals for minimizing the peak memory, in particular in the domain of numerical linear algebra. The first studies concentrated on tree-shaped task graphs. Liu [20] described how the storage requirements and computational dependencies of matrix factorization can be represented by in-trees, and has later shown how to find a peak memory-minimizing traversal of the tree [21, 3]. The problem of scheduling a task graph under memory constraints also appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics, and geophysical simulations. The problem of task graphs handling large data has been identified by Ramakrishnan et al. [22], who propose some simple heuristics. In the context of quantum chemistry computations, Lam et al. [23] also considered task trees. More recently, progress has been made towards more general graphs: Kayaaslan et al. [4] have proposed an optimal algorithm for SP-graphs under the *multiple data* model, while the problem has been shown NP-complete under the *single data* model by Jin et al. [5], who also propose a polynomial-time algorithm for SP-graphs with bounded degree. Minimizing the peak memory for general directed graphs is NP-complete even in the *single memory* model, as it generalizes the pebble game introduced by Sethi and Ullman [24, 25]. Scheduling these graphs in a parallel setting has also been studied for the peak minimization problem: Eyraud et al. [6] proved inapproximability results even for unit-weight trees, while others proposed dynamic scheduling heuristics for the case of limited memory [7, 8].

The *WSC* and *WLA* problems, related to minimizing the average memory under the *single data* and *multiple data* problems, have both been shown NP-complete on general directed graphs [26, 27]. Sum-cut problems have been studied on directed graphs only in the unweighted case; Bossart et al. [27] have proposed an algorithm for out-trees in $\mathcal{O}(n \log(n))$. A polynomial-time algorithm for a specific subclass of SP graphs (reduced 2-terminal series parallel graphs, r-2TSPG) has also been designed by Achouri et al. [28].

For (unweighted) Linear Arrangement problems, linear algorithms have been proposed for out-trees and pumpkins [29]. One of the closest results to the present study is the algorithm from Adolphson and Hu [18], which presents a solution for the *WLA* problem on out-trees, with complexity $\mathcal{O}(n \log(n))$. Lemmas 1 and 2 in Section 4 show that this algorithm can be also used on in-trees for *WLA* and *WSC*. In Section 4.3, we make use of this algorithm and extend some of its follow-up results to out-trees for the *WSC* problem.

Previous algorithms for the unweighted case of Linear Arrangement and Sum Cut [28, 27] rely on simple heuristics like BFS or largest subtree first that heavily exploit the structure of the graph. It turns out that these techniques

Problem Type	Weighted	Unweighted
Linear Arrgt. (multiple data memory model)	General: NP-C [26]	General: NP-C [26]
	Out-tree: $\mathcal{O}(n \log(n))$ [18]	Out-tree: $\mathcal{O}(n)$ [29]
	In-tree: $\mathcal{O}(n \log(n))$ Lemma 1	**In-tree**: $\mathcal{O}(n)$ Lemma 1
	Pumpkin: $\mathcal{O}(n \log(n))$ Theorem 1	Pumpkin: $\mathcal{O}(n)$ [29]
Sum Cut (single data memory model)	General: NP-C [27]	General: NP-C [27]
	In-tree: $\mathcal{O}(n \log(n))$ Lemma 2+[18]	Out-tree: $\mathcal{O}(n)$ [27]
	k-**chain**: $\mathcal{O}\left(n^{k+1} \log(n)\right)$ Theorem 2	**In-tree**: $\mathcal{O}(n)$ Lemma 2
	Pumpkin: $\mathcal{O}\left(n^{k+1} \log(n)\right)$ Theorem 3	r-2TSPG: $\mathcal{O}\left(n^2\right)$ [28]

Table 1: Main results for the two memory models (new contributions are highlighted).

cannot be applied in the weighted case and we prove in Appendix A.1 and Appendix A.2 that they can produce arbitrarily bad results.

Finally, note that both peak memory minimization and average memory minimization are variants of Linear Arrangement and Sum Cut problems on graphs, which have been extensively studied in the undirected case [30].

Table 1 summarizes the existing results for Sum Cut and Linear Arrangement problems, and highlights the main contributions of this paper.

4. Optimal algorithms

Since the problems are NP-complete for general graphs, we focus on special classes of graphs in order to derive exact algorithms. We start with some preliminaries in Section 4.1. Then, we first present an optimal algorithm for *WLA* on pumpkins in Section 4.2, before discussing the more complicated *WSC* problem in Section 4.3.

4.1. Preliminaries

Before designing algorithms for the target optimization problems, we present a few reductions between different problems on trees. These equivalences will be used as subroutines later on, allowing us to transform out-trees into in-trees. Because they rely on very simple graph transformations, these reductions do not add any algorithmic complexity to the problems. Hence, Lemma 1 shows the equivalence between in-trees and out-trees for the *WLA* problem, while Lemma 2 shows the equivalence of *WLA* and *WSC* on in-trees.

Definition 8 (Reverse graph and schedule)**.** *Let $G = (V, E)$ be a directed graph. We define its reverse graph $G' = (V, E')$ such that $(a_i, a_j) \in E' \Leftrightarrow (a_j, a_i) \in E$. Given a schedule ϕ for G such that $L_\phi = (a_1, \dots, a_n)$, we define its reverse schedule ϕ' such that $L_{\phi'} = (a_n, \dots, a_1)$.*

Lemma 1. *Let G be a weighted out-tree and ϕ be a schedule for G. Let G' be the reverse graph of G and ϕ' be the reverse schedule of ϕ. Then, ϕ' is an optimal schedule for WLA on G' if and only if ϕ is an optimal schedule for WLA on G.*

Proof sketch. One can first easily verify that ϕ' is a schedule for G'. We then prove that the cost of ϕ for G and ϕ' for G' only differ by a constant, and derive the optimality of ϕ'. The full version of this proof can be found in Appendix A.3. □

Lemma 2. *Let T be an in-tree and ϕ a valid schedule for T. Then, $WSC_T(\phi) = WLA_T(\phi)$.*

Proof. For in-trees, each node has exactly one successor, hence both objective functions are equal:

$$\begin{aligned} WSC_T(\phi) &= \sum_{u \in V} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\ &= \sum_{(u,v) \in E} w_{u,v}(\phi(v) - \phi(u)) = WLA_T(\phi). \end{aligned}$$

□

4.2. Weighted Linear Arrangement on pumpkins

In this section, we focus on the *multiple data* memory model, and we design an $\mathcal{O}(n \log(n))$ algorithm to solve *WLA* on pumpkins. The original idea of splitting the graph at its min-cut originates from work by Kayaaslan et al. [4], but the proofs are very different considering that we do not have the same objective function. We first note that we can easily find a min-cut in the pumpkin, by decreasing the weights on each chain of the pumpkin, thanks to Lemma 3.

Lemma 3. *Let G be a pumpkin graph of sink a_t, with $c = (a_1, \dots, a_i)$ a chain of G with non-zero weights. If G' is the graph where all the weights along c are decreased by 1, then an optimal schedule for WLA on G' is optimal for WLA on G.*

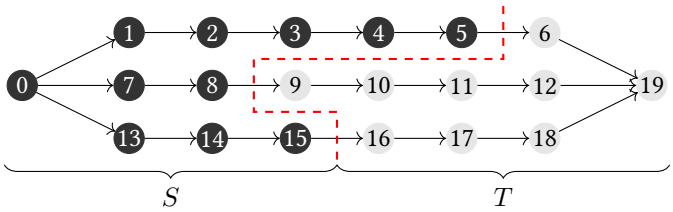
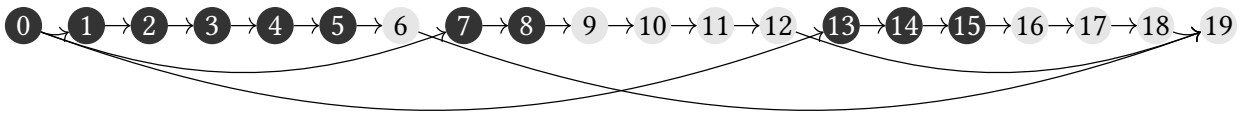
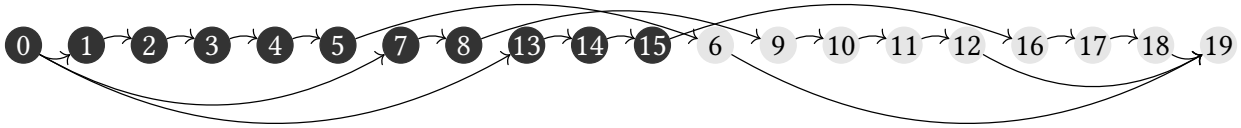


Figure 4: A pumpkin graph G with its min-cut (S, T) in red. Nodes before the cut (in S) are represented in black, while nodes after the cut (in T) are in white.



(a) A schedule ϕ valid for the graph G of Figure 4. We take the lexicographic schedule for simplicity.



(b) Transformed schedule $\psi = \phi[S] + \phi[T]$, where we first schedule all nodes before the min-cut (in black), then the nodes after the min-cut (in white). The resulting schedule is still valid and does not increase in cost.

Figure 5: We illustrate how to transform a schedule ϕ valid for the graph G of Figure 4 into a new schedule synchronizing at the min-cut.

Proof sketch. Because of the structure of the graph, data from the chain c is stored in memory up until the last node is executed. Whatever the considered schedule, this data stays the same amount of time in memory, therefore adding one to the whole chain only changes the cost by a constant. The detailed proof can be found in Appendix A.4. □

Given a pumpkin graph G, applying this lemma multiple times tells us that we can transform G into an equivalent pumpkin graph where each chain has at least one edge of weight 0. In particular, we can transform G into a pumpkin G' of min-cut 0, and such that if ϕ is optimal for *WLA* on G', it is optimal for *WLA* on G.

The idea is then to schedule first all the tasks before the cut, and then the remaining tasks, hence dealing with two trees on each side of the cut. An example of a transformation from any schedule to one synchronized at the cut is given in Figure 5 for the graph shown in Figure 4. Before formalizing this lemma, we first need to define the concept of a sub-schedule.

Definition 9 (Sub-schedule)**.** *Let $G = (V, E)$ be a DAG, and $S \subseteq V$ a subset of the vertices. Given a schedule ϕ of G with $L_\phi = (a_1, \dots, a_n)$, we define $\phi_{[S]}$ such that $L_{\phi_{[S]}} = (a'_1, \dots, a'_j)$ with $\{a'_1, \dots, a'_j\} = S$, as the schedule following the same task order as in ϕ on the subset of tasks S.*

Lemma 4. *Given a pumpkin graph G with a directed cut (S, T) of weight 0, there is an optimal schedule ϕ for WLA on G such that for all $u \in S$ and $v \in T$, we have $\phi(u) < \phi(v)$ (u is scheduled before v).*

Proof sketch. The schedule is divided into two parts, hence we use the notion of sub-schedule introduced in Definition 9. Let ϕ be an optimal schedule for *WLA* on G. We define $\psi = \phi_{[S]} + \phi_{[T]}$, as shown in Figure 5, where all tasks of S are scheduled before tasks of T. We prove that ψ is a valid and optimal schedule for *WLA* on G. The optimality is proven by comparison with ϕ: for any edge of the pumpkin, we prove that the transformation from ϕ to ψ strictly decreases the contribution of this edge to the final cost. The full case disjunction can be found in Appendix A.5. □

Lemma 4 tells us that we will always be able to find an optimal schedule for *WLA* on a pumpkin that stops at the min-cut. In particular, it means that we can solve both sides of the min-cut independently. This allows us to reuse algorithms that were developed for out-trees and in-trees, and to obtain a schedule for pumpkins.

Theorem 1. *Given a pumpkin graph G and a min-cut (S, T), let ϕ_S be an optimal schedule for WLA on the out-tree S, and ϕ_T be an optimal schedule for WLA on the in-tree T. Then, $\phi = \phi_{[S]} + \phi_{[T]}$ is an optimal schedule for WLA on G, and it can be found in time $\mathcal{O}(n \log(n))$.*

Proof sketch. This theorem is a direct implication from Lemmas 3 and 4. Since finding a min-cut in a pumpkin can be done in linear time, and that solving *WLA* on trees takes time $\mathcal{O}(n \log(n))$, the schedule can be constructed in time $\mathcal{O}(n \log(n))$. □

4.3. Weighted Sum Cut on k-chain trees and pumpkins

The problem turns out to be much more complicated for *WSC* and the *single data* model, because we cannot easily decrease weights on each chain. Indeed, the initial data is shared by all chains of the pumpkin. Actually, the problem is even difficult on k-chains. Hence, we start the study on k-chains, before extending the algorithm for pumpkins.

***WSC* on k-chains.** The main source of inspiration for this section is the paper published in 1973 by Adolphson & Hu [18], which considers the Linear Arrangement problem on out-trees. Although our proof is slightly different because of the different graph and objective function, there are some similarities, since we reuse some definitions. We follow the same proof principle, but some additional difficulties emerge from the *WSC* problem. After proving some useful results on the shape of an optimal schedule for *WSC*, we exhibit the real difficulty of the problem, and we give an $\mathcal{O}\big(n^{k+1}\big)$ exact algorithm.

Step 1 – Obtaining a compressed graph. We define below a quantity on vertices that directly correlates to the schedule order of nodes. In the original paper by Adolphson & Hu [18], this was sufficient to find an optimal schedule. Here, we need some additional work because of the very first edge of the chain that has a shared dependency with all the chains.

Definition 10 (Ratio)**.** *Let G be a k-chain tree and a_i, a_j, a_k elements of V such that $(a_i, a_j) \in E$ and $(a_j, a_k) \in E$. We define the ratio of a_j by:*

$$R_G(a_j) = \frac{w_{a_i,a_j} - w_{a_j,a_k}}{t_j}.$$

Whenever it is possible, we drop the index G from this ratio.

Definition 11 (Compressed graph)**.** *A k-chain graph G is a compressed graph if and only if:*

- *The weight of the edges is decreasing along each chain.*
- *The ratio of each node is decreasing along each chain.*

Lemma 5. *Let G be a k-chain tree of size n. We can construct in time $\mathcal{O}(n \log(n))$ a compressed graph G' that is equivalent to G, such that we can easily (in linear time) transform an optimal schedule for WSC on G' into an optimal schedule for WSC on G.*

Proof sketch. This proof is fairly long and involves defining new properties on edges, following what has been done in previous works [18], see Appendix A.6 for details. □

Because this proof is constructive, the main take-away from Lemma 5 is that we can construct a compressed graph G' that is equivalent to G, in that solving the *WSC* problem on G' gives the optimal schedule for G. From now on, we always work on the compressed representation of G.

Step 2 – Splitting the schedule. After having studied the local structure of the chains to get a compressed representation of G, we now delve into the global structure of a schedule. Let us consider a k-chain G of root s. It turns out that there are two very distinct phases in an optimal schedule nodes that are performed before all successors of the root s are treated, and nodes that are ordered after all these successors. The relative order of nodes heavily depends on which phase they belong to. For this, we split a schedule ϕ into two phases.

Definition 12 (Phases of a schedule)**.** *Let s be the root of the k-chain. We say that a_i is in the first phase of the schedule ϕ if $\phi(a_i) \leqslant \max_{a_j \in V^+(s)} \phi(a_j)$. Otherwise, we say that u is in the second phase of the schedule. An example of the two phases of a schedule is given in Figure 6.*

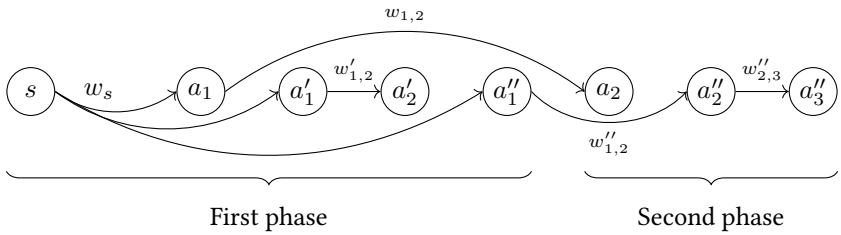
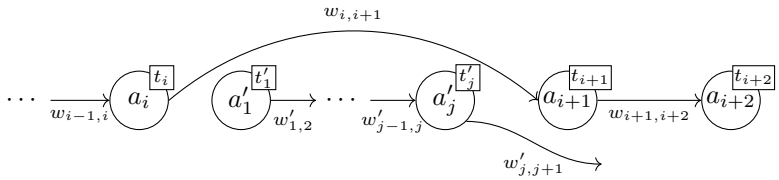
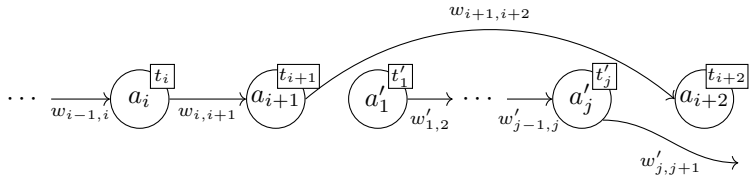


Figure 6: Example of the two distinct phases of a schedule for k-chains on a 3-chain graph with chains $s \to a_1 \to a_2$, $s \to a'_1 \to a'_2$, and $s \to a''_1 \to a''_2 \to a''_3$. The first phase ends when the data from s is removed from memory, i.e., when a_1, a'_1 and a''_1 have been started.



(a) Part of a schedule ϕ for a k-chain graph that has a crossing in the first phase. The chain $a'_1, \cdots a'_j$ starts between a_i and a_{i+1}.



(b) Part of the modified schedule ϕ', which differs only in the fact that the chain $a'_1, \cdots a'_j$ is now between a_{i+1} and a_{i+2}

Figure 7: Transformation of a schedule that presents a crossing in its first phase.

The intuition behind these two phases is the following: in the first phase, we have a data that is shared among all chains, and we cannot use the traditional *WLA* algorithm. However, in the second phase, we have no more data in common, and we can reuse the same results as in the *multiple data* case.

Now that these two phases are properly defined, we show that they both have unique properties, that make scheduling nodes inside each phase a simple process. For the first phase, we can associate to each chain a new quantity that directly defines how it should be ordered. For the second phase, the nodes can be scheduled by order of decreasing ratio, like what was done in the *WLA* case [18]. Both orderings are very easy to compute. This emphasizes that the real difficulty of the problem lies in finding the cut in the graph that defines both phases. This also paves the way to an $\mathcal{O}(n^k)$ algorithm to solve the *WSC* problem on k-chain trees.

The next two lemmas handle the schedule of the chains in the first phase of the schedule. We show that the chains do not cross each other. As soon as we start scheduling a chain in the first phase, we schedule the rest of the nodes of this chain immediately after. We then show that starting each chain by order of decreasing relative "weight" gives an optimal ordering.

Lemma 6. *Let T be a k-chain tree and ϕ an optimal schedule for T. In the first phase of ϕ, once a chain starts being scheduled, it will continue until the first phase ends.*

Proof. We only focus on two chains that cross in the first phase of the schedule of ϕ. It is as if the chains start with a weight of 0 as their associated data is kept in memory during the whole phase 1. We let $a'_1, \ldots, a'_j$ be the part of the chains that crossed another one, between a_i and a_{i+1}, as depicted in Figure 7.

We consider the schedule ϕ', whose only difference with ϕ is that a_{i+1} is scheduled before the elements $a'_1, \ldots, a'_j$. It is a valid schedule as the two chains are independent. Thus, we have $WSC(\phi) \leqslant WSC(\phi')$. This gives

(a) ϕ (b) ϕ'

Figure 8: Exchanging two nodes in the second phase of a schedule.

an immediate contradiction:

$$\begin{aligned} \mathit{WSC}(\phi) &\leqslant \mathit{WSC}(\phi') \\ \sum_{l=1}^{j} t'_l w_{i,i+1} + t_{i+1} w'_{j,j+1} &\leqslant \sum_{l=1}^{j} t'_l w_{i+1,i+2} \\ w_{i,i+1} &< w_{i+1,i+2} \end{aligned}$$

This is a direct contradiction with the definition of a compressed graph, that says that the weight of the edges should be decreasing along the chains. Therefore, an optimal schedule cannot have such a crossing in the first phase of a schedule. □

Lemma 7. *Suppose we know that the chains $c_1 = (a_{1,1}, \ldots, a_{1,k_1}), \ldots, c_{k-1} = (a_{k-1,1}, \ldots, a_{k-1,k_{k-1}})$ belong to the first phase of the schedule. If w_i is the weight of the outgoing edge from chain c_i, then ordering the chains by decreasing value of $t_i / \sum_{j=1}^{k_i} w_{i,j}$ gives an optimal solution to WSC.*

Proof sketch. We know from Lemma 6 that the first phase of the schedule only has chains scheduled continuously. We prove by contradiction that any schedule that is not ordered by decreasing value of $t_i / \sum_{j=1}^{k_i} w_{i,j}$ is sub-optimal. The full proof can be found in Appendix A.7. □

The next lemma deals with the ordering of nodes in the second phase.

Lemma 8. *Let T be a k-chain tree and ϕ an optimal ordering for WSC. If $a_1, \ldots, a_n$ are the nodes of the second phase of ϕ, then they are scheduled in order of non-increasing ratio. We have, for all $i < k$, $R(a_i) \geqslant R(a_{i+1})$.*

Proof. Let G be a $k-$chain graph. Suppose there exists an optimal schedule ϕ that has some increasing ratios in the second phase of the schedule. Let us say that we have $R(a_i) < R(a_j)$ and $\phi(a_i) + 1 = \phi(a_j)$.

Let ϕ' be the copy of ϕ except that a_i and a_j are switched. ϕ and ϕ' are represented in Figure 8. It is a valid schedule as a_i and a_j are from two different chains (we work on a compressed graph, so ratios are already decreasing along a chain). From the optimality of ϕ, we get:

$$\begin{aligned} \mathit{WSC}(\phi) &\leqslant \mathit{WSC}(\phi') \\ w_{i,i+1} t_j + w_{j-1,j} t_i &\leqslant w_{i-1,i} t_j + w_{j,j+1} t_i \\ \frac{w_{j-1,j} - w_{j,j+1}}{t_j} &\leqslant \frac{w_{i-1,i} - w_{i,i+1}}{t_i} \\ R(a_j) &\leqslant R(a_i) \end{aligned}$$

This contradicts the hypothesis that the ratios were increasing. □

Thanks to the previous lemmas, we are now able to design an algorithm to solve the *WSC* problem on k-chains, as expressed by the following theorem.

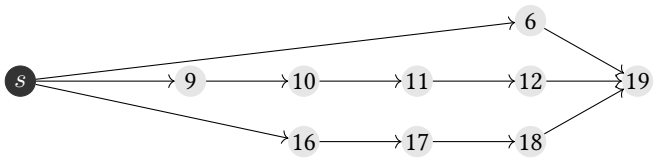


Figure 9: The graph G' corresponding to graph G of Figure 4.

Theorem 2. *Let G be a k-chain graph with chains of size $n_1, \dots, n_k$.*

Given a directed cut C, if there exists an optimal ordering ϕ for WSC on G such that C is exactly the first and second phase of ϕ, then we can find an optimal ordering for WSC in $\mathcal{O}(n \log(n))$.

Hence, there is an $\mathcal{O}\left(\prod_{i=1}^{k} n_i n \log(n)\right)$ algorithm to find an optimal solution to the WSC problem, by enumerating all possible cuts.

Proof. The proof mainly relies on the previous two lemmas, Lemma 7 and Lemma 8. Lemma 7 states that given the nodes in the beginning of the schedule, one can find an optimal arrangement of the chains by sorting them by decreasing value of $t_i / \sum_{j=1}^{k_i} w_{i,j}$. Lemma 8 states that sorting the end of the schedule by decreasing ratios gives an optimal solution. Both steps can be done in $\mathcal{O}(n \log(n))$ time. Because we know that there is an optimal ϕ that has such a set of beginning and end, and that both sides are solved optimally, this gives us an optimal solution for the whole graph.

Because all the cuts are enumerated, we are sure to find at least one optimal schedule, thus the algorithm is correct. There are exactly $\prod_{i=1}^{k} n_i$ cuts possible in G, as we can cut the chain i at n_i different places (we considered that not taking the first node of the chain is not possible in the beginning of the schedule). Then, the algorithm to find the corresponding schedule runs in $\mathcal{O}(n \log(n))$, hence the final complexity. □

***WSC* on pumpkins.** We are now ready to deal with pumpkins, building upon the optimal algorithm for *WSC* on k-chains. Since the algorithm is exponential for k-chains (see Theorem 2), we also derive an exponential algorithm for pumpkins. We even prove that we cannot find an exact algorithm with better complexity for pumpkins than for k-chain trees, using a polynomial reduction.

We extend the definition of first and second phase from k-chains to pumpkin graphs. The first phase is still constituted of the nodes scheduled before the first data is removed from memory. It turns out that the lemmas for the first part of the schedule are still valid for pumpkins. In particular, Lemma 7 states that given the cut (S, T) of the first and second phase of an optimal schedule, one can find an optimal schedule for the first phase in $\mathcal{O}(n \log(n))$.

Let $G = (V, E)$ be a pumpkin graph, of chains $c_1, \dots, c_k$. Let (S, T) be a cut of G. We define the graph $G' = (T \cup \{s\}, E')$ such that:

- $\forall (u, v) \in E, u \in T \Leftrightarrow (u, v) \in E'$;
- $\forall (u, v) \in E, u \in S \wedge v \in T \Leftrightarrow (s, v) \in E'$.

An an example, Figure 9 gives the graph G' corresponding to the graph G of Figure 4.

Lemma 9. *Let ϕ' be an optimal schedule for WLA on G', excluding s. If there exists an optimal schedule ϕ for WSC on G of first phase S and second phase T, then $\phi_{[S]} + \phi'$ is also an optimal schedule for WSC on G.*

The proof of this lemma is mainly computational, it is detailed in Appendix A.8. The main take-away is that the second phase of the schedule can be found by running the algorithm for *WLA* on in-trees, which runs in $\mathcal{O}(n \log(n))$.

Theorem 3. *Let G be a pumpkin graph with k chains of size $n_1, \dots, n_k$. We can find an optimal schedule for WSC on G in time $\mathcal{O}\left(\prod_{i=1}^{k} n_i n \log(n)\right)$.*

Proof. It follows the same idea as for Theorem 2. Given an optimal schedule ϕ for *WSC* on G, we consider the cut (S, T) such that $\phi_{[S]}$ is the first phase of ϕ. From Lemma 7, we know that from (S, T), we can find an optimal first phase of an optimal schedule for *WSC* on G. We can also compute an optimal schedule for *WLA* on G' by using the algorithm described in Theorem 1, which has complexity $\mathcal{O}(n \log(n))$. Then, using Lemma 4, we know that we can merge the two obtained schedules to get an optimal schedule for *WSC* on G.

It then remains to find such a cut (S, T). This can be done in a brute-force way: each branch c_i can be cut at n_i different places, so there are $\prod_{i=0}^{k} n_i$ different cuts. Hence, the final complexity of the algorithm is $\mathcal{O}\Big(\prod_{i=1}^{k} n_i n \log(n)\Big)$. □

5. Heuristics and lower bounds

Building upon the previous results, we design heuristics to find efficient schedules for general DAGs for the *WLA* and *WSC* problems. The first set of heuristics are simple greedy heuristics that are used as baselines (Section 5.1). We then present in Section 5.2 a more involved heuristic for general DAGs, that builds upon the optimal algorithms for pumpkins from Section 4. We also present heuristics for k-chains on *WSC*, which can be used within the heuristic on general DAGs while the optimal algorithm has high complexity (Section 5.3). Finally, we introduce lower bounds for the *WSC* problem on k-chains in Section 5.4, with the aim of studying the performance of the heuristics in Section 6.

5.1. Greedy algorithms for general DAGs

The greedy algorithms create a schedule starting from the source node of the graph, iteratively choosing among the available nodes the one that minimizes a given cost function. We design three variants of the greedy algorithm, using these cost functions:

- Greedy-Mem: Choose the vertex that frees as much memory as possible;
- Greedy-Time: Choose the vertex that takes the shortest time to execute;
- Greedy-Ratio: Choose the vertex with the biggest ratio $R_G(a_i)$ (see Definition 10).

The pseudocode of the greedy algorithms is in Algorithm 1. These algorithms apply to any graph and constitute a comparison baseline for the more specialized algorithms. Because each iteration requires an insertion into a priority queue, their time complexity is $\mathcal{O}(n \log(n))$.

Algorithm 1: minSCGreedy(root, cmp)

Input : root of G, cmp comparison function for vertices
Output: Schedule obtained greedily

```
pq ← PriorityQueue(cmp);
pq.push(root);
s ← Schedule();
while pq is not empty do
    u ← pq.pop();
    if u is not in s then
        s.append(u);
        foreach v ∈ V⁺(u) do
            pq.push(v);
        end
    end
end
return s;
```

5.2. An involved heuristic for general DAGs

To compute a schedule for a general DAG, we proceed in two phases: we first convert the graph to an SP-graph (SPization), and then we recursively find a schedule for the SP-graph. For the first phase, we use the SPization method of Marchal [31], which employs an algorithm adapted from Tarjan et al. [32]. The graph is traversed from the source nodes to the bottom by maintaining a tree structure that describes the parallel composition of the graph. This tree structure helps us to detect SP-conflicts, that is, when a node has several predecessors coming from different

branches. In that case, a synchronization node is added to the graph to close all these branches, enforcing the series-parallel structure. Since this results in adding synchronization points, not all schedules of the original graphs are valid on the transformed SP-graph. Consequently, there is no guarantee that an optimal schedule for the transformed graph corresponds to an optimal schedule in the original graph. However, since the graph is modified only when needed and as little as possible, we expect that this process can help us to find schedules with low average memory for the original graph.

The second part of the process consists in recursively building a schedule. Given an SP graph G:

- If G consists of a single edge $a_s \to a_t$, then we return the schedule (s, t).
- If $G = G_1 \to G_2$ is a series composition of SP graphs, then let ϕ_i be the schedule returned by the heuristic on G_i for $i = \{1, 2\}$, and we return $\phi_1 + \phi_2$;
- If G is the parallel composition of G_1, G_2, let ϕ_i be the schedule produced by the heuristic on G_i for $i = \{1, 2\}$. We define the pumpkin graph G' with two branches, where each branch corresponds to the graph G_i linearized according to schedule ϕ_i, and with their sources merged into one node s. We then run the exact algorithm on the pumpkin G', and return the resulting schedule.

This two-phase process uses the optimality of the pumpkin algorithms to produce a good schedule on a transformed graph. Since the pumpkin algorithm is of polynomial complexity for *WLA* (see Theorem 1), we obtain a polynomial-time algorithm, denoted AvgSPAlgo.

However, the problem is more complex for *WSC*, and it builds on an exponential algorithm for k-chains (see Theorem 2, complexity in $\mathcal{O}\big(n^{k+1}\log(n)\big)$). Hence, we present below polynomial-time heuristics for *WSC* on k-chains, that will be more usable in practice.

5.3. Heuristics for WSC on k-chains

One straightforward approach is to apply the greedy algorithms introduced earlier in Section 5.1. However, these algorithms have a limited perspective on the graph, which may lead to suboptimal performance. To address this limitation, we propose additional algorithms based on graph cuts. As mentioned in Section 4.3, once the relevant cut is identified, an optimal schedule can be determined in $\mathcal{O}(n\log(n))$. Building on this, we develop a new class of algorithms that focus on identifying a *good* cut. The simplest approach involves generating random cuts, computing the corresponding schedules, and selecting the one that minimizes the sum cut cost. The pseudocode for this Random-Cut algorithm is in Algorithm 2.

Algorithm 2: minSCRandomCut(G, N)

Input : Graph G, of root $root$. N number of cuts drawn
Output: Best schedule found

```
bestSchedule ← None
for i = 1 to N do
    cut ← RandomCut();
    s ← ConstructSchedule(G, cut);
    if SCCost(s, G) < SCCost(bestSchedule, G) then
        bestSchedule = s;
    end
end
return bestSchedule;
```

The problem remains that, in order for our algorithm to be efficient, we have to draw a polynomial amount of cuts, among a set of exponential size. It turns out that in practice, this is pretty inefficient. Hence, we refine this heuristic through some guided exploration with a local search.

The idea of Local-Search is the following (see Algorithm 3): starting from a random cut of the graph, we try to do some small modifications and see if it improves the solution. If it does, then we take this new solution and continue the exploration, otherwise we discard it and start again from the previous solution. In our case, the modification consists in adding or removing one node on one of the chains (see function *ModifyCut*).

Algorithm 3: minSCLocalSearch(G, N)

```
Function LocalSearch(G):
    bestCut ← RandomCut();
    bestSched ← ConstructSchedule(G, bestCut);
    for i = 1 to N do
        newCut ← ModifyCut(bestCut);
        newSched ← ConstructSchedule(G, newCut);
        if Cost(newSched, G) < Cost(bestSched, G) then
            bestSched ← newSched;
            bestCut ← newCut;
    return bestSched;
Function ModifyCut(cut):
    (c_1, ..., c_l) ← RandomBranch(G);
    if RandomBool() then cut.add(c_{j+1}) ;
    else cut.remove(c_j) ;
    return cut;
```

5.4. Lower bounds for the WSC problem on k-chains

Because of the high complexity of the optimal algorithm for *WSC*, we cannot use it for large instances, and hence evaluate the performance of the heuristics. Therefore, we design lower bounds that will allow us to assess how close the heuristics might be to the optimal. The first lower bound is a reduction to the Linear Arrangement problem, since *WLA* can be solved optimally in polynomial time, while the second one uses linear programming.

5.4.1. Lower bound using WLA

The first idea is simply to use the exact algorithm for *WLA* by transforming a *single data* graph instance to a *multiple data* graph instance.

Let $G = (a_s, c_1, \ldots, c_k)$ be a k-chain of root a_s and of chains $c_1, \ldots, c_k$. We want to construct a graph G' such that:

- For any schedule ϕ, ϕ valid for $G \Leftrightarrow \phi$ valid for G'.
- For any schedule ϕ, $WSC_G(\phi) \geqslant WLA_{G'}(\phi)$.

This way, we can solve optimally *WLA* on G', and use this cost as a lower bound for *WSC* on G.

In order to construct G', we just need to define weights on each outgoing edge of a_s. Let w_s the weight of the outgoing edges from a_s in the initial graph G. The following lemma guides us in defining weights for G':

Lemma 10. *Consider G' a copy of G, where the weight of the edge from a_s to c_i is w_i, for $1 \leqslant i \leqslant k$, and $\sum_{i=1}^{k} w_i = w_s$. Then, for any schedule ϕ, $WSC_G(\phi) \geqslant WLA_{G'}(\phi)$.*

Proof sketch. Only the cost of the very first edges changes, and we find the desired result with a simple computation that is described in Appendix A.9. □

Thus, we can transform any graph G in the single data model, by splitting the single data w_s into multiple data w_i's to create a graph in the multiple data model, while ensuring that $\sum_{i=1}^{k} w_i = w_s$. Any distribution of the w_i's could be considered, and we use $w_i = w_s/k$. We can then solve the problem for *WLA*, which gives us a lower bound for *WSC*.

5.4.2. Lower bound using Integer Linear Programming (ILP)

Another idea to get a lower bound on the cost of the optimal schedule for *WSC* is to formulate the problem as a set of linear constraints with integer variables, hence we can use an ILP solver to obtain the optimal solution or a bound on the solution. Note that although we present here the set of constraints for a k-chain graph, this method can be extended to general graphs without issues.

Variables:
- $\forall i \in [1, n]$, $\textsc{start}_i \in \mathbb{R}^+$: start of task i.
- $\forall (i, j) \in [1, n]^2$, $\textsc{pred}_{i,j} \in \{0, 1\}$: whether i is taken before j in the schedule.
- $\textsc{last} \in \mathbb{R}^+$: start of the last neighbor of a_s.

Constraints:
- Given any two tasks, one has to be scheduled first: $\forall i, j, \textsc{pred}_{i,j} + \textsc{pred}_{j,i} = 1$.
- The precedence constraints are respected: $\forall (i, j) \in E, \textsc{start}_i \leqslant \textsc{start}_j$.
- If i is scheduled before j, it must finish before j starts: $\forall i, j, \textsc{start}_i + t_i \leqslant \textsc{start}_j + \sum_k w_k \textsc{pred}_{j,i}$.
- Last neighbor of a_s: $\forall i \in V^+(a_s), \textsc{start}_i \leqslant \textsc{last}$.

Objective function:

$$\min \sum_{i \in V \setminus \{a_s\}} \sum_{(j,k) \in E, j \neq s} \textsc{pred}_{j,i} \textsc{pred}_{i,k} w_{j,k} + w_s \textsc{last}.$$

In this formulation, there is a product of variables in the objective. But since these variables are binary, we can linearize them. To linearize $a \times b$, we replace it with a new binary variable c and add three constraints: $c \geqslant a + b - 1$, $c \leqslant a$, and $c \leqslant b$. These constraints indeed guarantee that $c = a \times b$.

From this set of constraints, we can use an ILP solver to search for the optimal solution. However, the main issue is that the number of constraints grows quite fast, so it can sometimes be hard for the solver to even find one valid solution. One way to solve this issue is to feed our best known solution as initial solution to the solver. In the experiments, we use the best solution from Local-Search as a starting point and use a time budget of 180 seconds. Either the optimal solution is returned before reaching the time limit, or the solver returns both a heuristic solution and a lower bound from constructing the dual problem. We denote by ILP the heuristic solution, and by ILP lower bound the lower bound returned by the solver.

6. Experimental results

In this section, we pursue three objectives: (i) we compare the performance of the exact and heuristic algorithms presented in the previous section for both *WSC* and *WLA* models on various type of graphs (k-chains, series-parallel graphs and general DAGs); (ii) we compare the performance of our best heuristic for minimizing the average memory to the optimal algorithm minimizing the peak; and finally (iii) we study whether it is more interesting to focus on average memory or peak memory minimization when scheduling several task graphs in parallel on a shared-memory system for real-life task graphs. The code is available online and all our experiments are easily reproducible [33].

6.1. Experimental setup

All the algorithms have been implemented in C++. We use the Boost Graph Library for fast and flexible implementation of graph structures, as well as Gurobi, a fast and parallel ILP solver. For the sake of comparison with peak memory minimization, we use the optimal algorithm proposed by Kayaaslan et al. [4] for SP-graphs.

We use various tools to generate the task graphs on which we execute the heuristics. For k-chains, we randomly generate them, first by generating a root node and k starts of chains (by default, $k = 5$), then by adding the remaining nodes to randomly chosen chains (by default $n = 256$ nodes, hence 250 remaining nodes), and finally by choosing the weights uniformly in a set interval (by default, $[1, 10]$). We also use graphs generated by the TaGaDa library [34], which generates DAGs with different degrees of parallelism. We use a parallelism ratio of 30% and a size of $n = 256$

Parameter name	Base value	Range
k parallelism	5	$[1, 10]$
n size	256	$[64, 1024]$
max edge weight	10	$[1, 10000]$
max node weight	10	$[1, 10000]$
parallelism ratio (Tagada)	30	$[0, 100]$

Table 2: Parameters for graph generation, with their base value as well as testing range.

nodes unless stated otherwise. This same library also generates SP graphs using the `-fj` (fork join) option, which creates SP graphs. Table 2 recapitulates the different parameters that have been studied, as well as their base value and variation range. Whenever a parameter is not specified in a figure, its value is the default one (found in the *base value* column). A more comprehensive analysis of these parameters has been carried out in Appendix B, where each graph generation parameter is varied independently, according to the ranges specified in Table 2. Our findings in this section are consistent across all other graph configurations, indicating that the results generalize broadly.

Finally, we use a dataset of real task graphs that originate from QR decomposition tasks [35]. This dataset is composed of 29 different tree-shaped task graphs in the *WSC* model, with full information about the time and memory required per task. From these task graphs, we extract only 14 of them for which the schedule obtained from optimizing for average memory differs by more than 1% from the one optimizing for peak memory. The others being too constrained, we did not include them in the analysis.

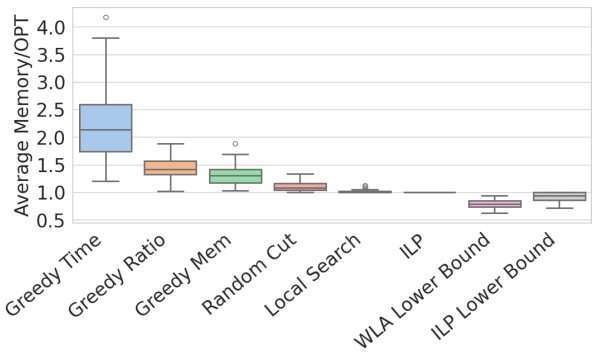
6.2. Heuristic comparison

Before evaluating the performance of the heuristics in a parallel setting, we start by comparing heuristics for the *WSC* problem on randomly generated k-chains. In Figure 10a, we plot the average memory usage normalized by Opt for the greedy algorithms, along with the Random-Cut, ILP and Local-Search heuristics, as well as the two lower bounds from Section 5.4. Since the graphs are small, we can compute Opt with the exponential algorithm as described in Section 4.3.

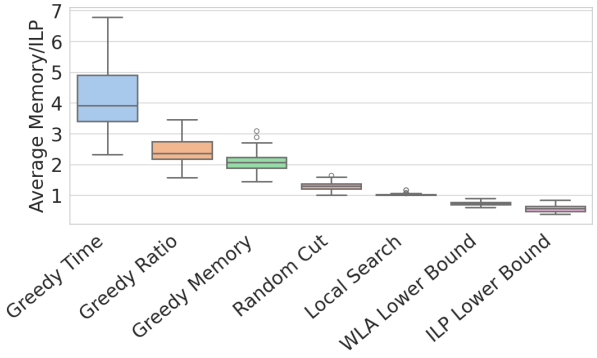
The boxplots display the median and interquartile range (IQR) of the data, with whiskers extending to the highest and lowest non-outlier points. Outliers are shown as individual points beyond the whiskers. Here, each boxplot corresponds to 50 different graphs.

As expected, the greedy algorithms perform poorly: Greedy-Mem consistently generates schedule with an average memory consumption 30% higher than Opt. This is due to the fact that they lack knowledge of the graph's structure, leading to short term memory gains but significant long term disadvantages. In contrast, heuristics with full graph awareness perform significantly better: Random-Cut is only 10% above Opt, and Local-Search almost always finds the optimal solution, failing by no more than 5%. Exploiting the cuts in the chains not only yields far better results than the greedy approaches, but it also proves to be a sufficiently powerful tool to obtain an optimal schedule in most cases. Additionally, we observe that although ILP can find the optimal solution for small graphs, its lower bound is not always tight, even with a 180-second time limit. The lower bound obtained by reducing the problem to *WLA* is also not tight, as the transformation to *WLA* loses too much critical information about the graph.

In larger graphs, calculating Opt becomes unfeasible due to its exponential complexity, and hence we compare the heuristics to the lower bounds. Figure 10b presents results similar to previously, but on larger k-chain graphs, and normalized by the best solution found by ILP. It is important to note that we still initialize the ILP with the best solution from Local-Search; however, it does not necessarily converge to the optimal solution. Therefore, we take its best result within a time limit of 180 seconds. Interestingly, the lower bound derived from the ILP is inferior to



(a) Heuristics comparison for *WSC* on small k-chains of size 32 that we can solve optimally.



(b) Heuristics comparison for *WSC* on large k-chains of size 256.

Figure 10: Heuristics comparison for WSC on k-chains

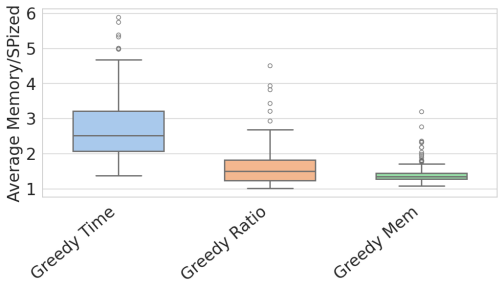


Figure 11: Comparison of heuristics for *WLA* on DAGs, normalized by the result of AvgSPAlgo.

that obtained by reducing the problem to *WLA*. Nevertheless, we observe a significant gap of approximately 25% between the Local-Search solution and the lower bound.

A more detailed study of the Local-Search's performance, compared to Opt and to the Peak optimal algorithm, has been conducted in Appendix B. We find that the performance of Local-Search stays pretty much unchanged for any k-chain, and that it also reaches a peak no more than 20% worse than PeakSPAlgo.

These results suggest that the lower bounds, rather than the heuristics, are the primary concern, as indicated by the observations on smaller task graphs. This reinforces our initial intuition: leveraging graph cuts leads to high-quality solutions that outperform greedy algorithms and approach the optimal solution for k-chains.

In a second step, we compare the average memory usage of greedy heuristics on general DAGs of size 256 under the *WLA* memory model. Computing the optimal average memory on these DAGs is out of reach, hence we compare the simple heuristics to AvgSPAlgo, which transforms the DAG into a SP graph in order to compute a good schedule. As shown in Figure 11, which reports memory usage normalized by AvgSPAlgo, the greedy approaches again perform poorly. Even, Greedy-Mem incurs an average memory overhead of 40% compared to AvgSPAlgo. This proves that a more involved heuristic such as AvgSPAlgo is needed to obtain good performance.

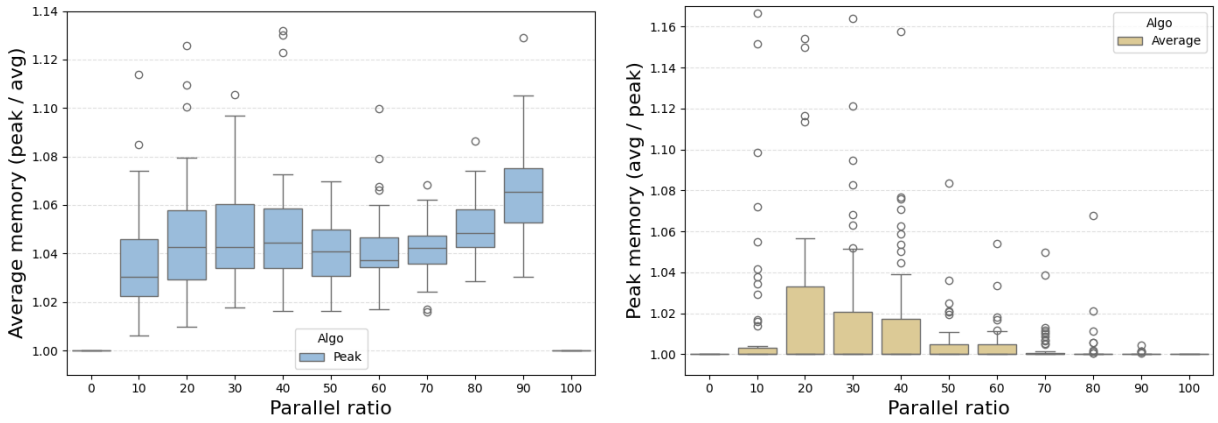
6.3. Comparison between peak and average-memory minimization on SP-graphs

After assessing that the sophisticated heuristic performs well for *WSC* and *WLA*, we check whether the peak memory achieved by this heuristic is close to the optimal. Since there exists an $\mathcal{O}(n \log(n))$ exact algorithm for the peak in the *multiple data* memory model on SP graphs [4], we perform experiments for *WLA* on SP graphs. We denote the optimal algorithm for peak memory as PeakSPAlgo, and we still refer to the heuristic for *WLA* on SP graphs as AvgSPAlgo.

Figure 12 shows two plots that compare PeakSPAlgo and AvgSPAlgo for both the average memory (*WLA*) and peak memory problem on SP graphs. In Figure 12a, we observe that the schedule aimed at minimizing peak memory consumption almost never achieves the optimal solution for average memory. For any non-trivial ratio of parallelism, AvgSPAlgo consistently outperforms PeakSPAlgo by 4% in terms of minimizing average memory. When the task graphs become even more parallel, it gets even better with a 6% gain for a ratio of 90. More importantly, as shown in Figure 12b, AvgSPAlgo performs well in minimizing peak memory. It nearly always finds the optimal solution, with only rare instances where the solution is worse than PeakSPAlgo. This indicates that, in addition to producing a schedule with better average memory, AvgSPAlgo still achieves a peak memory that remains relatively small. These observations are further developed in Appendix B, where each individual graph generation parameter is studied. The main results here stay the same, i.e., PeakSPAlgo is always outperformed by around 5% for the average metric, while AvgSPAlgo almost always finds the optimal peak schedule.

6.4. Parallel execution

Finally, we compare the schedules produced by AvgSPAlgo against the ones provided by PeakSPAlgo with task graphs executed in parallel. This corresponds to a setting where multiple jobs, each of them handling a task graph, are processed concurrently and share a common limited memory, and each application is scheduled independently from the others. The memory used is the sum of data used by each task graph at any given time. The task graphs used



(a) PeakSPAlgo on the average metric (memory normalized by AvgSPAlgo)

(b) AvgSPAlgo on the peak metric (memory normalized by PeakSPAlgo)

Figure 12: Comparison between PeakSPAlgo and AvgSPAlgo on SP graphs, in terms of average (*WLA*) and peak memory. The parallel ratio is a parameter from the TaGaDa graph generation library that goes from 0, a chain to 100, a pumpkin.

here originate from a QR factorization dataset [35]. They come with full dependency structure, flops per computation of each node and memory requirements. We always assume that the computational capability of the jobs is constant, hence the data about the amount of flops is transformed into a computational time per node. The setting we put ourselves in is that each job will schedule 16 of these task graphs in a row, then stop.

In Figure 13, we plot, for different numbers of jobs in parallel, the cumulated peak memory when all task graphs are scheduled with AvgSPAlgo, compared to that obtained when all task graphs are scheduled with PeakSPAlgo. As soon as we consider a parallel setting, the overall peak starts being smaller for AvgSPAlgo than for PeakSPAlgo. This happens in almost all of the cases for 16 task graphs, being on average 3.5% smaller. Our intuition is validated: minimizing the peak of each individual task graph (as done by PeakSPAlgo) does not give good results for parallel processing, since each task graph might reach its peak at a different time. Outside of these peaks, PeakSPAlgo has not incentive in decreasing the memory footprint, which explains why AvgSPAlgo outperforms PeakSPAlgo for the cumulative memory usage.

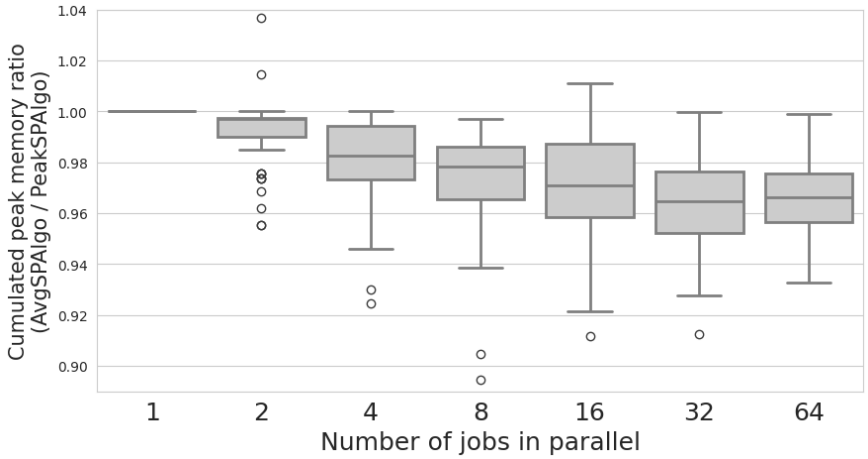


Figure 13: Comparison between PeakSPAlgo and AvgSPAlgo on parallel task graphs from QR factorization. Each job takes 16 different task graphs and schedules them in a row. The cumulative peak of all jobs is then measured, either while task graphs are optimized for the peak or for the average.

7. Conclusion

While most existing studies on designing memory-aware schedules for task graphs concentrate on peak-memory minimization, we believe that focusing on the average memory is important and constitutes a good proxy for the global peak-memory minimization of multiple independently scheduled task graphs. We have formally defined two optimization problems (*WSC* and *WLA*) for average memory minimization, corresponding to two memory models, and we have emphasized their connection with existing problems on graphs. We have designed new optimal algorithms for several classes of graphs (namely k-chains and pumpkins). Based on these algorithms, we have proposed heuristic strategies for series-parallel and general graphs.

Experiments on both synthetic and actual task graphs demonstrate that the proposed heuristics perform well in practice in a variety of settings, often finding solutions close to the optimal average memory. Additionally, our heuristics produce schedules with reasonably small peak memory for single task graphs. In the context of the parallel processing of task graphs, heuristics minimizing the average memory footprint of each individual application outperform those focusing on their peak memory, which validates the hypothesis that minimizing average memory can lead to lower overall peak memory usage in parallel executions. While the proposed strategies are experimentally validated, several limitations remain. We do not model the parallel execution of multiple task graphs or the complexity of memory hierarchies and resource interactions, and we rely on average memory usage without providing theoretical guarantees on the parallel peak. These aspects, along with the impact of non-deterministic data sizes and dynamic task dependencies, remain open directions for future investigation.

Acknowledgments: We thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

References

[1] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. 13th Annual ACM Symposium on the Theory of Computation (STOC)*, 1981, pp. 326–333.

[2] M. Drozdowski, "Scheduling parallel tasks – algorithms and complexity," in *Handbook of Scheduling*, J. Leung, Ed. Chapman and Hall/CRC, 2004.

[3] J. W. H. Liu, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, pp. 375–395, 1987.

[4] E. Kayaaslan, T. Lambert, L. Marchal, and B. Uçar, "Scheduling series-parallel task graphs to minimize peak memory," *Theoretical Computer Science*, vol. 707, pp. 1–23, Jan. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397517307053

[5] C. Jin, M. Purohit, Z. Svitkina, E. Vee, and J. R. Wang, "New Tools for Peak Memory Scheduling," Dec. 2023, arXiv:2312.13526 [cs]. [Online]. Available: http://arxiv.org/abs/2312.13526

[6] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory," *ACM Trans. Parallel Comput.*, vol. 2, no. 2, pp. 13:1–13:37, 2015. [Online]. Available: https://doi.org/10.1145/2779052

[7] M. Gonthier, L. Marchal, and S. Thibault, "A scheduler to foster data locality for GPU and out-of-core task-based linear algebra applications," *J. Parallel Distributed Comput.*, vol. 206, p. 105170, 2025. [Online]. Available: https://doi.org/10.1016/j.jpdc.2025.105170

[8] P. Fradet, A. Girault, and A. Honorat, "Parallel scheduling of task graphs with minimal memory requirements," in *IPDPS 2025 - 39th IEEE International Parallel and Distributed Processing Symposium*, Milano, Italy, Jun. 2025, pp. 1–12. [Online]. Available: https://inria.hal.science/hal-04879748

[9] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 16:1–16:16. [Online]. Available: https://doi.org/10.1145/3342195.3387524

[10] T. Kang and H. Yu, "Elastic vertical memory management for container-based stateful applications in kubernetes," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025*, J. Hong, S. Battiato, C. Esposito, J. W. Park, and A. Przybylek, Eds. ACM, 2025, pp. 201–208. [Online]. Available: https://doi.org/10.1145/3672608.3707723

[11] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, and G. Oren, "CXL Memory as Persistent Memory for Disaggregated HPC: A Practical Approach," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023, p. 983–994.

[12] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger, "Memory disaggregation: why now and what are the challenges," *SIGOPS Oper. Syst. Rev.*, vol. 57, no. 1, p. 38–46, Jun. 2023.

[13] F. Zacarias, P. Carpenter, and V. Petrucci, "Dynamic Memory Provisioning on Disaggregated HPC Systems," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, 2023, p. 973–982.

[14] R. Boëzennec, D. Carastan-Santos, F. Dufossé, and G. Pallez, "Allocation Strategies for Disaggregated Memory in HPC Systems," in *IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2024.

[15] K. Agrawal, M. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato, "Online Parallel Paging with Optimal Makespan," *ACM|Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, Jul. 2022, accepted: 2022-11-14T14:21:19Z ISBN: 9781450391467 Publisher: ACM|Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures. [Online]. Available: https://dspace.mit.edu/handle/1721.1/146373

[16] L. Marchal, B. Simon, and F. Vivien, "Limiting the memory footprint when dynamically scheduling DAGs on shared-memory platforms," *Journal of Parallel and Distributed Computing*, vol. 128, p. 30, 2019. [Online]. Available: https://inria.hal.science/hal-02025521

[17] "Technical Report CS0043," Feb. 2020. [Online]. Available: https://web.archive.org/web/20200222173146/https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/1975/CS/CS0043

[18] D. Adolphson and T. C. Hu, "Optimal Linear Ordering," *SIAM J. Appl. Math.*, vol. 25, no. 3, pp. 403–423, Nov. 1973, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://epubs.siam.org/doi/10.1137/0125042

[19] S. Rao and A. W. Richa, "New approximation techniques for some ordering problems," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '98. USA: Society for Industrial and Applied Mathematics, Jan. 1998, pp. 211–218.

[20] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

[21] ——, "On the storage requirement in the out-of-core multifrontal method for sparse factorization," *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 249–264, 1986.

[22] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensive workflows onto storage-constrained distributed resources," in *CCGRID'07*. IEEE, 2007.

[23] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," *Computer Languages, Systems & Structures*, vol. 37, no. 2, pp. 63–75, 2011.

[24] R. Sethi and J. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, vol. 17, no. 4, pp. 715–728, 1970.

[25] R. Sethi, "Complete register allocation problems," in *STOC'73*. ACM Press, 1973, pp. 182–195.

[26] S. Even, "NP-completeness of several arrangement problems," *Technical Report ; Department of computer Science*, vol. 43, 1975. [Online]. Available: https://cir.nii.ac.jp/crid/1572543024473274624

[27] T. Bossart, A. M. Kordon, and F. Sourd, "Memory management optimization problems for integrated circuit simulators," *Discrete Applied Mathematics*, vol. 155, no. 14, pp. 1795–1811, Sep. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166218X07000662

[28] S. Achouri, T. Bossart, and A. Munier-Kordon, "A polynomial algorithm for minDSC on a subclass of series Parallel graphs," *RAIRO - Operations Research*, vol. 43, no. 2, pp. 145–156, Apr. 2009, publisher: EDP Sciences.

[29] A. Quilliot and D. Rebaine, "Linear time algorithms to solve the linear ordering problem for oriented tree based graphs," *RAIRO - Operations Research - Recherche Opérationnelle*, vol. 50, no. 2, pp. 315–325, 2016. [Online]. Available: http://www.numdam.org/articles/10.1051/ro/2015024/

[30] J. Díaz, J. Petit, and M. Serna, "A survey of graph layout problems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 313–356, Sep. 2002. [Online]. Available: https://dl.acm.org/doi/10.1145/568522.568523

[31] L. Marchal, "memDAG," https://gitlab.inria.fr/lmarchal/memdag.

[32] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 298–313, 1982. [Online]. Available: https://doi.org/10.1137/0211023

[33] "Gitlab code," Feb. 2026. [Online]. Available: https://gitlab.inria.fr/lmarchal/memdag/-/tree/average-mem

[34] Pascal Raymond, "tagada – Task dAg GenerAtor Doubtful Acronym," 2024. [Online]. Available: https://ttk.gricad-gitlab.univ-grenoble-alpes.fr/prdistrib/tagada

[35] L. Marchal, B. Simon, O. Sinnen, and F. Vivien, "Malleable task-graph scheduling with a practical speed-up model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1357–1370, Jun. 2018. [Online]. Available: https://inria.hal.science/hal-01687189

Appendix A. Additional proofs

Appendix A.1. Naive recursion is arbitrarily bad for WSC on r-2TSPG

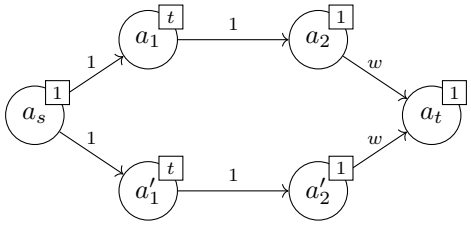


Figure A.14: r-2TSPG graph.

Lemma 11. *Treating recursively each subgraph of an r-2TSPG graph can produce arbitrarily bad outputs for the WSC problem.*

Proof. We consider schedules for the graph of Figure A.14. On the one hand, $L_\phi = (a_s, a_1, a_2, a'_1, a'_2, a_t)$ is an optimal schedule for *WSC* if we treat both branches separately. The associated cost is $C(\phi) = 2 + t \times 2 + w + 1 + t(w+1) + 2 \times w = tw + 3t + 3w + 3$. On the other hand, if we consider $L_{\phi'} = (a_s, a_1, a'_1, a_2, a'_2, a_t)$, a schedule that intertwines both subgraphs, its associated cost is:
$C(\phi) = 2 + w \times 2 + t \times 2 + w + 1 + 2w = 3w + 4t + 3$.

Not only is ϕ not optimal, but we have $C(\phi) = \mathcal{O}(tw)$ while $C(\phi') = \mathcal{O}(\max(t, w))$. □

In particular, this means that the $\mathcal{O}(n^2)$ algorithm given in [28] for the SC version of r-2TSPG is not optimal anymore in the *WSC* problem.

Appendix A.2. Naive recursion is arbitrarily bad for WLA on chains

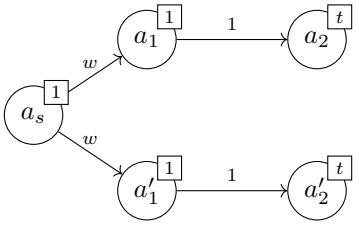


Figure A.15: 2-chain.

Lemma 12. *Treating each chain independently on a k-chain graph can produce arbitrarily bad outputs for the WLA problem.*

Proof. Consider two schedules for the graph in Figure A.15: $L_\phi = (a_s, a_1, a_2, a'_1, a'_2)$, the optimal schedule if we treat both chains separately, and $L_{\phi'} = (a_s, a_1, a'_1, a_2, a'_2)$ the optimal schedule.

We have $C(\phi) = \mathcal{O}(tw)$, while $C(\phi') = \mathcal{O}(\min(t, w))$. □

In particular, this means that the polynomial algorithm for optimal LA on out-trees [29] does not work out of the box for the weighted version.

Appendix A.3. Proof of Lemma 1

Proof. We first prove that ϕ' is a schedule for G'. Recall that $t_{\max} = \sum_{i=1}^{|V|} t_i$.

Overlap: Suppose that we have an overlap, i.e., we have two nodes a_i, a_j with:

$$\begin{aligned} \phi'(a_j) &< \phi'(a_i) + t_i \\ t_{\max} - (\phi(a_j) + t_j) &< t_{\max} - (\phi(a_i) + t_i) + t_i \\ \phi(a_i) &< \phi(a_j) + t_j \end{aligned}$$

and

$$\begin{aligned} \phi'(a_i) &< \phi'(a_j) + t_j \\ t_{\max} - (\phi(a_i) + t_i) &< t_{\max} - (\phi(a_j) + t_j) + t_j \\ \phi(a_j) &< \phi(a_i) + t_i \end{aligned}$$

This contradicts that ϕ is a schedule for G as it has an overlap.

Finished: Suppose that there is a node a_i that is not finished. We get:

$$\begin{aligned} \phi'(a_i) + t_i &> t_{\max} \\ t_{\max} - (\phi(a_i) + t_i) + t_i &> t_{\max} \\ \phi(a_i) &< 0 \end{aligned}$$

This once again contradicts the fact that ϕ is a valid schedule for G.

Dependencies: Suppose the dependencies of G' are not respected. We have $(a_i, a_j) \in E'$ such that:

$$\begin{aligned} \phi'(a_i) &\geqslant \phi'(a_j) \\ t_{\max} - (\phi(a_i) + t_i) &\geqslant t_{\max} - (\phi(a_j) + t_j) \\ \phi(a_j) + t_j &\geqslant \phi(a_i) + t_i \\ \phi(a_j) + t_j &> \phi(a_i) \end{aligned}$$

By construction of E', we know that $(a_j, a_i) \in E$, and because ϕ is a valid schedule for G, we get $\phi(a_j) < \phi(a_i)$, and even $\phi(a_j) + t_j \leqslant \phi(a_i)$ (no overlap). This gives a direct contradiction.

Therefore, ϕ' is a schedule for G'.

Next, we prove that $\mathit{WLA}_G(\phi) = \mathit{WLA}_{G'}(\phi') + C$.

$$\begin{aligned} \mathit{WLA}_{G'}(\phi') &= \sum_{(a_i,a_j)\in E'} w'_{a_i,a_j}(\phi'(a_j) - \phi'(a_i)) \\ &= \sum_{(a_i,a_j)\in E'} w'_{a_i,a_j}(\phi(a_i) + t_i - \phi(a_j) - t_j) \\ &= \sum_{(a_j,a_i)\in E} w_{a_j,a_i}(\phi(a_i) - \phi(a_j)) + \sum_{(a_j,a_i)\in E} w_{a_j,a_i}(t_i - t_j) \\ &= \mathit{WLA}_G(\phi) + C \end{aligned}$$

This means that when ϕ is an optimal schedule for G, ϕ' is also an optimal schedule for G'. Indeed, suppose that there is a better schedule ψ'. We can consider its reverse schedule ψ for G.

We get $\mathit{WLA}_G(\psi) = \mathit{WLA}_{G'}(\psi') + C < \mathit{WLA}_{G'}(\phi') + c = \mathit{WLA}_G(\phi)$, which contradicts that ϕ is an optimal schedule for G. □

Appendix A.4. Proof of Lemma 3

Proof. We recall that G is a pumpkin graph of sink a_t, with $c = (a_1, \dots, a_i)$ a chain of G with non-zero weights, and that G' is the graph where all the weights along c are decreased by 1. The main result is that for any schedule ϕ, $\mathit{WLA}_G(\phi) = \mathit{WLA}_{G'}(\phi) + \sum_{a_j \in V \setminus \{a_t\}} t_j$. This can be easily seen if we look at the memory used by the chain c: wherever in the schedule up to the very end, exactly one edge of c is in the memory. If we diminish by 1 the value of the whole chain, then the total cost decreases by the length of c, which is always $\sum_{a_j \in V \setminus \{a_t\}} t_j$.

Let ϕ be an optimal schedule for *WLA* on G'. Because the dependencies of G' are the same as for G, ϕ is also a schedule for G. Suppose that there exists ψ a schedule for G such that $\mathit{WLA}_G(\psi) < \mathit{WLA}_G(\phi)$.

We then have:

$$\mathit{WLA}_{G'}(\psi) = \mathit{WLA}_G(\psi) - \sum_{a_j \in V \setminus \{a_t\}} t_j < \mathit{WLA}_G(\phi) - \sum_{a_j \in V \setminus \{a_t\}} t_j = \mathit{WLA}_{G'}(\phi).$$

This contradicts the optimality of ϕ for *WLA* on G'. □

Appendix A.5. Proof of Lemma 4

Proof. Let ϕ be an optimal schedule for *WLA* on G, with $L_\phi = (a_1, \cdots, a_n)$. We can then define $\psi = \phi_{[S]} + \phi_{[T]}$. We will prove that ψ is a valid and optimal schedule for *WLA* on G.

Let us first prove that ψ is valid. Consider an edge $(a_i, a_j) \in E$. We want to prove that $\psi(a_i) < \psi(a_j)$. There are three cases:

- $a_i, a_j \in S$. Then $\phi_{[S]}(a_i) < \phi_{[S]}(a_j) \Rightarrow \psi(a_i) < \psi(a_j)$.
- $a_i, a_j \in T$. Then $\phi_{[T]}(a_i) < \phi_{[T]}(a_j) \Rightarrow \psi(a_i) < \psi(a_j)$.
- $a_i \in S, a_j \in T$. By construction, $\psi(a_i) < \psi(a_j)$.

We cannot have $a_i \in T, a_j \in S$ because (S, T) is a valid cut.

We now prove that $\mathit{WLA}_G(\psi) \leqslant \mathit{WLA}_G(\phi)$. For this, we consider the cost associated with a vertex $u \in V$ and a chain $c = (a_1, \dots, a_l)$ with $\phi(a_j) < \phi(u) < \phi(a_{j+1})$. The contribution for G is $t_u w_{j,j+1}$. Suppose that $u \in S$, once again, there are three cases:

- $a_j, a_{j+1} \in S$, we still have $\psi(a_j) < \psi(u) < \psi(a_{j+1})$, the contribution is still $t_u w_{j,j+1}$.
- $a_j \in S, a_{j+1} \in T$, then we still have $\psi(a_j) < \psi(u)$ because both are in S, and $\psi(u) < \psi(a_{j+1})$ because $u \in S, a_{j+1} \in T$. The contribution is still $t_u w_{j,j+1} = 0$.
- $a_j, a_{j+1} \in T$. Then, let k be the index such that $a_k \in S, a_{k+1} \in T$, with $a_k, a_{k+1} \in c$. We now have $\psi(a_k) < \psi(a_j) < \psi(u)$ and $\psi(u) < \psi(a_{k+1})$. Thus, the contribution is $t_u w_{k,k+1} = 0$.

In all three cases, the contribution of c for u is smaller or equal for ψ than for ϕ. If $u \in T$, we can do the same reasoning but in symmetry. Overall, we get $\mathit{WLA}_G(\psi) \leqslant \mathit{WLA}_G(\phi)$. □

Appendix A.6. Proof of Lemma 5

We first define different edge types for a chain, and give a formal definition of the notion of graph compression.

Definition 13 (Rigid and elastic edge)**.** *Let G be a DAG. An edge $(a_i, a_j) \in E$ is called rigid if, for any optimal schedule ϕ, we have $\phi(a_i) + t_i = \phi(a_j)$, i.e., a_j is scheduled immediately after a_i in any optimal schedule. If (a_i, a_j) is not rigid, we call it an elastic edge.*

Definition 14 (Merged nodes)**.** *Let $G = (V, E)$, $(a_i, a_j) \in E$. We define the merge operation on nodes a_i and a_j by:*

- *Add a node $a_{i,j}$ of weight $t_i + t_j$*

(a) A weighted graph G

(b) $G_{(a_{2,3})}$ with a_2 and a_3 merged

Figure A.16: Merge operation.

- *Replace each edge* (a_j, a_k) *by an edge* $(a_{i,j}, a_k)$
- *Replace each edge* (a_k, a_i) *by an edge* $(a_k, a_{i,j})$
- *Remove the nodes* a_i *and* a_j

The resulting graph is denoted as $G_{(a_{i,j})}$*. We also define the merged schedule* $\phi_{(a_{ij})}$*. An example is given in Figure A.16.*

Lemma 13. *Let* G *be a chain, and* $(a_i, a_j) \in E$ *be a rigid edge.* ϕ *is an optimal schedule for WSC on* G *if and only if* $\phi_{(a_{i,j})}$ *is an optimal schedule for WSC on* $G_{(a_{i,j})}$*.*

Proof. Let us say that the successor of a_j is a_k. The main result of the proof is that:

$$WSC_G(\phi) = WSC_{G_{(a_{i,j})}}(\phi_{(a_{i,j})}) + t_i(w_{i,j} - w_{j,k})$$

$$\begin{aligned}
WSC_G(\phi) &= \sum_{u \in V} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{u \in V \setminus \{a_i, a_j\} \cup \{a_{ij}\}} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) + w_{i,j}(\phi(a_j) - \phi(a_i)) \\
&\quad + w_{j,k}(\phi(a_k) - \phi(a_j)) - w_{j,k}(\phi_{(a_{i,j})}(a_k) - \phi_{(a_{i,j})}(a_{ij}))
\end{aligned}$$

We now use the fact that (a_i, a_j) is rigid:

$$\begin{aligned}
WSC_G(\phi) &= WSC_{G_{(a_{i,j})}}(\phi_{(a_{i,j})}) + w_{i,j}t_i + w_{j,k}t_j - w_{j,k}(t_i + t_j) \\
&= WSC_{G_{(a_{i,j})}}(\phi_{(a_{i,j})}) + w_{i,j}(t_i - t_j)
\end{aligned}$$

Because (a_i, a_j) is rigid, we can always transform a schedule ϕ for G into $\phi_{(a_{i,j})}$ for $G_{(a_{i,j})}$. Because both costs are equal up to a constant, the optimality of ϕ means the optimality of $\phi_{(a_{i,j})}$ and vice versa. □

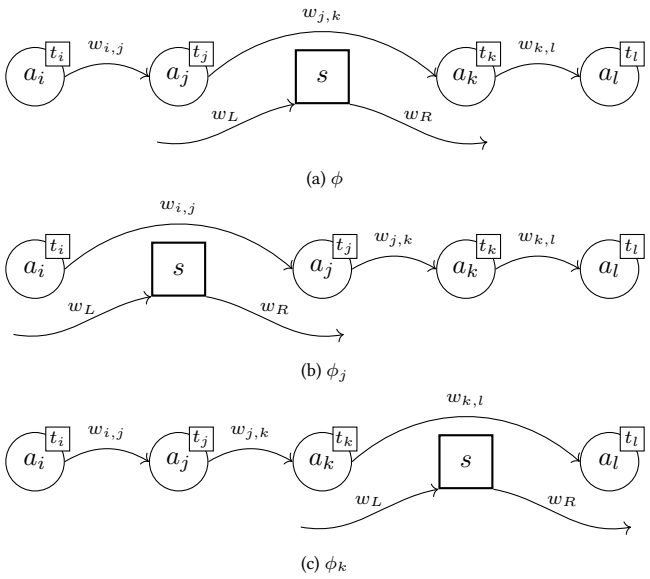
Lemma 14. *Let* G *be a* k*-chain tree. Let* $a_i, a_j, a_k, a_l \in V$ *be a part of one chain of* G*. If* (a_j, a_k) *is an elastic edge, then* $R(a_j) \geqslant R(a_k)$*.*

Proof. Because (a_j, a_k) is elastic, there exists an optimal ordering ϕ where a_k does not follow a_j immediately. We define the following quantities:

- S the set of nodes ordered between a_j and a_k in ϕ. We have $a \in S \Leftrightarrow \phi(v) < \phi(a) < \phi(w)$.
- $E_L = \{(u, v) \in E | \phi(u) < \phi(a_j), v \in S, \phi(v) = \max_{v' \in V^+(u)} \phi(v')\}$ the edges from other branches that are in the cut when a_j is scheduled and finish in S.
- $E_R = \{(u, v) \in E | \phi(a_k) < \phi(v), u \in S, \phi(v) = \max_{v' \in V^+(u)} \phi(v')\}$ the edges from other branches that are in the cut when a_k is scheduled and start in S.
- $w_L = \sum_{(u,v) \in E_L} w_{u,v}$.
- $w_R = \sum_{(u,v) \in E_R} w_{u,v}$.

We consider two new schedules ϕ_j and ϕ_k that derive from ϕ. If $L_\phi = (\dots, a_j, S, a_k, \dots)$, then $L_{\phi_j} = (\dots, S, a_j, a_k, \dots)$ and $L_{\phi_k} = (\dots, a_j, a_k, S, \dots)$. Both are still valid schedules as a_j and a_k belong to another branch as the nodes of S.

Instead of computing the whole cost of each schedule, we can compute the difference between two schedules by looking at the variation in lengths of the different elastic edges involved. Figure A.17 shows how these quantities evolve between each schedule. ϕ is supposed optimal, so we get:



(a) ϕ

(b) ϕ_j

(c) ϕ_k

Figure A.17: Local swaps.

$$\begin{aligned} \mathit{WSC}_G(\phi_j) &\geqslant \mathit{WSC}_G(\phi) \\ w_R t_j + w_{i,j} \sum_{a \in S} t_a &\geqslant w_L t_j + w_{j,k} \sum_{a \in S} t_a \\ R(a_j) = \frac{w_{i,j} - w_{j,k}}{t_j} &\geqslant \frac{w_L - w_R}{\sum_{a \in S} t_a} \end{aligned}$$

We also have:

$$\begin{aligned} \mathit{WSC}_G(\phi_k) &\geqslant \mathit{WSC}_G(\phi) \\ w_L t_k + w_{k,l} \sum_{a \in S} t_a &\geqslant w_R t_k + w_{j,k} \sum_{a \in S} t_a \\ \frac{w_L - w_R}{\sum_{a \in S} t_a} &\geqslant \frac{w_{j,k} - w_{k,l}}{t_k} = R(a_k) \end{aligned}$$

We can then combine both inequations to get $R(a_j) \geqslant R(a_k)$, concluding the proof. □

This first lemma gives a first way of compressing the graph: we know that for an edge to be elastic, the ratios along each chain have to be decreasing. If they are not, we can merge the nodes. There is a second criterion to merge more nodes together:

Definition 15 (Temp-min edge)**.** *Let T be a k-chain tree. We consider one of its chain $C = (a_1, \ldots, a_m)$. The edge (a_i, a_{i+1}) is a temporary minimal edge (abbreviated temp-min), if and only if $w_{i,i+1} = \min_{j \leqslant i} w_{j,j+1}$.*

In other words, it is a temporary minimal edge if its weight is the smallest on the chain yet. In particular, edges outgoing from the root are temp-mins.

Lemma 15. *Let T be a k-chain tree. If (a_i, a_j) is not a temp-min edge, then it is a rigid edge.*

This lemma states that the only edges that remain in the compressed graph are the temp-min ones, thus the weight of the edges along each chain will always be strictly decreasing. Once again, the proof follows the one from Adolphson and Hu [18].

Proof. Let (u, v) be a non temp-min edge in a chain C. Let $u_1, \ldots, u_n$ be the nodes of compressed chain C, i.e., with (u_i, u_{i+1}) all elastic edges. We suppose that (u, v) is elastic, so there is an i such that $(u_i, u_{i+1}) = (u, v)$. We choose (u, v) to be the first non temp-min edge that is elastic, so $(u_{i-1}, u_{i,i+1})$ is a temp-min. In particular, $w_{i-1,i} < w_{i,i+1}$.

We will add one edge of weight 0 in the end, together with a node u_{n+1}. This does not change the schedule nor its cost as we can always schedule u_{n+1} to be in last.

We have $R_G(u_i) = \frac{w_{i-1,i} - w_{i,i+1}}{w_i} < 0$. We know that the ratios are decreasing along the chain. We get $R_G(u_n) \leqslant R_G(u_{n-1}) \leqslant \ldots \leqslant R_G(u_i) < 0$. Thus $\frac{w_{n-1,n} - w_{n,n+1}}{w_n} < 0$, and finally $w_{n-1,n} < 0$ because we set $w_{n,n+1}$ to 0. This contradicts that all the weights of the graph are positive.

We get a contradiction, so (u, v) has to be a rigid edge. □

Appendix A.7. Proof of Lemma 7

Proof. Let us consider an optimal schedule ϕ for *WSC*, with the chain $c = (a_1, \ldots, a_i)$ ordered just before the chain $c' = (a'_1, \ldots, a'_j)$ in the first phase of the schedule, with

$$\frac{t_i}{\sum_{l=1}^{k_i} w_{i,l}} > \frac{t_j}{\sum_{l=1}^{k_j} w_{j,l}}.$$

We know from the previous lemma that both chains are scheduled continuously.

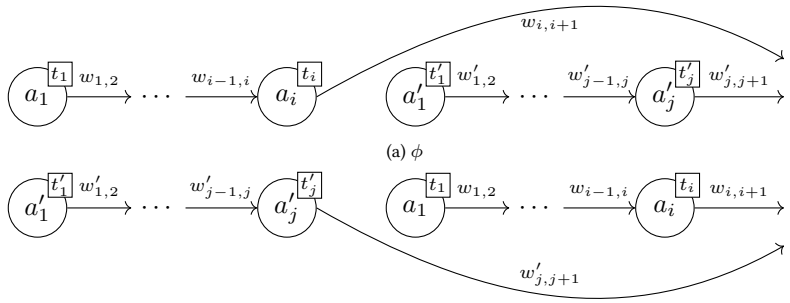
We define ϕ' as the copy of ϕ, but with the chains c and c' swapped. ϕ and ϕ' are represented in Figure A.18. We can see that the only edges modified are $w_{i,i+1}$ and $w'_{j,j+1}$. We get:

$$C(\phi) \leqslant C(\phi')$$
$$t_i \sum_{l=1}^{k_j} w_{j,l} \leqslant t_j \sum_{l=1}^{k_i} w_{i,l}$$

This is a direct contradiction with the initial assumption, so all the chains need to be ordered by decreasing values of $\frac{t_i}{\sum_{j=1}^{k_i} w_{i,j}}$. □

Appendix A.8. Proof of Lemma 9

Proof. We prove that, for a given schedule ϕ for G, $WSC_G(\phi_{[S]} + \phi_{[T]}) = C_{\phi_{[S]}} + WLA_{G'}(\phi_{[T]})$, where $C_{\phi_{[S]}}$ is a constant that only depends on $\phi_{[S]}$.



(a) ϕ

(b) ϕ'

Figure A.18: Exchanging two chains in the first phase of a schedule.

$$
\begin{aligned}
WSC_G(\phi_{[S]} + \phi_{[T]}) &= \sum_{u \in V} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{u \in S} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&\quad + \sum_{u \in T} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{u \in S} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&\quad + \sum_{\substack{(u,v) \in E \\ u \in T}} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{u \in S} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&\quad + \sum_{\substack{(u,v) \in E' \\ u \neq s}} w_{u,v}(\phi_{[T]}(v) - \phi_{[T]}(u))
\end{aligned}
$$

We can separate the first term whether u is next to the cut or not:

$$
\begin{aligned}
\textsc{First} &= \sum_{u \in S} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{\substack{u \in S \\ V^+(u) \cap T = \emptyset}} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) \\
&\quad + \sum_{\substack{(u,v) \in E \\ u \in S, v \in T}} w_{u,v}(\phi(v) - \phi(u)) \\
&= \sum_{\substack{u \in S \\ V^+(u) \cap T = \emptyset}} \max_{v \in V^+(u)} w_{u,v}(\phi_{[S]}(v) - \phi_{[S]}(u)) \\
&\quad + \sum_{\substack{(u,v) \in E \\ u \in S, v \in T}} w_{u,v}(\phi_{[T]}(v) - \phi_{[T]}(u)) \\
&= C_{\phi_{[S]}} + \sum_{\substack{(u,v) \in E' \\ u = a_s}} w_{u,v}(\phi(v) - \phi(u))
\end{aligned}
$$

Which gives us:

$$
\begin{aligned}
\mathit{WSC}_G(\phi_{[S]} + \phi_{[T]}) &= C_{\phi_{[S]}} + \sum_{(u,v) \in E'} w_{u,v}(\phi_{[T]}(v) - \phi_{[T]}(u)) \\
&= C_{\phi_{[S]}} + \mathit{WLA}_{G'}(\phi_{[T]})
\end{aligned}
$$

Let us suppose that $\phi[S] + \phi'$ is not optimal, i.e., $\mathit{WSC}_G(\phi[S] + \phi') > \mathit{WSC}_G(\phi[S] + \phi[T])$.

Thus we have $\mathit{WLA}_{G'}(\phi') = \mathit{WSC}_G(\phi[S] + \phi') - C_{\phi[S]} > \mathit{WSC}_G(\phi[S] + \phi[T]) - C_{\phi[S]} = \mathit{WLA}_{G'}(\phi[T])$, which contradicts the fact that ϕ' is optimal for *WLA* on G'. □

Appendix A.9. Proof of Lemma 10

Proof. Let ϕ be a valid schedule for *WSC* on G.

$$
\begin{aligned}
\mathit{WSC}_G(\phi) &\geqslant \mathit{WLA}_{G'}(\phi) \\
\sum_{u \in V} \max_{v \in V^+(u)} w_{u,v}(\phi(v) - \phi(u)) &\geqslant \sum_{(u,v) \in E'} w'_{u,v}(\phi(v) - \phi(u))
\end{aligned}
$$

because G and G' only differ at the start:

$$
\begin{aligned}
\max_{v \in V^+(a_s)} w_{s,v}(\phi(v) - \phi(a_s)) &\geqslant \sum_{v \in V'^+(s)} w'_{s,v}(\phi(v) - \phi(a_s)) \\
w_s \max_{v \in V^+(a_s)} \phi(v) &\geqslant \sum_{v_i \in V'^+(a_s)} w_i \phi(v_i) \\
\sum_{v_i \in V'^+(a_s)} w_i \max_{v \in V^+(a_s)} \phi(v) &\geqslant \sum_{v_i \in V'^+(a_s)} w_i \phi(v_i)
\end{aligned}
$$

□

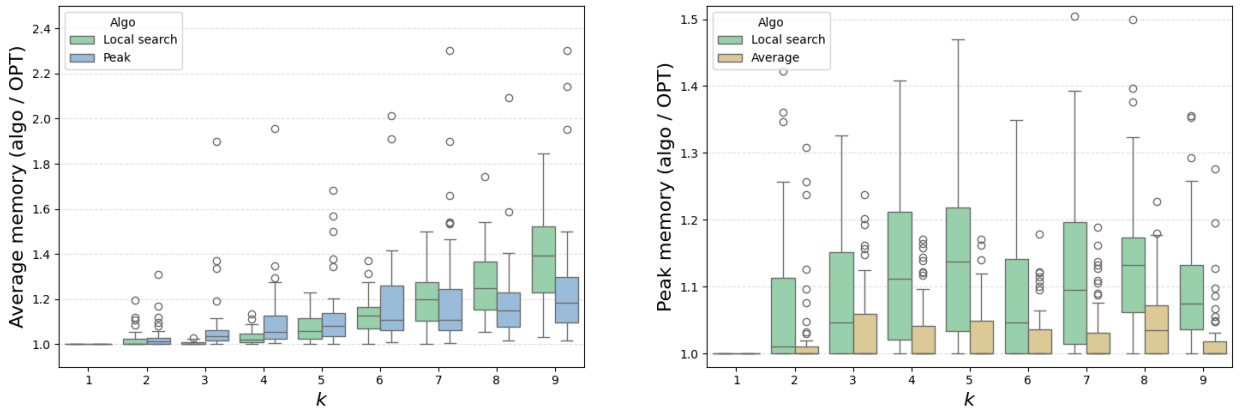
Appendix B. Parameter study

This appendix aims at strengthening the results from the experimental part of this paper by experimenting more thoroughly the different graph parameters. We first look at varying parameters for k-chains graphs for the *WSC* metric, then for SP-graphs for the *WLA* metric, each time comparing the relative performance of the algorithms and heuristics for both the peak and average metric.

Recall that Table 2 recapitulates the different parameters that have been studied, as well as their base value and variation range. Whenever a parameter is not specified in a figure, its value is set to the one from the *base value* column of this table.

Appendix B.1. k-chains

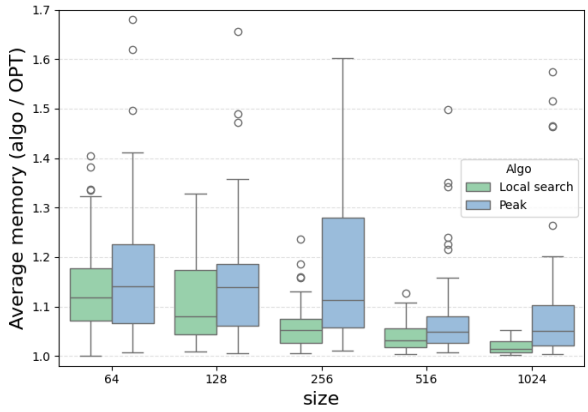
We start by comparing the Local-Search heuristic to Opt and Peak for *WSC* on k-chains. Our heuristics comparison shows that Local-Search comes very close to the optimal. We now compare it to Peak, and modify four parameters: k, n, and the node and edge weight distributions. For the weight distribution, we follow a uniform distribution between 1 and $max\ weight$. Figure B.19 shows the results for varying values of k. A bigger k means more parallelism, but also an increase in the number of cuts. Because our Local-Search only explores a fraction of these cuts, it is to be expected that its relative performance degrades with increasing k. More interestingly, it still remains fairly close to PeakSPAlgo for the peak metric, rarely losing more than 20%. In Figure B.20, we observe the converse: as the size increases, the shape of the graph is "less parallel" as k is fixed, making the problem easier. Here, our local search comes very close to Opt for the average metric, though it is still fairly underperforming for the peak. Finally, Figures B.21 and B.22 show that the distribution of edge and node weights does not significantly impact the results. The only significantly different case is when the edge weights are uniform (and equal to 1): Local-Search then always reaches optimality for the average, and both Local-Search and Opt reach optimality for the peak. This confirms that the choice of 10 for the default value is meaningful, as it is sufficient to express the difficulty of the problem.



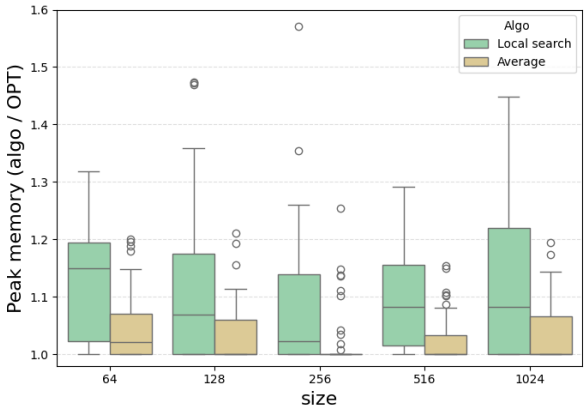
(a) Average memory consumption

(b) Peak memory consumption

Figure B.19: Relative performance of AvgSPAlgo, PeakSPAlgo and Local-Search on k-chains with an increasing value for k.

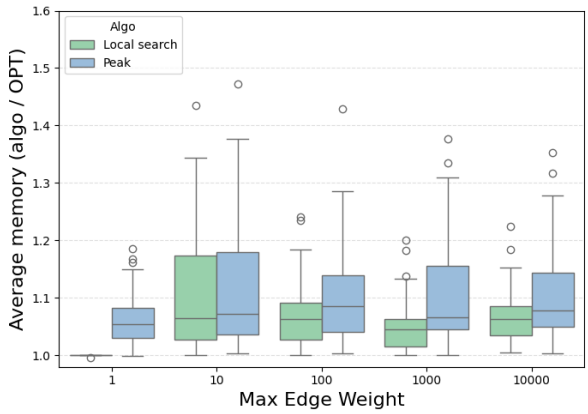


(a) Average memory consumption

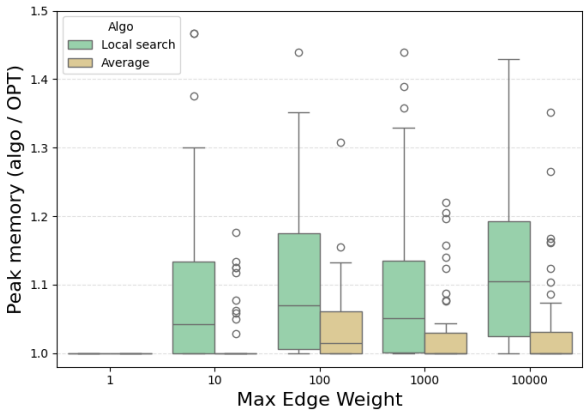


(b) Peak memory consumption

Figure B.20: Relative performance of AvgSPAlgo, PeakSPAlgo and Local-Search on k-chains with an increasing size.

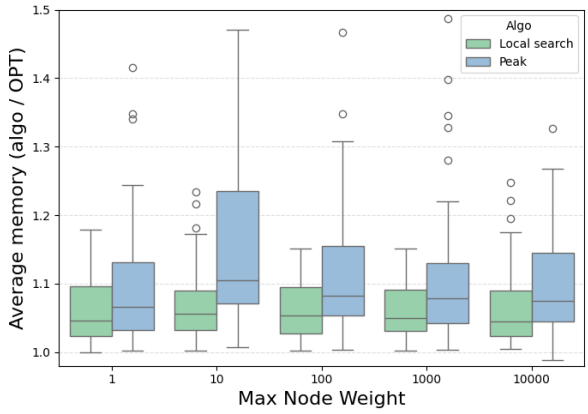


(a) Average memory consumption

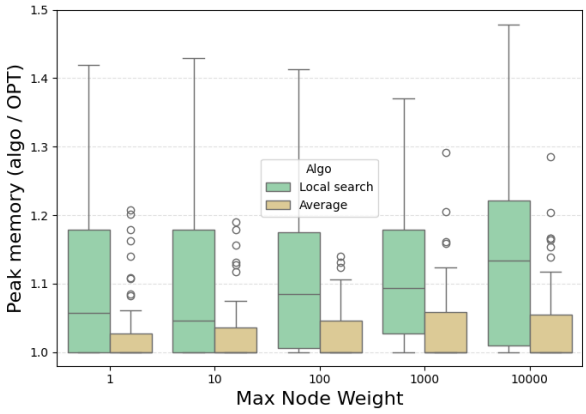


(b) Peak memory consumption

Figure B.21: Relative performance of AvgSPAlgo, PeakSPAlgo and Local-Search on k-chains with an increasing value of maximum edge weight.



(a) Average memory consumption

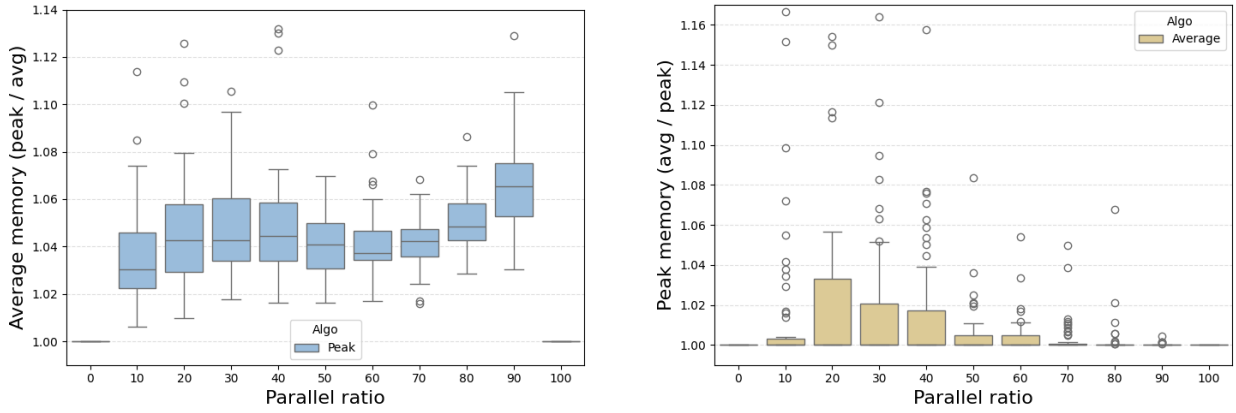


(b) Peak memory consumption

Figure B.22: Relative performance of AvgSPAlgo, PeakSPAlgo and Local-Search on k-chains with an increasing value of maximum node weight.

Appendix B.2. SP graphs

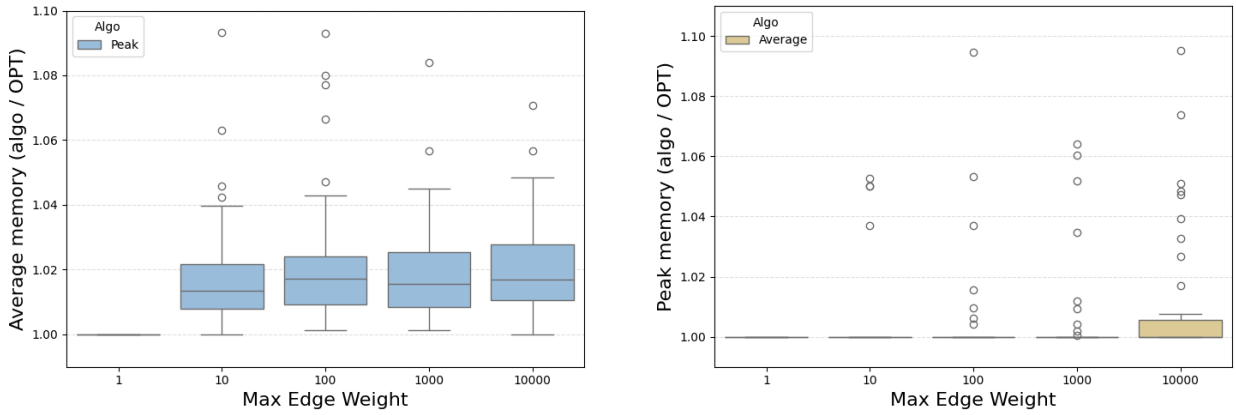
In this section, we now compare the PeakSPAlgo algorithm and AvgSPAlgo heuristic for the *WLA* problem on SP graphs. Here again, we modify the value of three parameters: parallelism ratio (Figure B.23) and maximum weight for both edges and nodes (Figure B.24 and Figure B.25). We make the same observations as for k-chains, that is that the value of the maximum weights does not change the performance of any heuristic, except for the specific case of uniform edge weights where AvgSPAlgo becomes optimal for both metrics.



(a) Average memory consumption

(b) Peak memory consumption

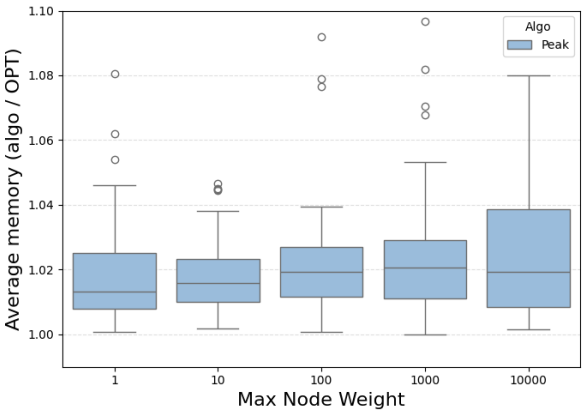
Figure B.23: Relative performance of AvgSPAlgo and PeakSPAlgo on SP-graphs with an increasing parallelism ratio.



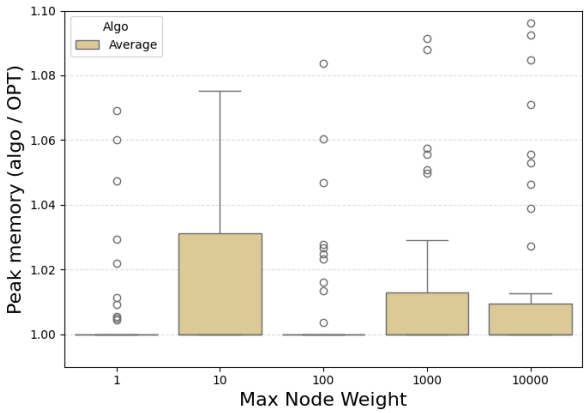
(a) Average memory consumption

(b) Peak memory consumption

Figure B.24: Relative performance of AvgSPAlgo and PeakSPAlgo on SP-graphs with an increasing value of maximum edge weight.



(a) Average memory consumption



(b) Peak memory consumption

Figure B.25: Relative performance of AvgSPAlgo and PeakSPAlgo on SP-graphs with an increasing value of maximum node weight.