Efficient Rematerialization for Deep Networks

Ravi Kumar
Google Research
Mountain View, CA 94043
ravi.k53@gmail.com

Manish Purohit
Google Research
Mountain View, CA 94043
mpurohit@google.com

Zoya Svitkina
Google Research
Mountain View, CA 94043
zoya@google.com

Erik Vee
Google Research
Mountain View, CA 94043
erikvee@google.com

Joshua R. Wang
Google Research
Mountain View, CA 94043
joshuawang@google.com

Abstract

When training complex neural networks, memory usage can be an important bottleneck. The question of when to rematerialize, i.e., to recompute intermediate values rather than retaining them in memory, becomes critical to achieving the best time and space efficiency. In this work we consider the rematerialization problem and devise efficient algorithms that use structural characterizations of computation graphs—treewidth and pathwidth—to obtain provably efficient rematerialization schedules. Our experiments demonstrate the performance of these algorithms on many common deep learning models.

1 Introduction

The world of deep learning is moving toward bigger model architectures. The recent successes in speech, language understanding, vision, and others have repeatedly demonstrated that bigger and deeper models yield the best results for a task, thereby advancing the state of the art. In addition to the size, the models themselves and the methods to train them are becoming increasingly complex and intricate in terms of data dependencies, gradient propagation, optimization steps, etc. Specialized hardware such as GPUs and AI accelerators have been vastly influential in training these complex models. They are particularly helpful from a computational point of view, but are limited by memory capacity that falls short of the peak demands of training these large models. Since memory turns out to be a bottleneck, it becomes an issue of feasibility—can a given model be trained at all?

While the growing model complexity is the root cause of severe demands on memory, the actual schedule in which the computation is carried out also plays a critical role in determining peak memory requirements. To see why, it is helpful to view the computational steps in training these models as a directed acyclic graph (e.g., Figure 1) whose nodes represent operations and directed edges represent data dependencies. (In TensorFlow parlance, this is a *dataflow* graph.) Each node consumes a set of inputs from its incoming edges, does some computation, and outputs the result of this computation on its outgoing edges; it is assumed that both inputs and outputs of this computation are to be held in memory. The order in which the nodes are computed, i.e., the schedule, will determine
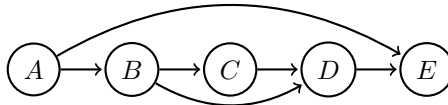

Figure 1: The schedule $\langle A, B, C, D, E \rangle$ needs four units of memory—while computing D, two units are needed for inputs to D, one for output from D, and one unit to keep the output of A as an input to E. The schedule $\langle A, B, C, D, A, E \rangle$ needs three units of memory—at node D the output of A need not be retained in memory since it will be recomputed right after computing D.

the peak memory usage. Indeed, consider Figure 1, where the output of each node occupies one unit of memory. Computing the nodes in the order $\langle A, B, C, D, E \rangle$ would need four units of memory, whereas computing them in the order $\langle A, B, C, D, A, E \rangle$ would only need three units of memory (see caption of Figure 1). This latter order involves *rematerializing* the output of node A instead of keeping it in memory. As this example illustrates, there can be a time-memory trade-off in our choice of schedule, where recalculating intermediate results can reduce what we store in memory. Judiciously choosing an appropriate schedule may make larger models feasible. In this paper we consider this rematerialization problem: given a computation graph as an input, construct a schedule, possibly rematerializing some nodes, that uses as little peak memory as possible[1].

When studied on computation graphs derived from training neural networks (i.e., graphs with forward computation and backward computations), rematerialization is often referred to as gradient checkpointing [10, 6, 8, 1, 15]. Of course, there are many other techniques to try to reduce memory usage, such as reusing memory regions [19] and trying to use both GPU and CPU memory [17, 16]. Rematerialization is a particularly nice approach because it only changes how the computation is done, but has no risk of changing the final result.

Compared to the gradient checkpointing line of work, we do not assume we have a forward/backward computation, but rather show how certain structural properties of the graph can be used to obtain a good solution. In particular, we identify *treewidth* of this graph as a key quantity that can be algorithmically exploited to yield a schedule with provable bounds on its length and peak memory usage. Informally, our main result is that there is a polynomial time algorithm that, given an n-node computation graph with treewidth k and unit memory output at each node, constructs a schedule of length $O(kn^{\log k})$ and peak memory usage of $O(k \log n)$. This algorithm uses a tree decomposition of the computation graph, which yields balanced separators and offers a natural way to partition the computations into independent sub-computations while allowing us to bound the memory use through a charging argument. Note that while finding the optimal tree decomposition is computationally hard, there are efficient approximation algorithms and heuristics, which makes our algorithm efficient, practical, and easy to realize. We demonstrate its efficacy by applying it to training large networks including feedforward, residual, and transformer networks. In all these cases, our schedule yields significant savings in peak memory over baseline schedules, both with and without rematerialization.

We also design a different algorithm that produces schedules that are asymptotically more efficient. This algorithm relies on the path decomposition of the computation graph and is more intricate, with an involved analysis; but currently less practical. This result, however, hints at the intriguing possibility of another structural property of the graph that better captures its rematerialization potential.

2 Preliminaries

2.1 Computation Graphs and Schedules

The input to our algorithms is a *computation graph*. Each node in this graph represents an operation that takes as input zero or more tensors and produces a single tensor as an output (this assumption is for simplicity). Let $G = (V, E)$ be a directed acyclic computation graph. For $u, v \in V$, a directed edge $(u, v) \in E$ represents data dependency, meaning that the output of node u is an input to node v. We are also given a *final* node $f \in V$ whose output tensor is required to be held in memory at the end of the computation. We assume, without loss of generality, that f has out-degree zero (i.e. no other operations use the tensor produced by f) and all nodes in G are needed to compute f. For any node $u \in V$, let $\text{in}(u)$ denote the immediate predecessors of u, i.e., $\text{in}(u) = \{u' \mid (u', u) \in E\}$. Let $n = |V|, m = |E|$, and $[n] = \{1, \ldots, n\}$. Throughout, $\log(\cdot)$ means $\log_2(\cdot)$.

A *schedule* for a computational graph $G = (V, E)$ is a sequence $\sigma = \sigma(G) = \langle u_1, \ldots, u_t \rangle$ of nodes in V with the following properties: (i) the final node f is represented in the schedule, and (ii) each node in the schedule occurs only after all of its predecessors, i.e., for each $j \in [t]$ and for each $u' \in \text{in}(u_j)$, there is some $j' < j$ such that $u_{j'} = u'$. Let $\text{prev}(u, j) = \max\{j' < j \mid u_{j'} = u\}$ be the most recent time u occurs in the schedule before step j. Note that a node in G can occur more than once in $\sigma(G)$ and this is precisely what enables the schedule length–memory usage trade-off.

---

[1] Of course, the real goal is to keep peak memory under the memory available and while minimizing the time to compute the schedule. Our results are easier to understand when viewed from a purely memory-minimization standpoint, but it is possible to stop our recursion early to obtain other trade-off points.

A schedule naturally implies time and memory bounds for computing G. Let $L(u)$ be the length of node u, representing the time required to execute the corresponding operation. The *length* of a schedule is given by $L(\sigma) = \sum_{i=1}^{t} L(u_i)$. Let $T_{\text{onepass}} = \sum_{u \in V} L(u)$ be the time required to execute every operation of the graph once. It lower bounds the length of any valid schedule.

The *peak memory* usage of the schedule, $M(\sigma)$, though intuitive, is a bit cumbersome to formalize. For $i \in [t]$, first define the set of tensors that need to be held in memory at step i as

$$U_i = \{u_i\} \cup \text{in}(u_i) \cup \bigcup_{j > i} \{u' \in \text{in}(u_j) \mid \text{prev}(u', j) \leq i\}.$$

Let $s(u)$ denote the size of the tensor output by node u. Now, the memory of the schedule at step i is $M(\sigma, i) = \sum_{u' \in U_i} s(u')$. Finally, $M(\sigma) = \max_{i=1}^{t} M(\sigma, i)$. The goal of an algorithm alg is to produce a schedule $\text{alg}(G)$ of G that minimizes the peak memory. Let $M_{\text{in}} = \max_{u \in V} \{\sum_{u' \in \text{in}(u)} s(u')\}$ be the maximum input size needed to compute any tensor. Let $M_{\text{max}} = \max_{u \in V} s(u)$ be the maximum size of any tensor. Clearly, for any schedule σ, $M(\sigma) \geq \max\{M_{\text{in}}, M_{\text{max}}\}$.

2.2 Treewidth and Tree Decompositions

Treewidth is a well-studied graph parameter expressing how close an undirected graph $G = (V, E)$ is to a tree. Intuitively, if a problem is easy on trees, then one might hope that it remains easy on graphs of small treewidth. Formally, the treewidth of a graph is defined via the notion of tree decompositions. A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(\mathcal{X}, T)$, where $\mathcal{X} \subseteq 2^V$ is a set of *bags*, with each bag a subset of the nodes, and T is a tree on the bags $\mathcal{X}$. The bags and tree must satisfy the following three properties: (i) each node in V is in some bag of $\mathcal{X}$, (ii) for each edge $(u, v) \in E$, both endpoints are together in some bag of $\mathcal{X}$, and (iii) for each node $v \in V$, the bags containing it (i.e., $\{X \in \mathcal{X} \mid v \in X\}$) form a connected subgraph of T.

Naturally, there are many tree decompositions of a particular graph G, including the trivial one that places all nodes into a single giant bag ($\mathcal{X} = \{V\}, T = \{\}$). We measure a tree decomposition by its *width*, which is the maximum bag size minus one: $\max_{X \in \mathcal{X}} |X| - 1$. The treewidth $\text{tw}(G)$ of G is the minimum width of any tree decomposition. We refer to $|\mathcal{X}|$ as the *size* of the decomposition. Note that $\text{tw}(G)$ can range from 1 (a tree) to $n - 1$ (a clique, and hence maximally far from a tree).

We will use treewidth and tree decompositions of our directed computation graphs. When doing so, we are actually referring to the undirected graph obtained by forgetting the direction of every edge. It is known that series-parallel graphs have a treewidth of two and control-flow graphs of all programs written in C (without goto statements) have a treewidth of at most six [20]. We postulate that computation graphs of neural networks in the inference mode also have similarly low treewidth and that, given a computation graph G for a neural network in the inference mode, the computation graph for training the network via backpropagation has treewidth at most twice as that of the original graph. Experimentally, we observe that computation graphs for training many common deep network architectures (ResNet, Transformer, and feedforward networks) have small treewidth (see Table 1).

Our results fall under the purview of fixed-parameter tractability, which studies the complexity of problems under particular parameters. Typically, we would hope to find an exact algorithm (computing the absolute memory-minimizing schedule) when treewidth is small. Unfortunately, this seems unlikely; such results typically come from Courcelle's theorem [7], which states that if a graph property can be expressed in second-order monadic logic, then it can be checked for in fixed-parameter tractable time relative to treewidth. Rematerialization is known to be PSPACE-complete [9]. If it were expressible in second-order monadic logic, then it would lie in the polynomial hierarchy (PH) and then PSPACE would collapse to PH. Hence, we must settle for approximation algorithms.

3 Efficient Rematerialization via Tree Decomposition

Our main algorithm uses a tree decomposition of the computation graph for a divide-and-conquer approach. The tree decomposition of a graph allows us to find balanced separators of small size. Additionally, the connectivity property of the tree decomposition guarantees that the nodes in the different components can be computed independently of each other except for interactions via the separator. Using these ideas, we recursively compute a memory-efficient schedule.

First, we consider the size of a tree decomposition and argue that it can be bounded.

Lemma 1. *Given an undirected graph $G = (V, E)$ and its tree decomposition $(\mathcal{X}, T)$ of width k, we can find another tree decomposition $(\mathcal{X}', T')$ of width k and size at most n in $O(|\mathcal{X}| \cdot (k + \alpha(|\mathcal{X}|)))$ time, where $\alpha(\cdot)$ is the inverse Ackermann function.*

Proof. The idea is to post-process $(\mathcal{X}, T)$ by repeatedly merging every adjacent pair of bags for which one bag is a subset of the other. This can be done with a single pass over all edges in T, since any adjacent pair of bags which cannot be merged at any time can never be merged in the future. For the sake of contradiction, imagine that bags X_1 and X_2 could not be merged due to some node $v \in X_1, v \notin X_2$. This problematic node will always be in X_1 since merging two bags only results in the addition of nodes to a bag. At the same time, it can never get added to X_2 because all bags that contain v are connected and hence X_1 is the only bag in the neighborhood of X_2 that contains v.

We can keep track of these merges using a standard Union-Find data structure on the $|\mathcal{X}|$ bags, which costs $O(\alpha(|\mathcal{X}|))$ time per operation. We perform at most $|\mathcal{X}|$ merges, which cost a total of $O(|\mathcal{X}| \cdot \alpha(|\mathcal{X}|))$ time. To check whether one bag is a subset of another, we can put the larger bag in a hash set and perform $k + 1$ membership checks. Hence we can perform all these checks in $O(|\mathcal{X}| \cdot k)$ time. Hence the overall time is the claimed $O(|\mathcal{X}| \cdot (k + \alpha(|\mathcal{X}|)))$.

We can see why this post-processing procedure works by taking the resulting tree decomposition $(\mathcal{X}', T')$ and rooting it at an arbitrary bag. Each non-root bag must contain a node not found in its parent bag because otherwise the bag should have been merged with its parent bag. Since the set of bags containing a node is connected, this assigns a unique node $v \in V$ to every non-root bag. Furthermore, the root cannot be empty since then it would have been merged, and its nodes cannot be assigned to any other bag due to the same property. Hence we can assign it one of these nodes. Since we have assigned each bag a unique node $v \in V$, there can be at most n bags. □

A classic result shows that a tree always has a balanced node separator.

Theorem 2 (Jordan [14])**.** *Any tree on n nodes has a node whose removal disconnects the tree into components of size at most $n/2$.*

Applying Jordan's theorem on the tree decomposition $(\mathcal{X}, T)$ directly yields the following lemma.

Lemma 3 (Balanced Separator)**.** *Given a tree decomposition $(\mathcal{X}, T)$, we can find, in time $O(|\mathcal{X}|)$, a bag $X^\star \in \mathcal{X}$ such that each connected component of $(\mathcal{X}, T) \setminus \{X^\star\}$ contains at most $|\mathcal{X}|/2$ bags.*

Our divide-and-conquer approach chooses a *balanced separator* $X^\star$ of the tree decomposition so that removing it results in subtrees with at most $|\mathcal{X}|/2$ bags each. Combining with Lemma 1, this guarantees that there are at most $\log n$ levels of recursion. Finding such a bag is a standard technique.

With these two ideas, we present Algorithm 1, which is a recursive function that schedules a subset V' of nodes with a requirement that the schedule contains all nodes in a specified subset S. It breaks the graph using the balanced separator, and schedules the predecessors of a node v in each of the resulting components before scheduling v itself. The produced schedule includes annotations about which tensors to keep in memory or to remove, which is just for ease of analysis, as in practice memory usage can be inferred from a schedule of operations. Initially, the function is called with arguments $(G, V, (\mathcal{X}, T), \{f\})$, where $f \in V$ is the final node.

Lemma 4. *Algorithm 1 produces a valid rematerialization schedule.*

Proof. The base case of the recursion is when there is a single bag in the tree decomposition, in which case we make no recursive calls and simply compute the desired outputs in some topological order. Inductively, we assume that the algorithm works correctly on tree decompositions with less than b bags, and show that it also works when there are b bags.

The reasoning centers around what happens when we remove the balanced separator $X^\star$ from the tree decomposition. Since the bags containing any node $v \in V$ form a connected component, if v is in two or more components of $\mathcal{C}$, it must also be in the separator $X^\star$. Hence this separator partitions our graph: for each subgraph $(\mathcal{X}', T') \in \mathcal{C}$, we can define the nodes in it to be $V' := \left(\bigcup_{X \in \mathcal{X}'} X \right)$ and we know that these V' together with $X^\star$ form a partition of V. Furthermore, by the definition of tree decomposition, we know that each edge must be present in some bag, so the only edges involving some V' go to other nodes in the same V' or to $X^\star$.

4

Algorithm 1: Efficient Rematerialization via Tree Decomposition.

Function: TWRemat $(G, V', (\mathcal{X}, T), S)$:

Data: $G = (V, E)$ a computation graph, $V' \subseteq V$ a subset of nodes to restrict to, $(\mathcal{X}, T)$ a tree decomposition of G restricted to V', $S \subseteq V'$ a subset of nodes to compute.

Result: An annotated schedule consisting of nodes in V' that contains all nodes in S.

if *this is the top level recursive call* **then**
> Shrink the size of the tree decomposition to at most n bags using Lemma 1;

Find a balanced separator (bag) $X^\star \in \mathcal{X}$ using Lemma 3;
Make a copy of $(\mathcal{X}, T)$, removing bag $X^\star$ and removing nodes of $X^\star$ from every other bag. Let $\mathcal{C}$ be the set of connected components that result (each a tree decomposition $(\mathcal{X}', T')$);
Initialize $schedule = \langle \rangle$;

for *node $v \in X^\star$ in any topological order (according to G)* **do**
> **for** *connected component $(\mathcal{X}', T') \in \mathcal{C}$* **do**
>> Let $S' = \text{in}(v) \cap \left(\bigcup_{X \in \mathcal{X}'} X \right)$ and $V'' = V' \cap \left(\bigcup_{X \in \mathcal{X}'} X \right)$;
>> Extend $schedule$ with TWRemat $(G, V'', (\mathcal{X}', T'), S')$ to compute the inputs of v in this component;
>> Add annotation to $schedule$ to keep S' in memory;
>
> Add v to $schedule$, keeping it in memory, and freeing all of its inputs not in $X^\star$;

for *connected component $(\mathcal{X}', T') \in \mathcal{C}$* **do**
> Let $S' = (S \setminus X^\star) \cap \left(\bigcup_{X \in \mathcal{X}'} X \right)$ and $V'' = V' \cap \left(\bigcup_{X \in \mathcal{X}'} X \right)$;
> Extend $schedule$ with TWRemat $(G, V'', (\mathcal{X}', T'), S')$ to compute the remaining outputs in this subgraph;
> Add annotation to $schedule$ to keep S' in memory;

Add annotation to $schedule$ to free the unneeded balanced separator nodes $X^\star \setminus S$;
return $schedule$;

We claim that whenever a recursive call to TWRemat is made (with arguments V'' and S'), all predecessors of S' which are not in V'' are already in memory of the caller's schedule. Consider some node $u \in S'$ and its predecessor $u' \notin V''$. It must be that $u' \in X^\star$ by the preceding discussion that an edge involving $u \in V''$ can only go to V'' or to $X^\star$. Suppose that the recursive call is made from the nested for loops in which the outer loop is processing a node $v \in X^\star$. Since u' is a predecessor of u and u is a predecessor of v (which we know from $u \in S'$), u' must come before v in a topological order of G. Thus, it has already been scheduled in a previous iteration of the outer for loop. If the recursive call is made from the other for loop, then all nodes of $X^\star$ are scheduled and in memory by that time.

We conclude that the precedence constraints are respected by the schedule—with respect to nodes in V'' by induction, and with respect to nodes in $X^\star$ by the above discussion. Furthermore, all nodes of S are scheduled in the later loop. $\qquad \square$

Theorem 5. *Given a computation graph $G = (V, E)$, its tree decomposition $(\mathcal{X}, T)$ of width at most k, and $S \subseteq V$ a subset of nodes to compute, Algorithm 1 runs in time $O(|\mathcal{X}| \cdot (k + \alpha(|\mathcal{X}|)) + kn \log n + kn^{1+\log(k+2)})$ and computes a rematerialization schedule of length $O(T_{\text{onepass}} \cdot kn^{\log(k+2)})$ that requires $O((M_{\text{in}} + kM_{\text{max}}) \log n)$ memory.*

Proof. We begin with the running time. We pay an upfront cost of $O(|\mathcal{X}| \cdot (k + \alpha(|\mathcal{X}|)))$ to invoke Lemma 1. We pay a total time of $O(n \log n)$ to invoke Lemma 3, since (i) each invocation requires linear time and (ii) we recurse into subcalls that partition the tree decomposition into pieces that are at most half the current size. Note that we will need to memoize these balanced separators to avoid recomputing them over and over. As a result we have $O(\log n)$ levels of recursion and over all subcalls in a level we do $O(n)$ work. The processing of tree decompositions (removing a bag, removing the nodes of a bag from other bags) can be done in $O(kn)$ time and follows the same recursion as finding balanced separators (i.e. subcalls partition the tree decomposition and have at most $O(\log n)$ depth), for a total of $O(kn \log n)$ work. Finally, the output is $O(kn^{1+\log(k+2)})$ in size (see the schedule length analysis), and we spend linear time to compute it.

Next, we check the schedule length. At each level, we make a recursive call to a particular subgraph $(|\mathcal{X}^*| + 1) \leq (k+2)$ times, so we wind up amplifying the total work by a factor of at most $k+2$ at each recursive level (except for the final recursive level, where we make no recursive calls). Carefully counting, we need at most $\lceil \log n \rceil + 1$ levels of recursion so we have amplified the computation time by $O((k+2)^{\lceil \log n \rceil})$. Since $a^{\log b} = b^{\log a}$, this is an amplification of $O(kn^{\log(k+2)})$. In other words, we make at most $O(kn^{\log(k+2)})$ copies of any operation, so this takes at most $O(T_{\text{onepass}} kn^{\log(k+2)})$ time as claimed.

Finally, we check the memory needed by the schedule. Consider a particular segment of the schedule and the `TWRemat` function call that added it. The content of memory at this place in the schedule can be charged to the active function calls at that point of execution as follows: we charge to a recursive level everything that it annotated to keep in memory except its outputs, which are charged to its caller. The balanced separator requires $O(kM_{\max})$ memory, one set of inputs to a balanced separator node requires $O(M_{\text{in}})$ memory (but since we free these we only need to hold one set of inputs). Since there are $O(\log n)$ levels of recursion, this results in a total memory of $O((M_{\text{in}} + kM_{\max}) \log n)$. $\qquad \square$

What remains is to compute a tree decomposition efficiently. Our corollary utilizes an approximation algorithm that runs in $n \cdot 2^{O(\text{tw}(G))}$ time and computes a decomposition of width at most $(5\text{tw}(G)+4)$ and size $O(n)$ [2]. Our actual implementation uses a minimum fill-in heuristic [3], which yields good tree decompositions.

Corollary 6. *Given a computation graph $G = (V, E)$, there is an algorithm that runs in $2^{O(\text{tw}(G))}n + O(n \cdot (\text{tw}(G) + \alpha(n)) + \text{tw}(G)n \log n)$ time and computes a rematerialization that requires computation time $O(T_{\text{onepass}} \text{tw}(G)n^{\log_2(5\text{tw}(G)+6)})$ and memory $O((M_{\text{in}} + \text{tw}(G)M_{\max}) \log n)$.*

4   Experiments

We experimentally evaluate the performance of our rematerialization algorithm on computational graphs for training commonly used deep neural networks. We remark that the memory optimizations proposed in this paper ensure that the computational graph is faithfully executed; this ensures that the gradients obtained at each train step are exactly equivalent to those obtained without any optimization, and hence do not affect convergence. We measure the theoretical peak memory usage of a schedule via an optimal static memory allocation plan. Since the primary purpose of these experiments is to evaluate the effect of rematerialization on memory usage, we do not consider other heuristic memory optimizations such as in-place operations, operation fusion, and buffer aliasing. Finally, we also measure the length of the schedule obtained by the different algorithms. For simplicity, in these experiments, we assume that each operation takes unit cost.

Algorithms. We compare the performance of the following three algorithms.

(i) `NoRemat`: Schedules all operations in a topological sort without any rematerialization.

(ii) `GreedyRemat`: This is an implementation of a greedy heuristic for rematerialization used by XLA[2] that works as follows. Starting with a topological sort of all operations, it processes each operation sequentially. At each stage, if the current memory usage is over a specified memory limit, the algorithm attempts to rematerialize an already scheduled operation. In particular, the operation whose rematerialization maximizes the amount of reduction in memory usage is chosen greedily at each step. If the memory usage cannot be reduced, the algorithm moves on to the next operation.

(iii) `TWRemat`: This is an implementation of Algorithm 1 that uses a tree decomposition; we use the minimum fill-in heuristic [3] to find the tree decomposition.

Models and Setup. We evaluate all algorithms on different families of widely used deep networks.

(i) *Deep Residual Networks (ResNet):* We first consider deep residual networks (ResNet) [13] as an example of convolutional networks for image classification. We use the official implementation of the ResNet model for the ImageNet task in TensorFlow[3]. We use different configurations to measure the effect of network depth (number of convolutional layers) on memory requirements of schedules obtained by the algorithms.

---

[2] `www.tensorflow.org/xla`

[3] `github.com/tensorflow/models/blob/master/official/resnet/imagenet_main.py`

(ii) *Feed forward networks (FFN):* We consider a simple feed-forward neural network to illustrate the trends in peak memory usage of the schedules obtained by the different algorithms as a function of the network depth. For this experiment, we setup a simple feed-forward network with ReLU activations (number of hidden layers is varied) and randomly generated inputs and outputs. We use mean squared error loss and train using standard gradient descent.

Model	n	m	tw
ResNet200	17,705	27,312	11
FFN (100 layers)	3,217	4,447	6
Transformer Base	15,842	21,771	18

Table 1: Computation graph statistics.

(iii) *Transformer:* We also evaluate the memory savings obtained by our rematerialization algorithms for training the transformer [21] network. Again, we use the official implementation of Transformer in TensorFlow[4] with all hyperparameters set to recommended defaults.

Table 1 gives summary statistics for representative models from each family. Crucially, we observe that even the largest graphs have tree decompositions with small width.

4.1 Effect on Peak Memory Usage

We first demonstrate the effect of the depth of the network on the peak memory usage required for training the network. Figure 2 compares the performance of the three algorithms on the ResNet and Feed-forward models described above. As expected, we observe that the peak memory usage of `NoRemat` that does not perform any rematerialization increases linearly with the number of layers on both model families. The `GreedyRemat` algorithm yields modest improvements ($\approx$ 2x) in memory usage for the ResNet models but still shows a linear growth with number of layers. We observe that `GreedyRemat` yields very little memory savings on the feed forward network. On the other hand, the `TWRemat` algorithm consistently gives memory savings on both the model families (up to 10x) and the growth in peak memory usage is distinctly sublinear.

(a) ResNet
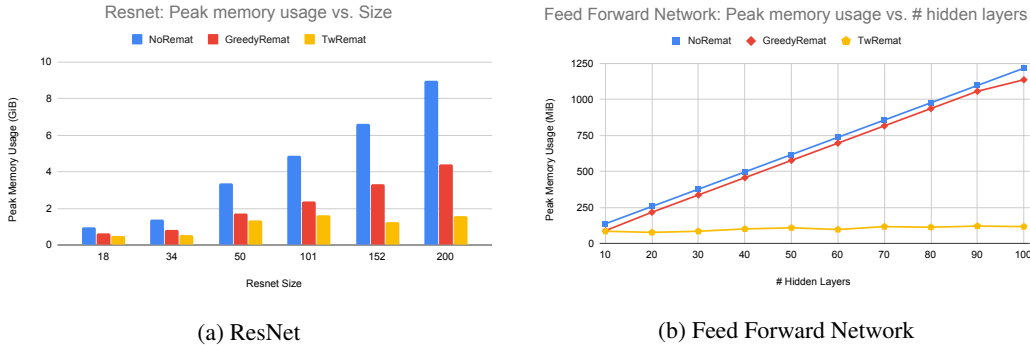
(b) Feed Forward Network

Figure 2: Peak memory usage vs. model depth.

Table 2 shows the memory usage and relative lengths of the schedules obtained by the three algorithms on two configurations of the transformer network. The `TWRemat` algorithm yields a 3.48x and 4.59x reduction in peak memory usage respectively, albeit at a cost of up to 10.6x in the schedule length.

	NoRemat	GreedyRemat		TWRemat	
	Mem. (GiB)	Mem. (GiB)	Rel. Len.	Mem. (GiB)	Rel. Len.
Transformer Base	3.97	2.92	1.21	1.14	10.61
Transformer Big	13.25	10.12	1.27	2.89	10.64

Table 2: Transformer: Peak memory usage and relative schedule lengths.

4.2 Effect on Schedule Length

Our algorithms are specifically designed to minimize peak memory consumption at the expense of additional computation. Figure 3 illustrates the increase in the schedule length relative to `NoRemat`.

---

[4] `github.com/tensorflow/models/blob/master/official/transformer/transformer_main.py`

We observe that `GreedyRemat` consistently yields schedules that are only marginally longer than the corresponding schedules of `NoRemat`. On the other hand, the schedules obtained via `TWRemat` are around 3x-4x longer. Despite the longer schedules, we expect the schedules produced by `TWRemat` to be beneficial in practice as the reduced memory usage allows the use of specialized hardware accelerators.
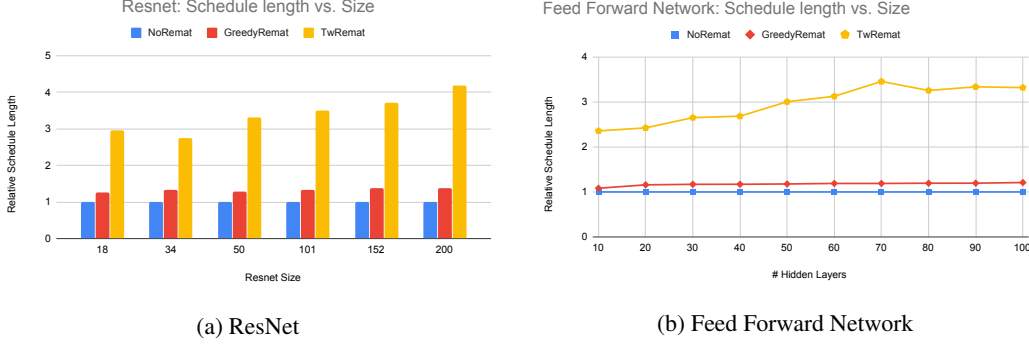

| (a) ResNet | (b) Feed Forward Network |

Figure 3: Schedule length vs. model depth.

4.3 Trading-off Memory Usage for Schedule Length

Algorithm 1 (`TWRemat`) uses the tree decomposition to find a balanced separator that breaks up the tree decomposition into smaller subtrees, and then recursively computes schedules to compute the required nodes in these subtrees. We observe that we can obtain a trade-off between memory usage and schedule length by preemptively stopping the recursion when the tree decomposition has few bags remaining. For any integer k, let `TWRemat` (k) be a variant of Algorithm 1 that stops the recursion when the tree decomposition has fewer than k bags. In the base case, we schedule the required nodes in an arbitrary topological order. In this notation, our `TWRemat` algorithm can be written as `TWRemat` (1). Indeed, by varying the recursion limit from $k = 1$ to $k = n$, we can interpolate between the `TWRemat` and `NoRemat` algorithms. Figure 4 shows the memory usage vs. schedule length trade-off obtained for the ResNet200 model.
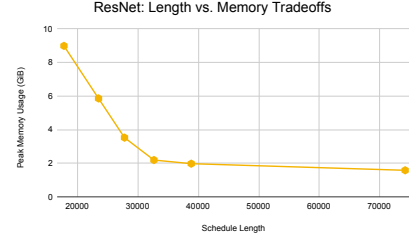

Figure 4: Resnet200: Mem. vs length.

5 Stronger Guarantees via Path Decomposition

Related to treewidth of a graph is the notion of *pathwidth*, which is defined as the minimum width of any *path decomposition*, where a path decomposition is a tree decomposition $(\mathcal{X}, T)$ under the additional constraint that T must be a path. We can order the bags according to the path and instead use the tuple $\mathcal{X} = (X_1, X_2, ..., X_{|\mathcal{X}|})$ to represent the path decomposition, where each $X_i \subseteq V$ is a bag and (path decomposition) edges run between X_i and X_{i+1}. We denote the pathwidth of a graph G by $\text{pw}(G)$. Assuming that a computation graph has a small constant pathwidth allows us to design an algorithm for rematerialization that leverages the path decompositions to yield stronger theoretical guarantees than in Theorem 5. In this section, we sketch the primary ideas, deferring the full algorithm and analysis to the Supplementary Material.

We first show one can add a directed Hamiltonian path (i.e., a *spine*) to any graph G so that the pathwidth of G only increases by a factor of ~ 2. This allows us to prove certain structural properties of the path decomposition. Suppose the vertices of G are ordered according to the spine, let $u_i \in V$ be the ith node, and let $last(X)$ denote the index of the last node in bag X. We show that if $X_c \in \mathcal{X}$ is a bag in the path decomposition that contains u_n, then for all $\ell < \ell' < c$, we have $last(X_\ell) \leq last(X_{\ell'}) \leq last(X_c)$ and for all $r > r' > c$, we have $last(X_r) \leq last(X_{r'}) \leq last(X_c)$.

Such a structural characterization allows a divide-and-conquer strategy that recurses on the *right* and *left* sides of the path decomposition. Unlike the tree decomposition algorithm where we argue that

the size of the tree decomposition reduces at each recursive call, the additional properties of the path decomposition allow us to argue that *both the size and width* of the decomposition decreases. The resulting algorithm yields a schedule that incurs a polylogarithmic increase in length (vs. polynomial blow up for the tree decomposition), but at the cost of polylogarithmic memory usage.

6 Related Work

Rematerialization has been considered in very limited settings for training deep networks. The work most relevant to ours is that of Chen et al. [6] and Gruslys et al. [11]. The former shows how to trade off memory and computation cost for simple chain-like networks. Their algorithm at a high level works by dividing a computation of length n into $\sqrt{n}$ many sub-computations, storing the internal states for each sub-computation and at the $\sqrt{n}$ check points; a second pass is needed to complete the computations. By recursing on this idea, one can get an $O(n \log n)$-pass algorithm using memory $O(\log n)$ for chain-like computations. Gruslys et al. [11] consider backpropagation through time and propose a dynamic-programming based approach for achieving the best time-memory trade off; their algorithm is tailored to work on RNNs. It is unclear how to extend either of these algorithms to work for general computation graphs, which is the focus of our work. There are some practical heuristics for rematerialization used in open-source efforts such as XLA; in fact, we used it as one of our baselines (`GreedyRemat`). Other heuristics including in-place operations and register sharing memory optimizations have been used in practice [5]. We, on the other hand, offer a principled approach to these problems.

Tree decomposition has been suggested as a tool to achieve time-memory trade off in register allocation problems in compilers [18, 4]. A recent blog post[5] informally suggests using tree decomposition for memory saving in deep networks in the context of gradient checkpointing,[6] which implements [6]. As noted control flow graphs of structured programs have treewidth ~ 6 [20]. Here, we work with the data flow graph to obtain a memory-efficient schedule, which may have larger treewidth in general.

View materialization in databases is also somewhat related to rematerialization [12]. The goal there is to pre-compute materialized views in order to efficiently answer future queries. While this is also a computation-memory trade-off, the end goals are clearly different from our setting.

7 Conclusions

We consider the rematerialization problem in the context of memory-efficient training of deep networks and obtain efficient algorithms based on tree decomposition for finding a provably good schedule with rematerialization. Although our path decomposition based algorithm yields asymptotically better schedules, the schedule length and memory depend exponentially on the pathwidth. It will be very interesting to make this algorithm more practical. Identifying the precise structural parameter that characterizes rematerialization of a given graph is a tantalizing research question.

References

[1] Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova. Optimal memory-aware backpropagation of deep join networks. Research Report RR-9273, Inria, 2019.

[2] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $(c^k n)$ 5-approximation algorithm for treewidth. *SICOMP*, 45(2):317–378, 2016.

[3] Hans L Bodlaender and Arie MCA Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI*, pages 311–321, 1992.

---

[5]`medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9`
[6]`github.com/openai/gradient-checkpointing`

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. Technical Report 1512.01274, arXiv, 2015.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. Technical Report 1604.06174, arXiv, 2016.

[7] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[8] Jianwei Feng and Dong Huang. Cutting down training memory by re-fowarding. Technical Report 1808.00079, arXiv, 2018.

[9] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. In *STOC*, pages 237–248, 1979.

[10] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *TOMS*, 26(1):19–45, 2000.

[11] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *NIPS*, pages 4132–4140, 2016.

[12] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[14] Camille Jordan. Sur les assemblages de lignes. *J. Reine Angew. Math*, 70(81):185–190, 1869.

[15] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. In *NeurIPS*, 2019.

[16] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by GPU memory optimization on TensorFlow. In *Proceedings of the ML Systems Workshop in NeurIPS*, 2017.

[17] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *IEEE/ACM International Symposium on Microarchitecture*, page 18, 2016.

[18] Ravi Sethi. Complete register allocation problems. *SICOMP*, 4(3):226–248, 1975.

[19] Koichi Shirahata, Yasumoto Tomita, and Atsushi Ike. Memory reduction method for deep neural network training. In *Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, 2016.

[20] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

A Stronger Guarantees via Path Decomposition

In this section, we describe an algorithm to find a schedule using polylogarithmic space and polylogarithmic blowup in schedule length for graphs having a low pathwidth. The main goal is to prove the following result.

Theorem 7. *Let $G = (V, E)$ be a computation graph with pathwidth $\mathrm{pw}(G)$. Then we can compute a schedule σ for G using memory $M(\sigma) = O((M_{\mathrm{in}} + M_{\max})4^{\mathrm{pw}(G)}(\log^{2\mathrm{pw}(G)+1} n))$ and length $L(\sigma) = 2^{O(\mathrm{pw}(G)\log\mathrm{pw}(G))}n(\log^{2\mathrm{pw}(G)+1} n)$.*

In the special case that G also has a directed path of length n (i.e., a directed path touching every node), then we can compute a schedule σ' for G using memory $M(\sigma) = O((M_{\mathrm{in}} + M_{\max})2^{\mathrm{pw}(G)}(\log^{\mathrm{pw}(G)-1} n))$ and length $L(\sigma) = 2^{O(\mathrm{pw}(G)\log\mathrm{pw}(G))}n(\log^{\mathrm{pw}(G)-1} n)$.

The proof is written across four subsections. In the first subsection, we explain how to do some preprocessing to establish a useful property that we will need for future subsections. Specifically, we show that for any graph G with pathwidth $\mathrm{pw}(G)$, we can add edges to G to create G' such that $\mathrm{pw}(G') \le 2\mathrm{pw}(G) + 3$ and further, G' contains a directed path of length n. In this case, we say G' has a *long spine*. Note that this is why we do even better on graphs that already come with a long spine; we can skip this preprocessing step.

The second subsection then leverages the long spine to prove useful structural properties of the path decomposition that will help us appropriately recurse. In particular, we explore ideas that will allow us to design an algorithm that reduces both the size (number of bags) and width (number of nodes in each bag) of the path decomposition as we recurse. For tree decompositions, we were only able to lower the number of bags; this is why we will be able to obtain improved bounds for path decompositions.

The third subsection helps us with the all important step of combining the schedules that work for two subproblems into a schedule for the current problem. To do so, we introduce the notion of "interleaved schedules" and prove several key properties about them that we will need for the algorithm.

Finally, the fourth subsection presents the algorithm and analyzes its correctness. It also bounds the length and memory usage for the resulting schedules.

A.1 Pathwidth-Preserving Spine-Addition

The goal of this subsection is to prove that a spine can be added to a computation graph while controlling the width of its path decomposition. It is important that we choose the spine to add, since it is not hard to construct counterexamples where the addition of a poorly-chosen spine increases pathwidth by a $\mathrm{poly}(n)$ factor. As a reminder, we restate the theorem we wish to prove.

Theorem 8. *Suppose we have a directed acyclic graph G on n nodes and a path decomposition of G with bag size $\mathrm{pw}(G) + 1$. Then there exists an algorithm that adds a spine to G while maintaining a valid path decomposition. The resulting path decomposition has bag size at most $2\mathrm{pw}(G) + 3$.*

Proof. We begin with some helpful notation. Given a path decomposition $(\mathcal{X}, T)$ we can label its bags $X_1, \ldots, X_b$ in path order. Define $\textsc{Interval}(u) \coloneqq \{i \in [b] \mid u \in X_i\}$. By the definition of path decomposition, this is always some contiguous interval $[\ell_u, r_u]$ since these bags form a subpath.

We will prove that Algorithm 2 has the desired properties. First, we will show that a spine is added to G. To do so, we first prove that all nodes are processed in some topological order. Clearly nodes are only processed in topological order, since we refuse to process any node before its in-neighbors. Hence it remains to show that all nodes are processed; which we do by contradiction. Fix an arbitrary topological order of G, and for the sake of contradiction suppose that v is the first node in this order which is not processed by our algorithm. All of its in-neighbors must be processed, or it wouldn't be the first such node. Furthermore, there must be at least one such in-neighbor, r. When its last in-neighbor is processed, it had no unexplored in-neighbors and hence v should have been processed. Hence, all nodes are processed.

We claim that when $\texttt{ProcessNode}(\ldots, u)$ is called, it adds a path to G starting from u and covering all nodes that appear in recursive subcalls to $\texttt{ProcessNode}$. This is because for the first recursive subcall, we already know there is a $u \to v_1$ edge and before subsequent recursive subcalls, we add an edge from the current end of the path to the next v_i. Since we already know $\texttt{ProcessNode}(\ldots, r)$ results in all nodes being processed, the path it adds must covers all nodes, i.e., must be a spine.

Now, we will show that we have properly updated the path decomposition and that no bag has more than $2\mathrm{pw}(G) + 3$ nodes. The former is by construction; we never add an edge without first ensuring that its endpoints share a bag. Regarding the latter, note that we begin with bags of size at most

---

Algorithm 2: Pathwidth-Preserving Spine Addition.

Function: `PathwidthPreservingSpineAddition(G, (X, T))`:

  Data: $G = (V, E)$ a DAG and $(\mathcal{X}, T)$ a path decomposition of G.

  Result: Adds a spine to G and updates the path decomposition accordingly.

  Add a node r to G, connect it to all other nodes, and add it to all bags $X_i \in \mathcal{X}$;

  Run `ProcessNode`$(G, (\mathcal{X}, T), \{\}, r)$;

  Remove node r from G and from all bags $X_i \in \mathcal{X}$;

Function: `ProcessNode(G, (X, T), S, u)`:

  Data: $G = (V, E)$ a DAG, $(\mathcal{X}, T)$ a path decomposition of G, $S \subseteq V$ a subset of explored nodes, and u a node of G.

  Result: Adds a path starting from u and covering all recursively processed nodes; returns the final node x of this path. Updates the path decomposition to handle this path and guarantees INTERVAL(x) includes r_u.

  Add u to the explored set S;

  Set the current final node $x \leftarrow u$;

  Let $v_1, v_2, \ldots, v_k$ be the out-neighbors of u which have no unexplored in-neighbors of their own, ordered by increasing right endpoint r_{v_i};

  for $i = 1, 2, \ldots, k$ **do**

    Update $x \leftarrow$ `ProcessNode`$(G, (\mathcal{X}, T), S, v_i)$;

    if $i < k$ **then**

      Extend the interval of x to the right until it includes $\ell_{v_{i+1}}$;

      Add edge $x \to v_{i+1}$ to G;

    else

      Extend the interval of x to the right until it includes r_u;

  return x;

$\mathrm{pw}(G) + 1$, we add r to all of them for bags of size at most $\mathrm{pw}(G) + 2$, we call `ProcessNode`$(\ldots, r)$ which will double the bags to size at most $2\mathrm{pw}(G) + 4$, and finally removing r from all bags results in bags of size at most $2\mathrm{pw}(G) + 3$ (our procedure only adds nodes to bags, so r is still in all bags).

The tricky part is proving that `ProcessNode`$(\ldots, r)$ at most doubles the size of all bags; this is due to two observations. The first observation is that if we examine how nodes are added to bags inside `ProcessNode`$(\ldots, u)$ ignoring all recursive subcalls, at most one node gets added to any bag and only to bags already containing u. After $x \leftarrow$ `ProcessNode`$(\ldots, v_i)$ we know that the right endpoint r_x is at least the right endpoint r_{v_i}. Hence during any iteration $i \in [k]$, at most one node gets added to any bag strictly to the right of r_{v_i}. But before iteration $i \in [k]$, nodes could only be added to bags as far right as r_{v_i}, since the left endpoint of an interval precedes the right endpoint of an interval and since $r_{v_1} < r_{v_2} < \cdots < r_{v_k}$. Nodes cannot be added to bags to the left of INTERVAL(u) because $r_{v_1} \in$ INTERVAL(u). Nodes cannot be added to bags to the right of INTERVAL(u) because $\ell_{v_k} \in$ INTERVAL(u).

The second observation is that the INTERVAL(u) is only altered after `ProcessNode`$(\ldots, u)$ completes. Hence `ProcessNode`$(\ldots, u)$ can only add a node to each bag that u was *originally in*. As a result, each bag can only gain a node for each node that was originally in it, so each bag at most doubles in size. This completes the proof. $\square$

A.2 Structural Properties for Long-Spined Graphs

Let $G = (V, E)$ be a directed acyclic graph on n nodes that has a long spine, i.e., there is a directed path of length n through G. Throughout, let $\mathcal{X} = X_1, X_2, \ldots, X_b$ be a path decomposition for G, and let $p = \mathrm{pw}(G) + 1$ be the maximum bag size in $\mathcal{X}$.

Since we have a path through the entire graph, there is a full topological ordering forced on us. Label the path in order $v_1 \prec v_2 \prec \cdots \prec v_n$, so v_1 is the first node and v_n is the last. It now makes sense to talk about the *last* element in a bag—it is the node v_i with the largest index i. Formally, for any bag X, let last(X) be the last node in X.

Path decompositions of DAGs with long spines have several useful properties which we will need. The first property states, roughly, that any path passing between two bags must also pass between any intermediate bag.

Lemma 9 (Intermediate Value Theorem for Bags). *Define G as above, and suppose we have three bags in our path decomposition, say X_k, X_{k^*}, and $X_{k'}$ with $k \leq k^* \leq k'$. Furthermore, suppose that $v_i \in X_k$ and $v_{i'} \in X_{k'}$ for some i, i'. Then there is an $i^* \in [\min(i, i'), \max(i, i')]$, such that $v_{i^*} \in X_{k^*}$.*

Proof. We induct on the difference between the nodes, $|i - i'|$. Our base cases are $|i - i'| = 0$ and $|i - i'| = 1$. The first base case $i = i'$ trivially follows from the definition of path decomposition (in particular, the bags that contain $i = i^* = i'$ form a contiguous interval). The second base case $|i - i'| = 1$ is a bit trickier to argue about. Our long spine implies an edge from v_i to $v_{i'}$ (or vice versa). No matter which way this edge runs, by the definition of path decomposition there is some bag X_ℓ that contains both v_i and $v_{i'}$. If this bag is to the right of our goal bag, i.e., $k^* \leq \ell$, then from the definition of path decomposition we can deduce that i is in the goal bag (it is in $k \leq k^*$ and $\ell \geq k^*$). If this bag is to the left of our goal bag, i.e., $\ell \leq k^*$, then from the definition of path decomposition we can deduce that i' is in the goal bag (it is in $\ell \leq k^*$ and $k' \geq k^*$). Hence in either case the we can find an i^* as desired.

The inductive case is similar to the second base case. We will focus on the situation where $i < i'$ (the proof for the $i > i'$ case is analagous). The long spine implies an edge from v_i to v_{i+1} and hence there is some bag X_ℓ that contains both v_i and v_{i+1}. If this bag is to the right of our goal bag, i.e., $k^* \leq \ell$, then we can apply our inductive hypothesis to the simpler problem $(k \leftarrow k, k^* \leftarrow k^*, k' \leftarrow \ell, i \leftarrow i, i' \leftarrow i + 1)$. If it is to the left of our goal bag, i.e., $\ell \leq k^*$, then we can apply our inductive hypothesis to the simpler problem $(k \leftarrow \ell, k^* \leftarrow k^*, k' \leftarrow k', i \leftarrow i + 1, i' \leftarrow i')$. Hence in either case we can find an i^* as desired. This completes the proof. $\qquad\square$
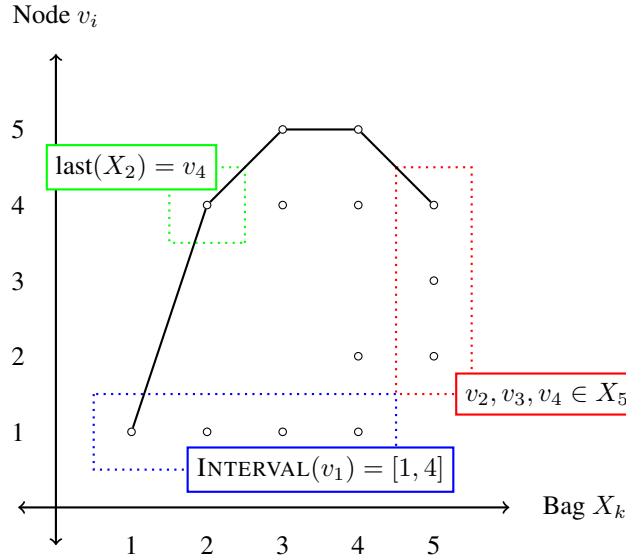

Figure 5: The $\mathrm{last}(\cdot)$ function is single-peaked. Notice that the peak may stretch over multiple bags (in this case, X_3 and X_4).

The next property is easier to understand with a picture; refer to Figure 5. Suppose we plot $\mathrm{last}(X_i)$ on a graph with the x-axis representing bag indices and the y-axis representing node indices. Then the curve traced out by $\mathrm{last}(\cdot)$ will have a single peak; it is nondecreasing before the peak and nonincreasing after. More formally, we have the following lemma.

Lemma 10. *Suppose the final node v_n is in some bag X_c. Then for any k, ℓ such that $c \leq k \leq \ell$, $\mathrm{last}(X_k) \geq \mathrm{last}(X_\ell)$. Additionally, for any k', ℓ' such that $\ell' \leq k' \leq c$, $\mathrm{last}(X_{\ell'}) \leq \mathrm{last}(X_{k'})$*

Proof. We prove just the $c \le k \le \ell$ case (the proof of the $\ell' \le k' \le c$ case is analogous). Let $i = \mathrm{last}(X_\ell)$. By Lemma 9, there is a $i^* \in [i, t]$ such that $v_{i^*} \in X_k$. This node is a lower bound on $\mathrm{last}(X_k)$; we deduce that $\mathrm{last}(X_k) \ge i^* \ge i = \mathrm{last}(X_\ell)$. This completes the proof. $\square$

This final property will be our main tool to bound the complexity of recursing. The main idea it tries to capture is that some particular ways to restrict our graph G into a subgraph G' result in stripping the last node from every bag and hence lower the width of the path decomposition by one.

Lemma 11. *Suppose that the final node v_n is in some bag X_c. Then for any k, ℓ such that $c \le k \le \ell$, the following is true. Let G' be G restricted to nodes with indices in the range $[1, \mathrm{last}(X_\ell) - 1]$ and that appear in bags from X_k to X_ℓ. Let G'' be G' with edges (u, v) added if both u and v are topologically before $\mathrm{last}(X_\ell)$ and either (1) $u, v \in X_k$ or (2) $u, v \in X_\ell$. Note that G'' has a long spine, i.e., a directed path through G'' that touches every node of G''.*

Then G'' has a path decomposition with bag size of $p - 1$. In fact, if we take the original path decomposition for G from X_k to X_ℓ and we remove all nodes not in G'', that is a valid path decomposition for G'' with bag size $p - 1$.

An analagous statement holds for bags on the "left side" ($\ell \le k \le c$).

Proof. In general, given graph H and subgraph H', if we have a path decomposition for H, then when we remove all nodes not in H', that is a valid path decomposition for H': The edge property still holds, and the between-ness property still holds. (We can also remove all empty bags while maintaining the path decomposition properties.)

So let $\mathcal{X}$ be the path decomposition for G, and let $\mathcal{X}''$ be the same decomposition when restricted to nodes in G''. Since every edge we added to G'' must be in either X_k or X_ℓ (even after restricting to nodes in G''), $\mathcal{X}''$ is a valid path decomposition for G''.

So we only need to show that for all $X \in \mathcal{X}''$, the size of X is at most $p - 1$. But X is to the left of X_ℓ, so $\mathrm{last}(X) \ge \mathrm{last}(X_\ell)$. Hence, X restricted to nodes in G' does not contain $\mathrm{last}(X)$. That is, it has size at most $p - 1$. This completes the proof. $\square$

Adding edges to G'' may seem like a strange technical condition, but it is much more natural to think about it in the following way. Given a path decomposition for G, it induces an interval for each node in G, as described in Section A.1. This in turn induces an interval graph: we have an edge between u and v iff the intervals for u and v overlap. This interval graph is necessarily a supergraph of G. Throughout our algorithm, we actually operate on this supergraph. After all, in the worst case, the induced interval graph and G are identical. (The one exception to this is that the actual memory used by the schedule is based on the true indegree of nodes rather than the expanded indegree.)

Note that in an interval graph, these edges that we would add already exist; i.e., $G'' = G'$.

A.3 Key Properties of Interleaved Schedules

Our recursive algorithm will need to take in a subgraph (along with a path decomposition) and return a low-memory schedule. To properly use the schedules returned by our recursive subcalls, we need an understanding of how to interleave them together. We will go over the properties here; the matching proofs appear in Subsubsection A.3.2. Also, note that we will be assuming throughout this subsection that there is a complete topological ordering on the vertices under consideration.

Definition 12. *Suppose we have a set of nodes $W \subseteq V$. A schedule $\sigma = u_1 u_2 \ldots u_t$ is said to be valid for known set W if for every $u_i \in \sigma$, either (1) $u_i \in W$, or each predecessor of u_i either (2a) is in the known set W or (2b) appears earlier in the schedule.*

We use $\circ$ to denote concatenation; for example $\sigma \circ u \circ \sigma'$ means schedule σ followed by u followed by schedule σ'.

When σ is a valid schedule for known set W, we define $M_W(\sigma)$ to be the peak memory taken by the schedule $\sigma' = w_1 \circ w_2 \circ \cdots \circ w_k \circ \sigma$, where we don't consider the memory for any w_i. Formally, suppose $\sigma' = u_1 u_2 \ldots u_{t'}$. As earlier, for $i \le t'$, define

$$U_i = \{u_i\} \cup \mathrm{in}(u_i) \cup \bigcup_{j > i} \{u' \in \mathrm{in}(u_j) \mid \mathrm{prev}(u', j) \le i\}.$$

14

Then $M_W(\sigma) = \max_{i:u_i \notin W} s(U_i)$, where $s(U_i) = \sum_{u \in U_i} s(u)$. (We let $s(u)$ be the size of tensor output for node u, as before.) Notice that if we need to hold some w_i in memory, we still must pay for it.

We define L as the length of the schedule σ, as before, without including the w_i. Notice that $L(\sigma)$ makes sense even if σ is not valid—it is still just the sum of the lengths of the operations in the schedule. So we can extend $L(\sigma)$ even for invalid sequences. We note that when σ does not contain any nodes from W that $L_w(\sigma) = L(\sigma)$ (where σ may be valid for known set W but not valid in general). Because of this, we will only consider $L(\sigma)$, regardless of the known set or validity of σ.

Property 13. *Let W be a set of nodes, x a node, and let $\sigma_1, \cdots, \sigma_k$ be schedules, with each $\sigma_i \circ x$ valid for known set W. Then $\sigma_1 \circ \cdots \circ \sigma_k \circ x$ is a valid schedule for known set W. Further,*

$$M_W(\sigma_1 \circ \cdots \circ \sigma_k \circ x) \leq \sum_i M_W(\sigma_i \circ x)$$

We also have

$$L(\sigma_1 \circ \cdots \circ \sigma_k \circ x) \leq \sum_i L(\sigma_i \circ x)$$

Property 14. *Let X be a set of nodes $x_1 \prec x_2 \prec \cdots x_k$, W a set of nodes, and $\sigma_1, \ldots, \sigma_k$ schedules such that for all i, the schedule $\sigma_i \circ x_i$ is valid for known set $W \cup X_{\prec x_i}$, where $X_{\prec x_i} = \{x \in X \mid x \prec x_i\}$. Then $\sigma_1 \circ x_1 \circ \sigma_2 \circ x_2 \cdots \circ \sigma_k \circ x_k$ is valid for known set W and*

$$M_W(\sigma_1 \circ x_1 \circ \sigma_2 \circ x_2 \cdots \circ \sigma_k \circ x_k) \leq \max_i(M_{W \cup X_{\prec x_i}}(\sigma_i \circ x_i) + s(W \cup X_{\prec x_i}))$$

We also have

$$L(\sigma_1 \circ x_1 \cdots \circ \sigma_k \circ x_k) \leq \sum_i L(\sigma_i \circ x_i).$$

Note that this bound is a little sloppy. For the kth term, we could have simply used $M_{W \cup X_{\prec x_k}}(\sigma_i, x_k)$, avoiding the $s(W \cup X_{\prec x_k})$. It won't matter for our proofs.

We now introduce our key concept: interleaved schedules.

Definition 15. *Let X be a set of nodes. An* interleaved schedule *on X is a set of tuples $\mathcal{I} = \{\langle \sigma_1, x_1 \rangle, \langle \sigma_2, x_2, \rangle, \ldots \langle \sigma_k, x_k \rangle\}$ such that each $x_i \in X$ and each σ_i is a schedule. We further require that for every $x \in X$, there is some tuple $\langle \sigma, x \rangle \in \mathcal{I}$.*

Let $X_{\prec x} = \{x' \in X \mid x' \prec x\}$, and let W be a set of nodes. We say $\mathcal{I}$ is valid for known set W *if for each $\langle \sigma, x \rangle \in \mathcal{I}$, the schedule $\sigma \circ x$ is valid for known set $W \cup X_{\prec x_i}$. If $W = \emptyset$, we say simply that $\mathcal{I}$ is* valid.

Suppose $\mathcal{I}$ in an interleaved schedule on X with $\mathcal{I} = \bigcup_{i,j}\{\langle \sigma_{ij}, x_i \rangle\}$, where each x_i is distinct and $x_1 \prec \cdots \prec x_k$ are the nodes in X. We define

$$M_W(\mathcal{I}) = \max_i \left(\sum_j M_{W \cup X_{\prec x_i}}(\sigma_{ij} \circ u_i) + s(W \cup X_{\prec x_i}\}) \right).$$

We also define

$$L(\mathcal{I}) = \sum_{i,j} L(\sigma_{ij} \circ u_i).$$

In addition, we need a few operations to combine interleaved schedules and convert them to [standard] schedules. Define

$$\mathtt{Merge}(\mathcal{I}) = \bigcup_i \{\langle \sigma_i \circ x_i \rangle\},$$

where $\sigma_i = \sigma_{i1} \circ \sigma_{i2} \circ \cdots$ for each i. Within each i, the σ_{ij} may be ordered arbitrarily. For concreteness, suppose they are ordered lexicographically.

Also define

$$\mathtt{Flatten}(\mathcal{I}) = \sigma_1 \circ x_1 \circ \sigma_2 \circ x_2 \circ \cdots \circ \sigma_k \circ x_k.$$

Let $X' \subseteq X$, and let $i_1 < i_2 < \cdots < i_{k'}$ be indices so that $X' = \{x_{i_j}\}$, where $k' = |X'|$. Let $\tau_j = \sigma_{i_{j-1}+1} \circ x_{i_{j-1}+1} \circ \sigma_{i_{j-1}+2} \circ x_{i_{j-1}+2} \circ \cdots \sigma_{i_j}$, where $i_0 = 0$ for convenience. Define

$$\mathtt{Condense}(X', \mathcal{I}) = \bigcup_j \{\langle \tau_j, x_{i_j} \rangle\}.$$

15

Notice that $\mathtt{Condense}(X',\mathcal{I})$ is an interleaved schedule on X'. We also have that $\mathtt{Condense}(X',\mathcal{I}) = \mathtt{Condense}(X',\mathtt{Merge}(\mathcal{I}))$ and $\mathtt{Flatten}(\mathcal{I}) = \mathtt{Flatten}(\mathtt{Merge}(\mathcal{I})) = \mathtt{Flatten}(\mathtt{Condense}(X',\mathcal{I}))$.

Property 16. *Let W be a set of nodes. Let $\mathcal{I}$ be an interleaved schedule on X and let $\mathcal{I}'$ be an interleaved schedule on X'. Then*

$$M_W(\mathcal{I} \cup \mathcal{I}') \leq M_W(\mathcal{I}) + M_W(\mathcal{I}')$$

and

$$L(\mathcal{I} \cup \mathcal{I}') \leq L(\mathcal{I}) + L(\mathcal{I}').$$

If $\mathcal{I}$ and $\mathcal{I}'$ are both valid for known set W, then $\mathcal{I} \cup \mathcal{I}'$ is valid for known set W.

Property 17. *Let W be a set of nodes. Let $\mathcal{I}$ be an interleaved schedule on X and $\mathcal{I}'$ be an interleaved schedule on X'. Further, suppose $x \prec x'$ for all $x \in X, x' \in X'$. Then we have*

$$M_W(\mathcal{I} \cup \mathcal{I}') = \max\{M_W(\mathcal{I}), M_{W \cup X}(\mathcal{I}')\}.$$

If $\mathcal{I}$ is valid for known set W and $\mathcal{I}'$ is valid for known set $W \cup X$, then $\mathcal{I} \cup \mathcal{I}'$ is valid for known set W.

Property 18. *Let $\mathcal{I}$ be an interleaved schedule on X that is valid for known set W. Then $\mathtt{Merge}(\mathcal{I})$ is an interleaved schedule on X that is valid on known set W. Also, if $X' \subseteq X$, then $\mathtt{Condense}(X',\mathcal{I})$ is an interleaved schedule on X' that is valid on known set W.*

Further, $M_W(\mathtt{Merge}(\mathcal{I})) \leq M_W(\mathcal{I})$ and $L(\mathtt{Merge}(\mathcal{I})) \leq L(\mathcal{I})$. Likewise, $M_W(\mathtt{Condense}(X',\mathcal{I})) \leq M_W(\mathcal{I})$ and $L(\mathtt{Condense}(X',\mathcal{I})) \leq L(\mathcal{I})$.

Property 19. *Let W be a set of nodes. If $\mathcal{I}$ is an interleaved schedule on X that is valid for known set W, then $\mathtt{Flatten}(\mathcal{I})$ is a valid schedule on known set W that computes every node in X. Further,*

$$M_W(\mathtt{Flatten}(\mathcal{I})) \leq M_W(\mathtt{Merge}(\mathcal{I})) \leq M_W(\mathcal{I}),$$
$$L(\mathtt{Flatten}(\mathcal{I})) \leq L(\mathtt{Merge}(\mathcal{I})) \leq L(\mathcal{I}).$$

A.3.1 Combining Interleaved Schedules with Path Decompositions

We need a few lemmas to better characterize interleaved schedules in the special cases we are considering. Let $\nu(\mathcal{X})$ be the set of nodes in $\mathcal{X}$.

Lemma 20. *Let $\mathcal{X} = (X_1, \ldots, X_b)$ be a path decomposition, and for some $i \in [b]$, let $\mathcal{X}' = (X_1, \ldots, X_{i-1})$. Let $\mathcal{I}$ be an interleaved schedule on X, and suppose that $\mathcal{I}$ is valid on known set $W \cup \overline{\nu(\mathcal{X}')}$, where $W \supseteq X_i$. Further, suppose that every node appearing in $\mathcal{I}$ is also in $\nu(\mathcal{X}')$. Then $\mathcal{I}$ is valid on known set $W \cup \overline{\nu(\mathcal{X})}$.*

Proof. Let $\langle \sigma, x \rangle \in \mathcal{I}$. We wish to show $\sigma \circ x$ is valid on known set $W \cup \overline{\nu(\mathcal{X})} \cup X_{\prec x}$. Take some $u \notin W \cup \overline{\nu(\mathcal{X})} \cup X_{\prec x}$ in $\sigma \circ x$, and consider some $v \in \mathrm{in}(u)$. If $v \in W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X})}$, then we are done. So suppose not. Then v must appear in $\mathcal{X}$ but not in $X_{\prec x}$. (Since $v \prec u \prec x$, we see that x cannot appear in X either.)

If $v \in \nu(\mathcal{X}')$, then we are done, since $\sigma \circ x$ is valid on known set $W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X}')}$. So suppose $v \notin \nu(\mathcal{X}')$ (but still $v \in \nu(\mathcal{X})$). There is an edge from v to u, so u and v must appear in a bag together. Since $u \in \nu(\mathcal{X}')$ but $v \notin \nu(\mathcal{X}')$, either u or v must appear in $X_i \subseteq W$ by the betweenness property of path decompositions. That is a contradiction. So $\sigma \circ x$ is valid on known set $W \cup \overline{\nu(\mathcal{X})} \cup X_{\prec x}$.

The claim follows. $\qquad\square$

Lemma 21. *Let $\mathcal{X} = (X_1, \ldots, X_b)$ be a path decomposition, and for some $i \in [b]$, let $X = X_i$, let $\mathcal{X}^\ell = (X_1, \ldots, X_{i-1})$, and let $\mathcal{X}^r = (X_{i+1}, \ldots, X_b)$. Let $\mathcal{I}^\ell$ be an interleaved schedule on X^ℓ, and let $\mathcal{I}^r$ be an interleaved schedule on X^r, and suppose $X^\ell \cap X^r = X$. Suppose that $\mathcal{I}^\ell$ is valid on known set $W \cup \overline{\nu(\mathcal{X}^\ell)}$ and $\mathcal{I}^r$ is valid on known set $W \cup \overline{\nu(\mathcal{X}^r)}$. Then $\mathtt{Merge}(\mathcal{I}^\ell \cup \mathcal{I}^r)$ is valid on known set $W \cup \overline{\nu(\mathcal{X})}$.*

Proof. Choose $x \in X$, and let $\langle \sigma^\ell, x \rangle \in \mathtt{Merge}(\mathcal{I}^\ell)$ and $\langle \sigma^r, x \rangle \in \mathtt{Merge}(\mathcal{I}^r)$. We wish to show $\sigma^\ell \circ \sigma^r \circ x$ is valid on known set $W \cup \overline{\nu(\mathcal{X})} \cup X_{\prec x}$.

16

Take some u in $\sigma \circ x$. If $u \in X$, without loss of generality, we can assume $u = x$. Consider $v \in \text{in}(x)$. If $v \in W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X})}$, we are done. So suppose not. Then $v \notin X_{\prec x}$, which means $v \notin X$ since $v \prec x$. Further, $v \in \nu(\mathcal{X})$, which means either $v \in \nu(\mathcal{X}^\ell)$ or $v \in \nu(\mathcal{X}^r)$. Without loss of generality, suppose $v \in \nu(\mathcal{X}^\ell)$. Since $\sigma^\ell \circ x$ is valid on known set $W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X}')}$, it must be the case that v appears in σ^ℓ. Hence, all vertices of $\text{in}(x)$ are either known or appear in the schedule before x.

Now, suppose $u \notin X$. Without loss of generality, say u appears in σ^ℓ. Consider some $v \in \text{in}(u)$. If v appears in $W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X})}$, then we are done. So suppose not. Then v must appear in $\nu(\mathcal{X})$, but not in $X_{\prec x}$ (hence, not in X since $v \prec x$). If $v \in \nu(\mathcal{X}^\ell)$, then, since $\sigma^\ell \circ x$ is valid for known set $W \cup X_{\prec x} \cup \overline{\nu(\mathcal{X}^\ell)}$, we're done. Otherwise, $v \in \nu(\mathcal{X}^r)$. But $v \in \text{in}(u)$, meaning that most u and v must appear in a bag together. By the betweenness property of path decompositions, that means either u or v must be in X. But that's a contradiction. Hence, all vertices of $\text{in}(u)$ are either known or appear in the schedule before u.

Hence, $\sigma^\ell \circ \sigma^r \circ x$ is valid on known set $W \cup \overline{\nu(\mathcal{X})}$, and the claim follows. $\qquad\square$

A.3.2 Proofs for the Key Properties

Proof of Property 13. Let $\sigma' = \sigma_1 \circ \cdots \circ \sigma_k \circ x$. It is an immediate consequence of the definition that σ' is valid on known set W. Consider some $u_i \notin W$ appearing in σ' where the peak memory occurs. Either u_i appears in σ_ℓ for some ℓ, or u_i is the last element in σ' (i.e., node x). First, take the case that u_i appears in σ_ℓ for some ℓ. Recall the definition

$$U_i = \{u_i\} \cup \text{in}(u_i) \cup \bigcup_{j > i} \{u' \in \text{in}(u_j) \mid \text{prev}(u', j) \leq i\},$$

and peak memory is $s(U_i)$. Let $U = \{u' \in \text{in}(u_j) \mid \text{prev}(u', j) \leq i\}$, and let $U' = \{u' \in \text{in}(u_j) \mid \text{prev}(u', j) \leq i$ such that u_j appears in a sequence after $\sigma_\ell\}$. Notice that $M_W(\sigma_\ell \circ x) \geq s(U_i \setminus U')$, so we only need to bound $s(U')$.

For each $u \in U'$, there is some u_j such that u_j appears in $\sigma_{\ell'}$ with $\ell' > \ell$ and $u' \in \text{in}(u_j)$ with $\text{prev}(u', j) \leq i$. Let $U^{\ell'}$ be the set of all such u. Then $s(U^{\ell'}) \leq M_W(\sigma_{\ell'} \circ x)$. We have

$$s(U_i) = s(U_i \setminus U') + \sum_{\ell' > \ell} s(U^{\ell'}) \leq M_W(\sigma_\ell \circ x) + \sum_{\ell' > \ell} M_W(\sigma_{\ell'} \circ x).$$

The claim follows for this case.

In the special case that u_i is the last element in σ', i.e., node x, say $x = u_{t'}$. Since x appears last, our definition simplifies somewhat. We have $U_{t'} = \{x\} \cup \text{in}(x)$. Every node $u \in U_{t'}$ (for $u \notin x$) belongs to $\nu(\sigma_\ell)$ for some ℓ. Let U^ℓ be the set of such u. We have

$$s(U_{t'}) = 1 + \sum_\ell s(U^\ell) \leq \sum_\ell M_W(\sigma_\ell \circ x).$$

Our bound on M_W follows.

To bound time, we simply note that every element appearing on the left-hand side of the inequality also appears on the right-hand side (with x appearing multiple times). $\qquad\square$

Proof of Property 14. We induct on k. The case for $k = 1$ is trivially true. Consider general k. Let $\sigma = \sigma_1 \circ x_1 \cdots \circ \sigma_{k-1} \circ x_{k-1}$.

By induction, σ is valid for known set $W \cup X_{\prec x_{k-1}}$. Since $\sigma_k \circ x_k$ is valid for known set $W \cup X_{\prec x_k}$, and all values of $X_{\prec x_k}$ appear in the schedule earlier than σ_k, we see $\sigma_1 \circ x_1 \cdots \circ \sigma_k \circ x_k$ is valid for known set W.

We now bound M_W. To do so, consider some segment of σ, say from $\sigma_\ell \circ x_\ell$. The set of nodes that must be held in memory for some future segment is at most $W \cup X_{\prec x_\ell}$. So if peak memory occurs during this segment, it is bounded by $M_{W \cup X_{\prec x_\ell}}(\sigma_\ell \circ x_\ell) + s(W \cup X_{\prec x_\ell})$. Since peak memory occurs in one of these segments, we have

$$M_W(\sigma_1 \circ x_1 \circ \sigma_2 \circ x_2 \cdots \circ \sigma_k \circ x_k) \leq \max_i (M_{W \cup X_{\prec x_i}}(\sigma_i \circ x_i) + s(W \cup X_{\prec x_i}))$$

which is what we wanted.

Finally, the bound on time follows easily: every element on the left-hand side appears on the right-hand side. □

Proof of Property 16. Let $Z = X \cup X'$. Note that $\mathcal{I} \cup \mathcal{I}'$ is an interleaved schedule on Z. Also note that $Z_{\prec x} \supseteq X_{\prec x}$ and $Z_{\prec x} \supseteq X'_{\prec x}$ for any x.

We first show $\mathcal{I} \cup \mathcal{I}'$ is valid on known set W. For any $\langle \sigma, x \rangle \in \mathcal{I}$, we know $\sigma \circ x$ is valid for known set $W \cup X_{\prec x}$, hence it is also valid for known set $W \cup Z_{\prec x}$. Similarly, for any $\langle \sigma', x' \rangle \in \mathcal{I}'$, we see that $\sigma' \circ x'$ is a valid schedule for known set $W \cup Z_{\prec x}$. Hence, $\mathcal{I} \cup \mathcal{I}'$ is a valid interleaved schedule on known set W.

To see the memory bound, we have

$$
\begin{aligned}
M_W(\mathcal{I} \cup \mathcal{I}') &= \max_{x \in X \cup X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I} \cup \mathcal{I}'} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup Z_{\prec x}) \Big) \\
&\leq \max_{x \in X \cup X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup X_{\prec x}) \\
&\qquad + \sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}'} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup X'_{\prec x}) \Big) \\
&\leq \max_{x \in X} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup X_{\prec x}) \Big) \\
&\qquad + \max_{x \in X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}'} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup X'_{\prec x}) \Big) \\
&\leq \max_{x \in X} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}} M_{W \cup X_{\prec x}}(\sigma \circ x) + s(W \cup X_{\prec x}) \Big) \\
&\qquad + \max_{x \in X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}'} M_{W \cup X'_{\prec x}}(\sigma \circ x) + s(W \cup X'_{\prec x}) \Big) \\
&= M_W(\mathcal{I}) + M_W(\mathcal{I}').
\end{aligned}
$$

To bound the time, we have

$$
\begin{aligned}
L(\mathcal{I} \cup \mathcal{I}') &= \sum_{\langle \sigma, x \rangle \in \mathcal{I} \cup \mathcal{I}'} L(\sigma \circ x) \\
&\leq \sum_{\langle \sigma, x \rangle \in \mathcal{I}} L(\sigma \circ x) + \sum_{\langle \sigma, x \rangle \in \mathcal{I}'} L(\sigma \circ x) \\
&= L(\mathcal{I}) + L(\mathcal{I}').
\end{aligned}
$$

This completes the proof. □

Proof of Property 17. We first show $\mathcal{I} \cup \mathcal{I}'$ is valid for known set W. Let $Z = X \cup X'$, and note that $\mathcal{I} \cup \mathcal{I}'$ is an interleaved schedule on Z.

Let $\langle \sigma, x \rangle \in \mathcal{I} \cup \mathcal{I}'$. If $\langle \sigma, x \rangle \in \mathcal{I}$, then (since $\mathcal{I}$ is valid), we have $\sigma \circ x$ is valid on known set $W \cup X_{\prec x} \subseteq W \cup Z_{\prec x}$, which is what we wanted. On the other hand, if $\langle \sigma, x \rangle \in \mathcal{I}'$, then $\sigma \circ x$ is valid on known set $W \cup X \cup X'_{\prec x} \subseteq W \cup Z_{\prec x}$ since x is topologically larger than all nodes in X. Hence, $\mathcal{I} \cup \mathcal{I}'$ is valid for known set W.

To bound the memory, first note that for $x \in X$, we have $Z_{\prec x} = X_{\prec x}$, and for $x \in X'$, we have $Z_{\prec x} = X \cup X'_{\prec x}$. Thus, we have

$$
\begin{aligned}
M_w(\mathcal{I} \cup \mathcal{I}') &= \max_{x \in X \cup X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I} \cup \mathcal{I}'} M_{W \cup Z_{\prec x}}(\sigma \circ x) + s(W \cup Z_{\prec x}) \Big) \\
&= \max \Big\{ \max_{x \in X} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}} M_{W \cup X_{\prec x}}(\sigma \circ x) + s(W \cup X_{\prec x}) \Big), \\
&\qquad\quad \max_{x \in X'} \Big(\sum_{\sigma : \langle \sigma, x \rangle \in \mathcal{I}'} M_{W \cup X \cup X'_{\prec x}}(\sigma \circ x) + s(W \cup X \cup X'_{\prec x}) \Big) \Big\} \\
&= \max \{ M_W(\mathcal{I}), M_{W \cup X}(\mathcal{I}') \}.
\end{aligned}
$$

This completes the proof. $\qquad\square$

Proof of Property 18. If $\sigma \circ x$ is valid on known set W' and $\sigma' \circ x$ is valid on known set W', then clearly $\sigma \circ \sigma' \circ x$ is valid on known set W'. The proof for `Merge` follows from this observation and Property 13.

The reasoning for `Condense` is the same as in Property 14, and the proof follows from the definition of M_W and L. $\qquad\square$

Proof of Property 19. We first show $\texttt{Flatten}(\mathcal{I})$ is a valid schedule. Let u be some element in this schedule, and consider some $v \in \text{in}(u)$. Notice that u corresponds to some element in $\mathcal{I}$, say it appears in σ_ℓ for some $\langle \sigma_\ell, x \rangle \in \mathcal{I}$. We know $\sigma_\ell \circ x$ is valid for known set $W \cup X_{\prec x}$. Hence, either $v \in W \cup X_{\prec x}$ or it appears in σ_ℓ earlier than u. The only potential problem is when v does not appear earlier in σ_ℓ and $v \notin W$. In this case, $v \in X_{\prec x}$. But we know all items in $X_{\prec u} = X_{\prec x}$ have been produced earlier in $\texttt{Flatten}(\mathcal{I})$, so the schedule is valid.

To prove the memory bound, note that $\texttt{Flatten}(\mathcal{I}) = \texttt{Flatten}(\texttt{Merge}(\mathcal{I}))$, and by Property 13, $M_W(\texttt{Merge}(\mathcal{I})) \leq M_W(\mathcal{I})$. So without loss of generality, we will assume that for each $x \in X$, there is exactly one σ such that $\langle \sigma, x \rangle \in \mathcal{I}$.

But in this case, Property 14 shows directly that $M_W(\texttt{Flatten}(\mathcal{I})) \leq M_W(\mathcal{I})$.

The time bound also follows directly from Property 13 and Property 14. $\qquad\square$

A.4 Rematerialization Algorithm with Path Decomposition

In this section we present and analyze a rematerialization algorithm that works on path decompositions.

We present our main theorem.

Theorem 22. *Let $\mathcal{X} = \{X_1, \ldots, X_b\}$ be a path decomposition for a graph $G = (V, E)$, where G is a directed acyclic graph with a directed path of length $n = |V|$ and whose max indegree is bounded by a constant. Let u be the last node topologically in G, and let $\sigma = \texttt{PWRemat}(\mathcal{X}, u, \emptyset)$, as described in Algorithm 3. Then σ is a valid schedule to compute u.*

Furthermore, if the maximum bag size in $\mathcal{X}$, i.e., $\max_i \{|X_i|\}$, is bounded by p, then $L(\sigma) = O(2^p (p!)^2 b \log^{p-2} b)$ and $M(\sigma) = O((M_{\text{in}} + M_{\text{max}}) 2^p p \log^{p-2} b)$.

We break the proof into two parts. The first shows correctness. The second bounds the time and space.

Proof of correctness. We prove two claims simultaneously using induction:

1. $\texttt{PWRemat}(\mathcal{X}, u, W)$ produces a schedule σ that is valid schedule for known set $W \cup \overline{\nu(\mathcal{X})}$.

2. $\texttt{InterleavedSchedule}(X, \mathcal{X}, W)$ produces an interleaved schedule $\mathcal{I}$ on X that is valid for known set $W \cup \overline{\nu(\mathcal{X})}$.

Algorithm 3: More Efficient Rematerialization via Path Decomposition.

Function: `PWRemat(`$\mathcal{X}$`,` u`,` W`)`:

Data: $\mathcal{X} = (X_1, X_2, \ldots, X_b)$ a path decomposition, u a node to compute, $W \subseteq V$ a subset of nodes whose value is known.

Result: Returns a rematerialization schedule.

if $u \in W \cup \overline{\nu(\mathcal{X})}$, *or every input to u appearing in $\nu(\mathcal{X})$ is in W* **then**
  return *the schedule containing only u*

Remove any nodes in $\mathcal{X}$ that are after u. Call this new decomposition $\mathcal{X}' = (X_1', \ldots, X_b')$;

if $\mathcal{X}'$ *has pathwidth 1* **then**
  return *a valid schedule for u for known set W on nodes in $\nu(\mathcal{X}')$ with no rematerialization*

Find i such that $u \in X_i$;

Compute $\mathcal{I}^\ell \leftarrow \texttt{InterleavedSchedule}(X_i', \mathcal{X}^\ell, W)$ where $\mathcal{X}^\ell = (X_{i-1}', X_{i-2}', \ldots, X_1')$;

Compute $\mathcal{I}^r \leftarrow \texttt{InterleavedSchedule}(X_i', \mathcal{X}^r, W)$ where $\mathcal{X}^r = (X_{i+1}', X_{i+2}', \ldots, X_b')$;

return `Flatten(`$\mathcal{I}^\ell \cup \mathcal{I}^r$`)`;

Function: `InterleavedSchedule(`X`,` $\mathcal{X}$`,` W`)`:

Data: $X = \{x_1, x_2, \ldots, x_k\}$ a set of nodes, $\mathcal{X} = (X_1, X_2, \ldots, X_b)$ a path decomposition, $W \subseteq V$ a subset of nodes whose value is known.

Result: Returns an interleaved schedule for X, i.e., a set of tuples $\langle \sigma, x \rangle$, where $x \in X$ and σ is a schedule.

if $X \subseteq W$ **then**
  return *an empty interleaved schedule on X, i.e.* $\{\langle \emptyset, x \rangle \mid x \in X\}$;

Let $\mathcal{X}_{inner} = (X_1, X_2, \ldots, X_{b/2-1})$ and $\mathcal{X}_{outer} = (X_{b/2+1}, X_{b/2+2} \ldots, X_b)$;

Initialize $\mathcal{I}_{inner} \leftarrow \emptyset$ and $W' \leftarrow W$, and let $X' = \{x \in X \mid x \prec last(X_{b/2})\}$;

Let $u_1 \prec u_2 \prec \cdots \prec u_j$ be the nodes in $X' \cup X_{b/2}$;

for $i = 1$ *to* k **do**
  Assign $\sigma_i \leftarrow \texttt{PWRemat}(\mathcal{X}_{inner}, u, \{u_1, u_2, \ldots, u_{i-1}\})$ where u is the node before u_i in the spine for G;
  Add $\langle \sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \sigma_i, u_i \rangle$ to $\mathcal{I}_{inner}$;

Compute $\mathcal{I}_{outer} \leftarrow \texttt{InterleavedSchedule}(X_{b/2}, \mathcal{X}_{outer}, W)$;

Compute $\mathcal{I}_{lower} \leftarrow \texttt{InterleavedSchedule}(X \setminus X', \mathcal{X}_{inner}, W \cup X' \cup X_{b/2})$;

return `Condense(`$X, \mathcal{I}_{inner} \cup \mathcal{I}_{outer} \cup \mathcal{I}_{lower}$`)`;

Function: `Flatten(`$\mathcal{I}$`)`:

Data: An interleaved schedule, $\mathcal{I} = \bigcup_{i,j} \langle \sigma_{ij}, x_i \rangle$ with $x_1 \prec x_2 \prec \cdots$.

Result: Returns a schedule.

return $\sigma_1 \circ x_1 \circ \sigma_2 \circ x_2 \circ \cdots$, *where each* $\sigma_i = \sigma_{i1} \circ \sigma_{i2} \circ \sigma_{i3} \circ \cdots$;

Function: `Condense` *(*$X, \mathcal{I}$*)*:

Data: An interleaved schedule, $\mathcal{I} = \bigcup_{i,j} \langle \sigma_{ij}, x_i \rangle$ with $x_1 \prec x_2 \prec \ldots$.

Result: Returns an interleaved schedule.

return *an interleaved schedule $\mathcal{I}'$ on X such that* `Flatten(`$\mathcal{I}$`) = Flatten(`$\mathcal{I}'$`)`, *as defined in text.*

For the base case in Claim 1, we produce the schedule containing only u. Either $u \in W \cup \overline{\nu(\mathcal{X})}$ or all of its inputs are in $W \cup \overline{\nu(\mathcal{X})}$. In either case, u (viewed as a schedule) is valid for known set $W \cup \overline{\nu(\mathcal{X})}$. In the case the $\mathcal{X}'$ has pathwidth 1, we again return a valid schedule for known set $W \cup \overline{\nu(\mathcal{X})}$.

For the base case in Claim 2, we produce an empty interleaved schedule on $X \subseteq W$, which is valid on known set W.

More generally, $\texttt{PWRemat}(\mathcal{X}, u, W)$ finds schedules $\mathcal{I}^\ell = \texttt{InterleavedSchedule}(X, \mathcal{X}^\ell, W)$ and $\mathcal{I}^r = \texttt{InterleavedSchedule}(X, \mathcal{X}^r, W)$ and returns $\texttt{Flatten}(\mathcal{I}^\ell \cup \mathcal{I}^r)$. By induction, $\mathcal{I}^\ell$ is

valid on known set $W \cup \overline{\nu(\mathcal{X}^\ell)}$ and $\mathcal{I}^r$ is valid on known set $W \cup \overline{\nu(\mathcal{X}^r)}$. Hence, by Lemma 21, $\mathtt{Merge}(\mathcal{I}^\ell \cup \mathcal{I}^r)$ is valid on known set $W \cup \overline{\nu(\mathcal{X})}$. Hence, by Property 19, $\mathtt{Flatten}(\mathcal{I}^\ell \cup \mathcal{I}^r) = \mathtt{Flatten}(\mathtt{Merge}(\mathcal{I}^\ell \cup \mathcal{I}^r))$ is valid on known set $W \cup \overline{\nu(\mathcal{X})}$, as we wanted.

Continuing, $\mathtt{InterleavedSchedule}(X, \mathcal{X}, W)$ produces an interleaved schedule on X consisting of the union of $\mathcal{I}_{inner}, \mathcal{I}_{outer}$, and $\mathcal{I}_{lower}$. We wish to show this union is an interleaved schedule that is valid on known set $W \cup \overline{\nu(\mathcal{X})}$.

We first consider $\mathcal{I}_{inner}$. We claim that after each iteration of the for loop, $\mathcal{I}_{inner}$ is valid on known set $\overline{\nu(\mathcal{X}_{inner})}$. To see this, consider tuple $\langle \sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i \circ u_i \rangle$ added to $\mathcal{I}_{inner}$ on the ith iteration of the loop. In the base case, $i = 1$, we see that σ_1 is valid for known set $\overline{\nu(\mathcal{X}_{inner})}$ since it came from a call to $\mathtt{PWRemat}(\mathcal{X}_{inner}, u, \emptyset)$. To see that adding u_1 does not change the validity, let $v \in \mathrm{in}(u_1)$. If $v \notin \overline{\nu(\mathcal{X}_{inner})}$, then consider the spine from v to u. That is, there are nodes $v = v_k, v_{k+1}, \ldots, v_\ell = u$ for which each (v_j, v_{j+1}) is an edge along the long spine in G. All of these nodes (other than u itself) are ancestors of u, hence either all must appear in the valid schedule, or at least one must be known. (We assumed $v \notin \overline{\nu(\mathcal{X}_{inner})}$, so it cannot be known.) But say some ancestor of u, say u', is known. Then $u' \in \overline{\nu(\mathcal{X}_{inner})}$, meaning it appears in a bag outside of $\mathcal{X}_{inner}$. By Lemma 9, this means there is some v' on the path from v to u' such that v is in X' or $X_{b/2}$. But that is a contradiction, since $v' \prec u$ and u is the first node in $X \cup X_{b/2}$. Hence, every $v \in \mathrm{in}(u_1)$ appears in σ_1, meaning that $\sigma_1 \circ u_1$ is valid.

Continuing, again we wish to show the schedule $\sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i \circ u_i$ is valid, for general i. By induction, it is not hard to see $\sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i$ is valid. So we just need to argue that concatenating u_i does not change validity. The proof is analogous to the base case. Let $v \in \mathrm{in}(u_i)$ but $v \notin \overline{\nu(\mathcal{X}_{inner})} \cup \{u_1, u_2, \ldots, u_{i-1}\}$, and let j be as small as possible so that $v \prec u_j$. We claim the part of the spine from v to u_j is contained in σ_j. To see this, all these nodes (other than u_j itself) are ancestors of u_j, hence either must all appear in σ_j, or at least one must be known. Again, we assumed v is not known. But suppose there is some other ancestor of u, say u', on the spine is known. The $u' \in \overline{\nu(\mathcal{X}_{inner})} \cup \{u_1, \ldots, u_{i-1}\}$, meaning it appears in a bag outside of $\mathcal{X}_{inner}$; else, we did not choose the first u_j. By Lemma 9, this means there is some v' on the path from v to u' such that v is in X' or $X_{b/2}$. But that is a contradiction, since $v' \prec u_j$ and u_j is the first node in $X \cup X_{b/2}$ such that v precedes it. Hence, every $v \in \mathrm{in}(u_i)$ appears in σ_j, meaning that $\sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i \circ u_i$ is valid.

Putting this together, we see that the ith tuple added to $\mathcal{I}_{inner}$ is valid on known set $\overline{\nu(\mathcal{X}_{inner})} \cup \{x_1, \ldots, x_{i-1}\}$. By Property 17, adding this tuple to $\mathcal{I}_{inner}$ produces a new interleaved schedule that is also valid on $\overline{\nu(\mathcal{X}_{inner})}$, hence valid on $W \cup \overline{\nu(\mathcal{X}_{inner})}$, as we wanted.

Next, $\mathcal{I}_{outer}$ is valid on known set $W \cup \overline{\nu(\mathcal{X}_{outer})}$ by induction. So we apply Lemma 21 to see that $\mathtt{Merge}(\mathcal{I}_{inner} \cup \mathcal{I}_{outer})$ is valid on set $W \cup \overline{\nu(\mathcal{X})}$, as we wanted.

Finally, $\mathcal{I}_{lower}$ is valid on known set $W \cup X' \cup X_{b/2} \cup \overline{\nu(\mathcal{X}_{inner})}$ by induction. By Lemma 20, we have $\mathcal{I}_{lower}$ is valid on known set $W \cup X' \cup X_{b/2} \cup \overline{\nu(\mathcal{X})}$. Further, $\mathcal{I}_{lower}$ is an interleaved schedule on $X \setminus X'$, while $\mathtt{Merge}(\mathcal{I}_{inner}, \mathcal{I}_{outer})$ is an interleaved schedule on $X' \cup X_{b/2}$. So by Property 17, $\mathtt{Merge}(\mathcal{I}_{inner}, \mathcal{I}_{outer}) \cup \mathcal{I}_{lower}$ is valid on $W \cup \overline{\nu(\mathcal{X}_{inner})}$. The call to $\mathtt{Condense}$ ensures that we return an interleaved schedule on X; notice that $\mathtt{Condense}(\mathtt{Merge}(\mathcal{I}_{inner}, \mathcal{I}_{outer}) \cup \mathcal{I}_{lower}) = \mathtt{Condense}(\mathcal{I}_{inner} \cup \mathcal{I}_{outer} \cup \mathcal{I}_{lower})$, so we may omit the $\mathtt{Merge}$.

This shows by induction that our two claims are true.

Given that (and in particular, the fact that the first claim is true), we see that when $\mathcal{X}$ is the path decomposition for G, $\mathtt{PWRemat}(\mathcal{X}, u, \emptyset)$ produces a schedule σ that is valid for known set $\emptyset \cup \overline{\nu(\mathcal{X})}$. But $\overline{\nu(\mathcal{X})} = \emptyset$, so σ is a valid schedule to compute u. $\qquad\square$

Before finishing the proof, we will need the following lemma.

Lemma 23. *Consider Algorithm 3. Whenever* $\mathtt{InterleavedSchedule}()$ *is called with a path decomposition of pathwidth k, the call to* $\mathtt{PWRemat}()$ *operates on a path decomposition of pathwidth at most $k - 1$.*

Proof. This is a corollary of Lemma 11. Notice that it is clearer to think of operating on the induced interval graphs rather than a subgraph of G. $\square$

Proof of time and space for Theorem 22. Let's now bound the time and memory taken by the schedule.

We first bound the maximum size of W, the known set, on any call. Notice that W grows by at most $2p$ on any call, and the total recursive depth is $\log b$. Since we call PWRemat with $W = \emptyset$ initially, we see that it grows to size at most $2p \log b$. So $s(W) \leq 2pM_{\max} \log b$ over all calls.

Next, a bit of notation. Let $T_p(b)$ be the time for a schedule $\sigma \circ u$ produced by a call to PWRemat$(\mathcal{X}, u, W)$ with path decomposition of length b and maximum bag size p, and let $S_p(b)$ be its memory, for any W such that $s(W) \leq 2pM_{\max} \log b$. In particular, if PWRemat is called with known set W (bounded as above) and $\mathcal{X}$ with maximum bag size of p and it returns σ, then $S_p(b) = M_W(\sigma)$. Likewise, $T_p(b) = L(\sigma)$.

Similarly, let $T_p^{\mathcal{I}}(b)$ be the time for an interleaved schedule produced by InterleavedSchedule with path decomposition of length b and maximum bag size p, and let $S_p^{\mathcal{I}}(b)$ be its memory, for any W such that $s(W) \leq 2pM_{\max} \log b$. Note that if $\mathcal{I}$ is an interleaved schedule returned by InterleavedSchedule with bounded known set W and maximum bag size p, we have $S_p^{\mathcal{I}}(b) = M_W(\mathcal{I})$.

Throughout, we will assume that $T_p(b)$ is superlinear (otherwise, our schedule is linear in length). Hence, $T_p(b_1) + T_p(b_2) \leq T_p(b_1 + b_2)$.

Looking at the call to PWRemat, we have

$$\begin{aligned} S_p(b) &= m_W(\mathcal{I}^\ell \cup \mathcal{I}^r) \\ &\leq m_W(\mathcal{I}^\ell) + m_W(\mathcal{I}^r) \\ &\leq 2S_p^{\mathcal{I}}(b). \end{aligned}$$

Similarly,

$$T_p(b) \leq T_p^{\mathcal{I}}(b_\ell) + T_p^{\mathcal{I}}(b_r) \leq T_p^{\mathcal{I}}(b) \text{ by superlinearity.}$$

Here, b_ℓ and b_r are the sizes of the left and right path decompositions.

Looking at the call to InterleavedSchedule, we first bound $M_W(\mathcal{I}_{inner})$. On the ith iteration of the loop creating $\mathcal{I}_{inner}$, we added the tuple $\langle \sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i \circ u_i \rangle$. We have that each σ_j is valid for known set $\{u_1, \ldots, u_{j-1}\}$, hence $\sigma_j \circ u_j$ is valid for known set $\{u_1, \ldots, u_{j-1}\} \cup \text{in}(u_j)$. Let $W = \bigcup_j \text{in}(u_j)$, and let $U = \{u_1, \ldots, u_k\}$. By Property 14, we can thus bound

$$\begin{aligned} M_W(\sigma_1 \circ u_1 \circ \sigma_2 \circ u_2 \circ \cdots \circ \sigma_i \circ u_i) &\leq \max_j \{M_{W \cup U_{\prec u_j}}(\sigma_j \circ u_j) + s(W \cup U_{\prec u_j})\} \\ &\leq \max_j \{M_{U_{\prec u_j}}(\sigma_j) + s(\text{in}(u_j)) + s(W \cup U_{\prec u_j})\} \\ &\leq S_{p-1}(b/2) + 2pM_{\max} + (2p+1)M_{\text{in}}, \end{aligned}$$

where the relation $M_{U_{\prec u_j}}(\sigma_j) \leq S_{p-1}(b/2)$ follows from Lemma 11.

As we grew $\mathcal{I}_{inner}$ from $\emptyset$ to its final value, we repeatedly invoked Property 17. Each time we added some $\langle \sigma, u \rangle$ to the schedule; from above, $M_W(\sigma \circ u) \leq S_{p-1}(b/2) + 2pM_{\max} + (2p+1)M_{\text{in}}$. So we can bound the value of $M_W(\mathcal{I}_{inner})$ by $S_{p-1}(b/2) + 2pM_{\max} + (2p+1)M_{\text{in}} \leq S_{p-1}(b) + 3p(M_{\max} + M_{\text{in}})$.

Letting $W' = W \cup X' \cup X_{b/2}$, we have

$$\begin{aligned} S_p^{\mathcal{I}}(b) &= m_W(\mathcal{I}_{inner} \cup \mathcal{I}_{outer} \cup \mathcal{I}_{lower}) \\ &\leq \max\{M_W(\mathcal{I}_{inner}) + M_W(\mathcal{I}_{outer}), M_{W'}(\mathcal{I}_{lower})\} \\ &\leq \max\{S_{p-1}(b) + 3p(M_{\max} + M_{\text{in}}) + S_p^{\mathcal{I}}(b/2), S_p^{\mathcal{I}}(b/2)\} \\ &= S_p^{\mathcal{I}}(b/2) + S_{p-1}(b) + 3p(M_{\max} + M_{\text{in}}). \end{aligned}$$

Time is simpler. Note that $\mathcal{I}_{inner}$ consists of at most $2p$ tuples created from calls to PWRemat, where each tuple consists of at most $2p$ copies of schedules returned from PWRemat. So $L(\mathcal{I}_{inner}) \leq$

$(2p)^2 L_{p-1}(b/2)$. We have

$$\begin{aligned}
T_p^{\mathcal{I}}(b) &= L(\mathcal{I}_{inner} \cup \mathcal{I}_{outer} \cup \mathcal{I}_{lower}) \\
&\leq L(\mathcal{I}_{inner}) + L(\mathcal{I}_{outer}) + L(\mathcal{I}_{lower}) \\
&\leq (2p)^2 T_{p-1}(b/2) + T_p^{\mathcal{I}}(b/2) + T_p^{\mathcal{I}}(b/2) \\
&\leq 2T_p^{\mathcal{I}}(b/2) + 2p^2 T_{p-1}(b).
\end{aligned}$$

Combining these inequalities, we have

$$\begin{aligned}
S_p^{\mathcal{I}}(b) &\leq S_p^{\mathcal{I}}(b/2) + 2S_{p-1}^{\mathcal{I}}(b) + 3p(M_{\max} + M_{\text{in}}) \\
T_p^{\mathcal{I}}(b) &\leq 2T_p^{\mathcal{I}}(b/2) + 2p^2 T_{p-1}^{\mathcal{I}}(b).
\end{aligned}$$

Repeatedly replacing the first term on the right-hand side for each, we see

$$\begin{aligned}
S_p^{\mathcal{I}}(b) &\leq S_p^{\mathcal{I}}(b/2^k) + 2kS_{p-1}^{\mathcal{I}}(b) + 3pk(M_{\max} + M_{\text{in}}) \\
T_p^{\mathcal{I}}(b) &\leq 2^k T_p^{\mathcal{I}}(b/2^k) + 2p^2(T_{p-1}^{\mathcal{I}}(b) + 2T_{p-1}^{\mathcal{I}}(b/2) + \ldots 2^k T_{p-1}^{\mathcal{I}}(b/2^k)) \\
&\leq 2^k T_p^{\mathcal{I}}(b/2^k) + 2kp^2 T_{p-1}^{\mathcal{I}}(b).
\end{aligned}$$

Hence,

$$\begin{aligned}
S_p^{\mathcal{I}}(b) &\leq S_p^{\mathcal{I}}(1) + 2(\log b)S_{p-1}^{\mathcal{I}}(b) + 3p(\log b)(M_{\max} + M_{\text{in}}) \\
&\leq 2(\log b)S_{p-1}^{\mathcal{I}}(b) + 4p(\log b)(M_{\max} + M_{\text{in}}) \\
T_p^{\mathcal{I}}(b) &\leq bT_p^{\mathcal{I}}(1) + 2(\log b)p^2 T_{p-1}^{\mathcal{I}}(b) \\
&\leq 2(\log b)p^2 T_{p-1}^{\mathcal{I}}(b) + b.
\end{aligned}$$

Unrolling each, we see

$$\begin{aligned}
S_p^{\mathcal{I}}(b) &\leq 2^{p-2}(\log^{p-2} b)S_2^{\mathcal{I}}(b) + 4p(\log b)(M_{\max} + M_{\text{in}})(1 + 2\log b + \ldots 2^{p-3}\log^{p-3} b) \\
&\leq 2^{p-2}(\log^{p-2} b)S_2^{\mathcal{I}}(b) + 2^p p(\log^{p-2} b)(M_{\max} + M_{\text{in}}) \\
T_p^{\mathcal{I}}(b) &\leq 2^{p-2}(\log^{p-2} b)(p!)^2 T_{p-1}^{\mathcal{I}}(b) + b(1 + 2\log b + \ldots 2^{p-3}\log^{p-3} b) \\
&\leq 2^{p-2}(\log^{p-2} b)(p!)^2 T_{p-1}^{\mathcal{I}}(b) + 2^{p-2} b\log^{p-3} b.
\end{aligned}$$

In the case when $p = 2$, we have a path, so $T_2^{\mathcal{I}}(b) = b$ and $S_2(b) = M_{\max} + M_{\text{in}}$. So we have

$$\begin{aligned}
S_p^{\mathcal{I}}(b) &\leq 2^{p+1} p(\log^{p-2} b)(M_{\max} + M_{\text{in}}) \\
T_p^{\mathcal{I}}(b) &\leq 2^{p-1}(p!)^2 b(\log^{p-2} b).
\end{aligned}$$

To complete the proof, note that $L(\sigma) = T_p(b) \leq T_p^{\mathcal{I}}(b)$ and $M(\sigma) = S_p(b) \leq 2S_p^{\mathcal{I}}(b)$. $\qquad\square$

To complete the argument for Theorem 7, we note that any path decomposition on n nodes can be reduced to one of length $O(n)$ without increasing the pathwidth. The bounds follow.