MOCCASIN: Efficient Tensor Rematerialization for Neural Networks

Burak Bartan [1] **Haoming Li** [2] **Harris Teague** [1] **Christopher Lott** [1] **Bistra Dilkina** [2]

Abstract

The deployment and training of neural networks on edge computing devices pose many challenges. The low memory nature of edge devices is often one of the biggest limiting factors encountered in the deployment of large neural network models. Tensor rematerialization or recompute is a way to address high memory requirements for neural network training and inference. In this paper we consider the problem of execution time minimization of compute graphs subject to a memory budget. In particular, we develop a new constraint programming formulation called MOCCASIN with only $O(n)$ integer variables, where n is the number of nodes in the compute graph. This is a significant improvement over the works in the recent literature that propose formulations with $O(n^2)$ Boolean variables. We present numerical studies that show that our approach is up to an order of magnitude faster than recent work especially for large-scale graphs.

1. Introduction

The need for efficient deep neural network (DNN) computations for both training and inference continues to grow. Most compute architectures for DNNs make use of a limited amount of fast, *local memory* in conjunction with a much larger amount of *global memory* that is significantly slower to access. For example, accelerators for mobile devices today will dedicate several MB of local memory (per core) for storage of intermediate output tensors before resorting to the devices' main memory; training of large DNNs are done primarily on GPU's local GDDR memory before offloading tensors to DDR memory on the motherboard. However, there are many DNNs in practice for which the local memory is not enough to store all the required intermediate outputs.

[1]Qualcomm AI Research, Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc. [2]University of Southern California, Los Angeles, USA. Correspondence to: Burak Bartan <bbartan AT qti.qualcomm.com>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

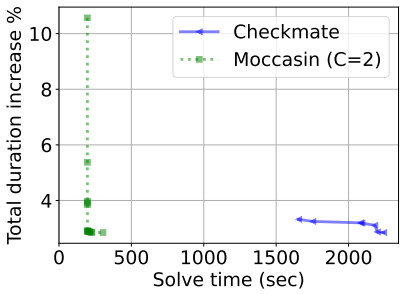

Figure 1. Percentage of total duration increase against solve time in seconds for our proposed method (MOCCASIN) and CHECKMATE (Jain et al., 2020) on a real-world graph with $n = 442$ nodes and $m = 1247$ edges. The memory budget is set to 80% of the peak memory for the initial topological order without rematerialization. The time until the first data point in each curve is spent on presolve.

Effective optimization of the local memory footprint during the computation can make a difference in meeting latency targets for computations since global memory accesses can be too slow (Sze et al., 2017, Sec. V). In addition, the trend toward on-device training will further stress the limited resources due to the need to store intermediate data for gradient back-propagation (tinyML Foundation, 2022).

There are several techniques for DNN computation sequencing that can be used to manage the local memory footprint, and a well-developed toolchain for this purpose is likely to employ multiple methods working together. For example, *operation sequencing* is a technique studied in (Gagrani et al., 2022) and its references. Our paper focuses on another technique called *rematerialization*. When input data is needed for a compute operation, it can be read from local memory, or alternatively be recomputed (rematerialized). In some cases the extra compute can prove to be advantageous relative to occupying valuable local memory with the data until it is ready to be used, with the latter leaving a larger memory footprint. Deciding what operations to rematerialize (or not) in order to optimize latency and memory-use is a PSPACE-complete problem (Gilbert et al., 1979). Further, we often need a method that can tackle large computation graphs with acceptable "compile time", thus need an approach that scales well with graph size.

While there is a history of rematerialization research for managing register use by compilers (Kubota, 1998; Briggs

et al., 1992), more recently, these techniques have been further developed and studied for use in DNNs, where the size of computation graphs and the amount of data movement can be extremely large (Jain et al., 2020; Patil et al., 2022; Schuler et al., 2022). These works formulate a mixed integer linear program (MILP) with a Boolean variable for each use of a compute node's output – either read from memory or rematerialized. We have found that this formulation has limitations when attempting to scale to large graphs due to the need for $O(n^2)$ Boolean variables (where n is the number of nodes in the computation graph). While linear relaxation followed by rounding is an approximation that is leveraged in these works, we have found that these rounded solutions can be far from optimal, limiting the applicability of this approach.

The general rematerialization problem is stated here, and will be made more explicit in Section 2. Given a directed acyclic graph (DAG), $G = (V, E)$, with $|V| = n$ and $|E| = m$, let nodes $v \in V$ represent compute operations and directed edges $(u, v) \in E$ represent data dependencies such that the output tensors of all the nodes $\{w : (w, v) \in E\}$ (predecessors of v) are required to be present in local memory before computing v. Let $seq(G)$ be a *rematerialization sequence* of nodes (ordered list) that contains each node at least once. The rematerialization problem statement is then as follows:

MEMORY-CONSTRAINED COMPUTATION GRAPH SEQUENCING WITH REMATERIALIZATION

$$\underset{seq(G)}{\text{minimize}} \quad \text{total execution duration}$$

subject to

- $seq(G)$ meets the data dependencies of G
- peak memory footprint of $seq(G)$ $\leq$ local memory capacity M

This definition is intentionally high-level and lacking some details that will be provided later. Since our later formulations do not directly optimize the sequence but rather optimize other variables from which memory footprints and the final sequence are computed, we skip further description of such details in this section.

Our work introduces a new optimization model formulation that defines the problem variables as *retention intervals*, allowing use of only $O(n)$ integer variables. We then leverage constrained programming (CP), allowing complex data dependency and cumulative memory threshold constraints to be enforced easily and efficiently during optimization. We further show that our proposed approach demonstrates significant scaling improvements relative to the MILP formulation of CHECKMATE (Jain et al., 2020). Figure 1 provides a numerical comparison of the two methods.

1.1. Related Work

The problem of rematerialization has its roots in compiler optimization (Colombet et al., 2015; Lozano & Schulte, 2019; Lozano et al., 2019) and automatic differentiation with checkpointing (Kubota, 1998; Griewank & Walther, 2008; 2000). Recently, there is a stream of works on rematerialization in machine learning, motivated by the need to train large DNN on GPU with limited memory capacity (Kumar et al., 2019; Kusumoto et al., 2019; Beaumont et al., 2021; Chen et al., 2016; Kirisame et al., 2021; Mostafa, 2022; Huang et al., 2019); most relevant to us are the works based on combinatorial optimization (Jain et al., 2020; Patil et al., 2022; Schuler et al., 2022).

The scope of our work is most closely aligned with that of (Jain et al., 2020), where they develop a mixed integer linear program (MILP) for duration minimization under memory constraints referred to as CHECKMATE. The formulation of CHECKMATE defines Boolean matrix variables R, S, F for node/tensor recomputation, local memory storage, and deallocation, respectively, resulting in $O(n^2)$ Boolean variables. While the paper demonstrates the capability of rematerialization to meet memory limits while minimizing duration, the paper does not sufficiently address the problem complexity and scaling to larger graphs. The authors do mention this issue and offer a method of linear relaxation of the Boolean variables followed by 2-stage rounding to accelerate finding a solution. However, our experiments show that for many graphs, this rounding approach produces results that are far from optimal, and often the rounded solution does not meet the memory threshold and is thus infeasible (see Section 3).

As a consequence of formulating the problem using matrices of variables where the columns represent specific nodes (permutations are not supported), in CHECKMATE all solutions are constrained to follow an input sequence of the graph nodes without rematerialization. We call this the "input topological order". This input must be a valid topological order for solutions that meet the node precedence requirements to be possible. The question then arises, "what input topological order should be used?" Indeed this constraint significantly reduces the space of potential valid rematerialization sequences so can be valuable to limit algorithm complexity. However, as shown in (Gagrani et al., 2022), there can be a wide variability of peak memory footprint for different sequences (without rematerialization).

One of the attractive features of our new formulation is that it does not require an input topological order. However, when studying the graphs used in (Jain et al., 2020), we found no variations in the peak memory footprint across 50 randomly generated topological orders for each graph (without rematerialization). To compare directly with prior methods, we add the input topological order constraint (see Section 2.3) and use this for all of our experiments. Remov-

ing the input topological order constraint and/or studying the impact of different input topological orders on solution quality and solve time are an interesting topic for future research.

A follow-on to CHECKMATE (Patil et al., 2022) recognizes that memory optimization may combine rematerialization with paging – strategically reading and writing intermediate data to external memory. While these external memory accesses are slow, they still may be required if the available local memory is too small to fit any rematerialization solution of practical total duration. This work, however, still suffers from a scaling issue as it inherits the $O(n^2)$ Boolean variable formulation. An extension of our formulation to incorporate paging would be valuable future research. Another extension is (Schuler et al., 2022) which builds on the CHECKMATE formulation to allow making rematerialization decisions across two heterogeneous cores. This extension also suffers from the same fundamental scaling issue.

In (Kumar et al., 2019), a novel method is introduced for finding a valid rematerialization sequence (one that meets the required memory threshold) by performing divide-and-conquer using tree decomposition. This work focuses on proving an attractive complexity for generating such a sequence. However, it is of limited use in practice since it does not contain any explicit mechanism for minimizing total duration of the final sequence produced by the algorithm.

The potential for reducing peak local memory footprint using rematerialization depends on the computation graph topology. For example, a line graph (string of nodes connected by single edges) offers no potential for improvement since the local memory footprint at each node does not depend on past deallocations. In contrast, a simple U-net typically allows significant opportunities for footprint savings. In general, we observe that networks with "long skip connections" are the topologies that exhibit more potential for rematerialization gains. DNN training computation graphs have a "U-net-like" structure since they have forward and backward paths with edges that cross between the two to support gradient computation in the backward path. (Jain et al., 2020) only experiments with training graphs. In contrast, we study rematerialization on both training and inference graphs, and find compelling rematerialization gains also for inference graphs, particularly ones with complex interconnect topology.

1.2. Contributions

- *Retention intervals formulation of rematerialization:* We introduce MOCCASIN, a new formulation with significantly better scaling of solution complexity to large graphs. We demonstrate that this formulation can be solved effectively using *constraint programming* (CP).

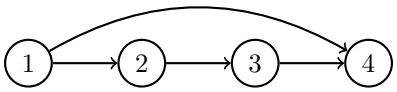

Figure 2. Example graph with 4 nodes.

- *Constrained number of allowed rematerializations:* We introduce parameter C_v that defines the maximum number of times a node v can be computed in the final sequence. We demonstrate empirically that this complexity reduction retains solution quality even for very small values of C_v.

- *Compare and contrast approaches:* We provide comparison of solution speed for CHECKMATE vs MOCCASIN and demonstrate equivalence of solutions.

- *Impact of memory limit:* Prior works make simple assumptions on local memory limit in their evaluation. We show the impact of a range of local memory limits on solution speed and final solution value.

2. Retention Intervals Formulation

The main building block of our formulation is the concept of *output retention intervals* which, as we will show, simplifies the problem formulation greatly. For each node in the computation graph, we will define intervals that indicate the retention of its output in local memory. More precisely, each rematerialization of a node will have its own interval. Here, with a misuse of terminology, we use the term rematerialization for the first time that a node is computed as well. We will assume a node $v \in V$ is allowed to be rematerialized up to C_v times, hence C_v intervals will be defined for node v. The parameter C_v could be considered a hyperparameter. We will show via numerical experiments that picking C_v to be as small as 2 for all of the nodes is often good enough in practice and is without loss of optimality (see Section 3).

We define the retention intervals for each node $v \in V$ by its "start" and "end" times

$$s_v^i, e_v^i \in \mathcal{D}, \ \forall i \in \{1, \ldots, C_v\},$$

where the domain $\mathcal{D}$ is a set of integers of size $O(n)$ that will be defined explicitly in the subsequent subsections. A memory block of size m_v is allocated at the start and deallocated at the end of the interval.

The concept of time in our formulation is defined through *compute events*. This is one of the key components of our proposed formulation. To better illustrate this point, consider Figure 3, which depicts our formulation for the 4-node example graph in Figure 2. In this example, we have set $C_v = 2$ for all nodes v. Figure 3 illustrates the formulation

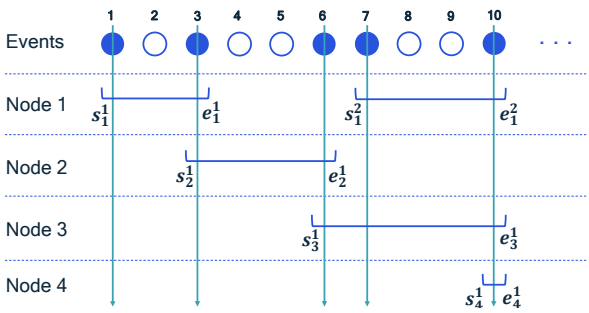

Figure 3. Visualization of the retention intervals formulation.

for this graph where the vertical axis corresponds to different nodes and the horizontal axis is the time partitioned into events. The formulation as visualized in Figure 3 shows two intervals for node 1 and one interval for the other nodes. The reason for this is that the optimal solution for this example case requires only one active interval for nodes 2, 3, 4 while requiring two active intervals for node 1. We note that the starting and ending positions of the intervals in Figure 3 are obtained by solving the optimization problem. These are not known ahead of time, but rather modeled as variables in the optimization problem. Furthermore, this figure depicts circles as visual representation of potential node execution events. For example, in Figure 3, node 1 is computed during event 1 and its output is kept in memory until event 3. The second interval for node 1 starts in event 7 indicating that node 1 is recomputed during event 7. Observe that event 1 has a memory usage of m_1, while event 7 has a memory usage of $m_1 + m_3$, as the output of node 3 is retained in memory when node 1 is recomputed.

The total duration of the rematerialization sequence is a weighted sum over the events in which a node is being executed, where the weights are simply the actual durations of the nodes (in seconds or processor cycles). Such events are marked with a filled circle in Figure 3. We say that the *peak memory footprint* of a solution is the maximum memory usage among all events. The cyan colored arrows in Figure 3 represent the memory usage of node execution events. The peak memory footprint of the solution in the example is 3, assuming each node outputs a unit-size tensor, realized at event 10. Observe that events marked with empty circles do not contribute to either the duration or the peak memory footprint of the solution.

The primary reason why we work with this event-based definition for time is to keep the domain of the variables s_v^i, e_v^i at a small size. More concretely, the CP-SAT solver (Perron & Furnon, 2022), which we use to solve numerically our proposed optimization problem, accepts only integer variables as it implements Lazy Clause Generation (Ohrimenko et al., 2009), known to be effective for constrained scheduling problems (Schutt et al., 2013). If we instead set the time domain to be seconds (which would require quantization) or processor cycles, then the corresponding CP formulation would require variable domains of larger sizes. The domain size of the variables in the formulation has a direct impact on the solver speed especially for CP solvers.

Note that for each interval, the first time slot (shown using a filled circle in Figure 3) is dedicated to the computation of node v while the rest of the interval indicates the retention of the output of node v. It follows that intervals may not start at the same time due to our system model where nodes are executed sequentially, i.e. there is no parallel computing. The empty circles in Figure 3 indicate that no interval starts at that particular time, i.e., no computation occurs. The decision of which circles will be filled or empty is a by-product of the optimization problem and not known ahead of time. Furthermore, it is important to note that a memory block of size m_v is allocated during the entirety of the interval. Also, observe that since the first event of an interval indicates the compute of that node, this is when the predecessors of that node must be available in memory.

2.1. Optimization Problem Formulation

In addition to the start and end times of the intervals, we define the Boolean variables a_v^i to model whether the i'th interval of node v is active or inactive. This grants us the flexibility of not requiring a node v to be rematerialized exactly C_v times. If inactive (i.e. $a_v^i = 0$), the corresponding interval does not contribute to the sum in the objective as well as the sums in the memory and precedence constraints. We assume that the duration w_v and output size m_v for each node are known. Using the retention intervals as the foundation of our formulation, we state the rematerialization problem as an optimization problem as follows:

$$\underset{s,e,a}{\text{minimize}} \quad \sum_{v,i} w_v a_v^i \tag{1}$$

$$\text{subject to} \quad s_v^i \leq e_v^i,\ \forall v, i \tag{2}$$

$$e_v^i \leq s_v^{i+1},\ \forall v, \forall i < C_v \tag{3}$$

$$\sum_{v,i\,:\,s_v^i \leq t \leq e_v^i} m_v a_v^i \leq M,\ \forall t \in \mathcal{D} \tag{4}$$

$$\forall (u,v) \in E,\ \forall i \in \{i : a_v^i = 1\},\ \exists j \text{ such that}$$

$$a_u^j = 1, s_u^j + 1 \leq s_v^i \leq e_u^j \tag{5}$$

$$s_v^i \neq s_u^j,\ \forall v, u \in \{v, u : v \neq u\},\ \forall i, j \tag{6}$$

$$a_v^1 = 1,\ \forall v \tag{7}$$

$$s_v^i, e_v^i \in \mathcal{D},\ a_v^i \in \{0,1\},\ \forall v, i\,. \tag{8}$$

The objective (1) is the total duration. Constraints (2) enforce that the end time of each interval comes after its start time. Constraints (3) ensure that intervals of the same node

do not overlap. The memory budget constraints are given in (4), where the parameter t represents time. The precedence constraints (5) enforce that there exists an overlapping and active interval for all predecessors of v at all start times s_v^i of v. Observe that the memory and precedence constraints are nonlinear constraints. We describe in detail how the *cumulative constraint* and the *reservoir constraint* from constraint programming are used to model these nonlinear constraints in the next subsection. The constraints in (7) are intended to make the first retention interval for every node active. The constraints (6) ensure that the starting times of the intervals are all different, i.e. compute events do not overlap.

The domain of the variables s_v^i, e_v^i is as follows

$$\mathcal{D} = \{1, 2, \ldots, \sum_v C_v\}\,, \tag{9}$$

where $\sum_v C_v$ is equal to the number of intervals, which is upper bounded by $|\mathcal{D}| \leq n \max_v C_v$. Observe that we may not have more than $\sum_v C_v$ filled circles in Figure 3.

2.2. Memory and Precedence Constraints using CP

Memory constraints: The set of constraints in (4) enforces that the resource usage never exceeds the local memory budget M at any time t. Note that the constraint (4) is not linear in the variables s_v^i, e_v^i. This type of inequality can be modeled using the CP constraint `cumulative`.

The cyan colored arrows in Figure 3 are intended to visually represent how we model the memory constraint. Each vertical arrow can be viewed as a separate inequality constraint. In particular, the sum of output sizes m_v for each interval an arrow intersects is less than or equal to M. Recall that it is not needed to have an arrow for each circle in Figure 3 since the memory use could not increase during the empty circles. Hence, we only need to consider the filled circles. However, we do not know which circles are to be filled ahead of time. This is readily addressed by the cumulative modeling which considers only the beginning of intervals for memory calculation.

We have used the function `AddCumulative` from CP-SAT (Perron & Furnon, 2022) for the memory constraints. This function requires intervals, demands, and capacity as input arguments. Intervals are the retention intervals from our formulation defined by the start and end variables s_v^i, e_v^i and considered only when $a_v^i = 1$. The demand for each interval is the output size m_v for the corresponding node v. Finally, the capacity is simply the memory budget M. The pseudo-code representation of this constraint is as follows:

$$\texttt{cumulative}(\{s_v^i, e_v^i, a_v^i, m_v\}_{v \in V, i \in \{1..C_v\}}, M)\,.$$

Precedence constraints: We recall that prior to the execution of each compute task, the outputs of all of its predecessors must be available in memory. This could be written as the constraints in (5). Note that these are nonlinear constraints in the variables. We employ the `reservoir` constraint from constraint programming to model the data dependencies. We view each predecessor as a resource whose level needs be maintained such that it does not go below 0 while its successor is being executed.

Let us define the resource level change events $f(\cdot)$ for each edge $(u, v) \in E$ and $i \in \{1, \ldots, C_v\}$:

$$\begin{aligned}
&f(s_v^i) = -1 \\
&f(s_v^i + 1) = 1 \\
&f(s_u^1) = 1, f(s_u^2) = 1, \ldots, f(s_u^{C_u}) = 1 \\
&f(e_u^1) = -1, f(e_u^2) = -1, \ldots, f(e_u^{C_u}) = -1\,.
\end{aligned} \tag{10}$$

The level change function $f(\cdot)$ returns 0 at every other point in time except for those in (10). The function `AddReservoirConstraintWithActive` from CP-SAT takes the following input parameters: times, demands, actives, minimum level. Times and demands are as defined in (10). Actives are the Boolean variables a_v^i and the minimum level is set to 0. This constraint ensures that for node v, one of the intervals for each of its predecessors u overlaps with the starting time of the node v.

Finally, the nonlinear constraints (6) could be addressed using the `alldifferent` constraint from CP.

2.3. Complexity Reduction by Enforcing an Initial Topological Ordering

The proposed framework offers the flexibility of allowing solutions that are not limited to a predetermined topological ordering. However, one could enforce an input topological ordering to reduce the size of the search space and in turn reduce the solve time. We expand upon this point in this subsection.

We start by considering a new domain for the integer variables s_v^i, e_v^i:

$$\mathcal{D} = \left\{1, 2, \ldots, \frac{n(n+1)}{2}\right\}, \tag{11}$$

where the domain size is now $O(n^2)$ rather than $O(n)$. We now introduce the concept of *stages* into our formulation, which has been previously employed in (Jain et al., 2020). The j'th stage contains j events; this is depicted in Figure 4. Given a topological ordering of the nodes, the first stage enforces the computation of the first node in the order. In subsequent stages, the j'th node in the input topological order is enforced to be computed in the last event of stage j, while the previous events in stage j allow nodes $1, \ldots, (j-1)$ to be computed. In particular, we enforce that node j could be computed only in event j of a stage. Since this

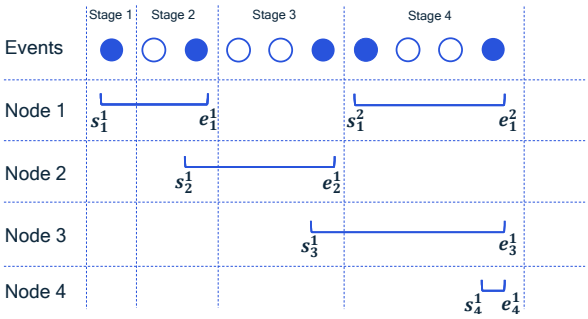

Figure 4. Illustration of events grouped into stages.

restricts the starting times of each interval to specific events, we no longer need to explicitly include the constraint (6).

Note that the start time for the first retention interval for each node is no longer a variable but a fixed value. More precisely, for node v, the value of s_v^1 is equal to $j(j+1)/2$, where j is the index of node v in the input topological ordering.

2.4. Optimization in Two Phases

Our approach consists of two phases. In Phase 1, we solve a variant of the optimization problem in (1) - (8) to find a memory feasible solution. This solution is then used as initialization for Phase 2, which is the optimization problem stated in (1) - (8). The problem in Phase 1 has the objective

$$\underset{s,e,a,M_{var}}{\text{minimize}} \quad \max(M_{var}, M) \tag{12}$$

where we introduce the variable $M_{var} \in \mathbb{R}$ for the peak memory footprint. We modify the memory constraint in (4) as follows for Phase 1:

$$\sum_{v,i\,:\,s_v^i \le t \le e_v^i} m_v a_v^i \le M_{var}, \forall t \in \mathcal{D}\,. \tag{13}$$

The goal of Phase 1 is to arrive at an intermediate solution, whose peak memory footprint is below the local memory target M. We address this by considering the maximum of the peak memory variable M_{var} and memory budget M as the objective. This objective could be linearized by introducing an auxiliary variable $\tau \in \mathbb{R}$ and then considering: minimize τ subject to $\tau \ge M_{var}$ and $\tau \ge M$. The other constraints of the optimization problem are unchanged.

Note that any topological ordering of the graph G provides a trivial feasible solution to the problem in Phase 1 since it does not have a hard memory constraint. For the rematerialization problem itself, however, there is no easy solution that could be constructed and provided as an initial solution. Phase 1 is intended to fill this gap.

2.5. Overview of Formulations

Table 1 provides a complexity comparison for our approach and the approach of (Jain et al., 2020). When comparing the number of constraints for different formulations, it is important to keep in mind that the constraints in the MILP are all linear constraints while the constraints of the CP could be linear or nonlinear.

Remark 2.1. The optimization problem in (1) - (8) is specified using integer variables s_v^i and e_v^i while the discrete variables of the MILP in (Jain et al., 2020) are all Boolean. The fact that our formulation is stated using integer variables as opposed to Boolean variables is only for convenience. For a more direct comparison of variable counts, it is straightforward to rewrite this optimization problem by representing the integer variables as Boolean sequences, in which case the number of Boolean decision variables will be $O(Cn \log n)$ (taking $C = \max_v C_v$) as opposed to $O(n^2)$ Boolean variables in the CHECKMATE formulation.

Note that the size of search space in Table 1 for MOCCASIN is dominated by the term n^{Cn}. For the version of MOCCASIN with Boolean decision variables, the search space size would scale with $2^{Cn \log n}$, which is the same as n^{Cn}. This is smaller than the size of the search space for CHECKMATE, which is 2^{n^2+nm}.

3. Numerical Results

3.1. Graphs for Evaluation

The graphs we use for evaluation of the method presented in this paper come from several sources. Additional descriptions can be found in Appendix A.2.

Checkmate Graphs: We use graphs from the checkmate repository (Jain, 2020). They represent the single-batch training computation graphs for a selected set of neural networks. Use of these graphs allows us to compare directly to results presented in that work.

Random Layered Graphs: We leverage the synthetically constructed random layered graphs introduced in (Gagrani et al., 2022, Appendix A) as examples of graphs with complex interconnect topology.

Real-world Graphs: While our synthetic and standard, public-domain graphs are valuable for experimentation, we also see value in presenting results obtained from neural computation graphs used for commercial development of artificial intelligence hardware and software products. We sample a set of representative graphs that have diverse architectures and size.

Table 1. Overview of formulation complexities for CHECKMATE and our proposed method.

Formulation	# Bool. vars	# integer vars	Size of search space	# constraints
CHECKMATE - MILP	$O(n^2 + nm)$	-	$O(2^{n^2+nm})$	$O(n^2 + nm)$
MOCCASIN - CP	$O(Cn)$	$O(Cn)$ (domain size = $O(n)$)	$O(2^{Cn} + n^{Cn})$	$O(Cm)$

3.2. Experimental Setup

We use the open source CP-SAT solver from Google OR-Tools (Perron & Furnon, 2022) to solve the CP. For the comparisons against CHECKMATE, we have used the implementation provided in the codebase for (Jain et al., 2020), which utilizes Gurobi (Gurobi Optimization, LLC, 2023) and CVXPY (Diamond & Boyd, 2016; Agrawal et al., 2018). We have run all of the numerical experiments on a 16-CPU core workstation with a 32 GB of RAM. Furthermore, we note that once a solution to the optimization problem is obtained, the corresponding sequence of tasks can be executed on any hardware, CPU or GPU. We note that this is due to the generality of our approach where we do not make any assumptions on the computing unit that will be used to execute the computations.

3.3. Experimental Results

Figure 5 shows the solve progress plots for 4 random layered graphs $G_1, \ldots, G_4$ of varied sizes and 4 different local memory budget values for each graph. The curves of MOCCASIN are all shifted to the right by the amount of time spent in Phase 1 for fair comparison. We have set the time limit to 30 minutes for G_1, G_2, G_3 and 1 hour for G_4.

For G_1, the solve times are in the order of a few seconds and we observe that MOCCASIN is faster to obtain similar objective values as CHECKMATE. For G_2, which has $n = 250$ nodes, when the memory budget is tight, we see that CHECKMATE fails to find a feasible solution within the given time limit of 30 minutes. For the highest memory budget value, it finds the optimal solution in 10 minutes while MOCCASIN finishes in a few seconds. For graphs G_3 and G_4, which have $n = 500$ and $n = 1000$ nodes respectively, CHECKMATE times out with no solution even if we consider a higher time limit of 3 hours while MOCCASIN converges to a good solution (i.e. low total duration increase) under less than an hour. In addition to the solve time, the memory required for optimization also becomes a critical factor for large graphs. In fact, for G_3 and G_4, CHECKMATE exits with an out-of-memory error. The results in Figure 5 are best understood by contrasting the number of discrete variables in CHECKMATE, which grows quadratically in n while it grows $n \log(n)$ in MOCCASIN (see Table 1).

In all of the experiments, we have set $C_v = 2$ for all $v \in V$, which we specified in the plot legends by $C = 2$. We have used the version of the formulation in Section 2.3 where we enforce an input topological ordering, which we have randomly generated.

Table 2 provides numerical results for a range of different computation graphs. In Table 2 we have selected the memory budget values for each graph to be the 80% and 90% of the initial peak memory without rematerialization. In addition to the methods of CHECKMATE and MOCCASIN, we include results for the rounding algorithm proposed in (Jain et al., 2020) under the "LP+rounding" column of the table. This method consists of relaxing the MILP into a linear program (LP) and then rounding the solution (see (Jain et al., 2020) for further details on this algorithm). Note that solution produced by the rounding algorithm is not guaranteed to satisfy the memory budget constraint. This could be seen in Table 2 where in most cases the peak memory for the relaxation and rounding approach is higher than the memory budget M.

Table 2 shows that the random layered graphs and real-world graphs are the most challenging ones among the graph set. The solve times for these graphs are higher than the CM graphs, which is consistent with the fact that they have higher edge densities and more complex edge connectivities.

It is critical to highlight that MOCCASIN is able to scale to graphs with few hundreds of nodes and a few thousands of edges. This is while achieving a few percents of total duration increase, which is essential to performance.

4. Conclusion

Rematerialization is a valuable technique that allows to save peak local memory footprint at the expense of longer total duration of computation. However, finding a rematerialization sequence that minimizes total duration while meeting a prescribed local memory limit is not easy - especially for graphs with complex interconnections. Previous research (Jain et al., 2020) demonstrates a MILP formulation that can address this optimization problem, however our experiments show that that approach does not scale well as the graph size, and the complexity of graph topology grow.

We introduce MOCCASIN, a new formulation that expresses the decision variables as *retention intervals* specified by the start and end event indices of each node's output tensor. We further manage complexity using the hyper-parameter C_v

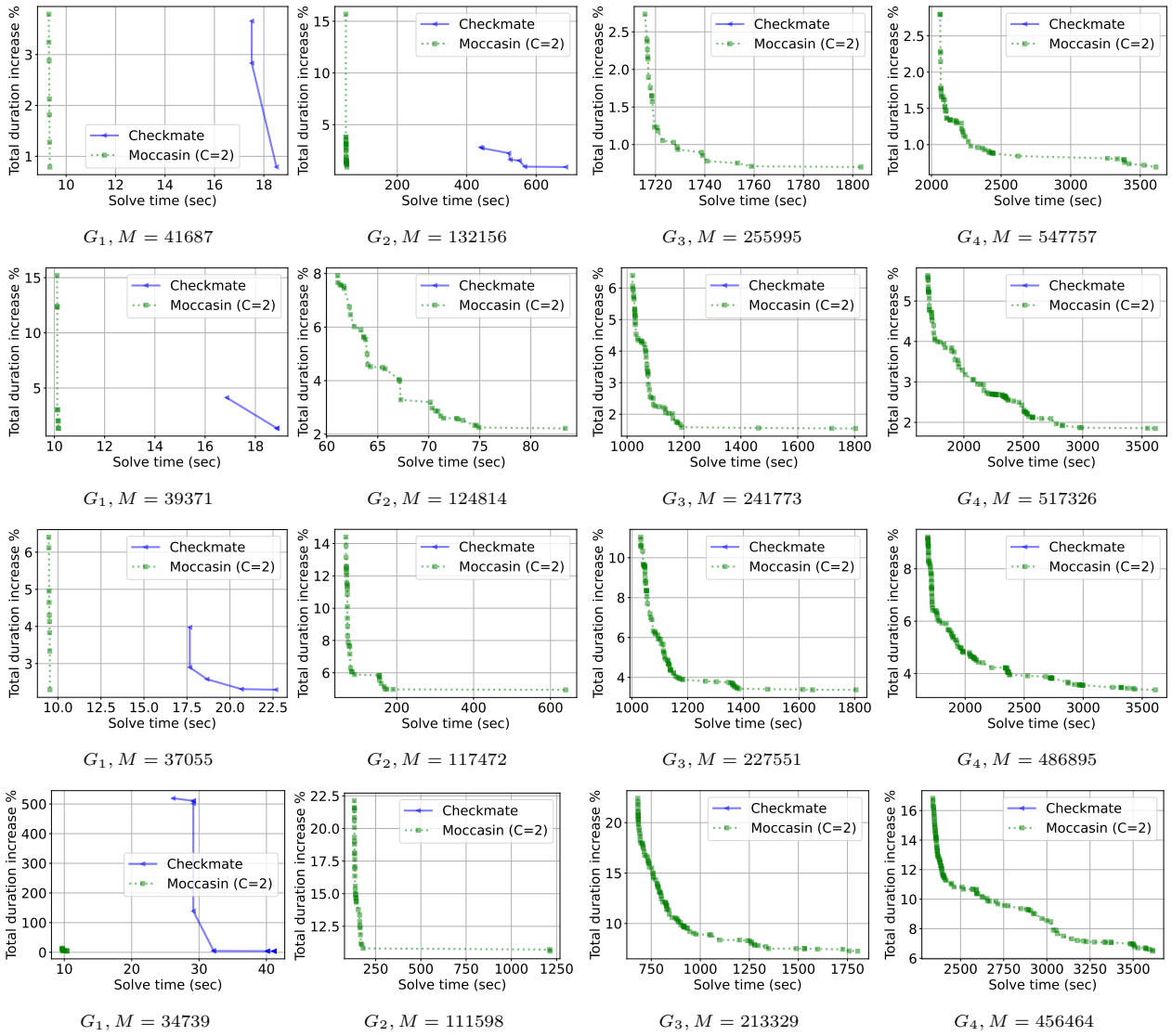

Figure 5. Solve progress plots for random layered graphs under different memory budget constraints. The graph name and memory budget are as indicated in the caption for each plot. The node and edge counts (n, m) for the graphs are as follows: $G_1 : (100, 236), G_2 : (250, 944), G_3 : (500, 2461), G_4 : (1000, 5875)$.

which limits the maximum number of retention intervals for each node. Our experimental results include CHECKMATE graphs (training graphs with simple interconnect topology), synthetically generated random layered graphs (that model inference graphs with complex topology), and real-world inference graphs in active use commercially. We demonstrate that MOCCASIN provides the same optimization result as CHECKMATE with up to an order-of-magnitude less solve time for mid-sized graphs (100-250 nodes). It also enables us to find solutions for larger graphs (up to 1000 nodes) and for more challenging peak local memory limits, cases where the MILP formulation of CHECKMATE failed to return any solution within the time limit.

Finally, rematerialization is seen as one component among a set of tools for managing memory use during execution of complex computation graphs arising in deep learning applications. Joint optimization of this with other methods such as sequencing, scheduling for parallel compute, and paging to global memory are viewed as valuable topics for further research.

References

Agrawal, A., Verschueren, R., Diamond, S., and Boyd, S. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

Table 2. Experiment results. The abbreviations used in the table are as follows: RL: Random layered graphs, RW: Real-world graphs, CM: graphs used in (Jain et al., 2020) to evaluate CHECKMATE, where CM 1 is FCN with VGG layers and CM 2 is the ResNet50 model, n: number of nodes, m: number of edges, M: memory budget, TDI: total duration increase in percentage, peak mem: peak memory of the resulting rematerialization sequence. The column 'Time (s)' indicates the elapsed time in seconds until the best solution. Dashes "-" indicate that no solution is found. The best solution in each row is shown in bold font. Full version of this table could be found in the Appendix.

			CHECKMATE MILP			CHECKMATE LP+Rounding			MOCCASIN		
Graph	(n, m)	M	TDI %	Peak mem	Time (s)	TDI %	Peak mem	Time (s)	TDI %	Peak mem	Time (s)
RL G_2	(250, 944)	132,156	0.9	132,130	685.1	93.0	178,200	401.7	0.9	131,831	**55.0**
		117,472	-	-	-	328.6	181,200	696.9	4.9	117,400	**639.5**
RL G_4	(1000, 5875)	547,757	-	-	-	-	-	-	0.7	547,660	**3612.9**
		486,895	-	-	-	-	-	-	3.4	486,880	**3611.8**
RW 1	(358, 947)	20,227,276	2.3	20,226,048	1340.0	-	-	-	2.3	20,226,048	**123.5**
		17,979,801	4.5	17,977,344	1605.4	-	-	-	4.5	17,977,344	**122.0**
RW 2	(442, 1247)	10,817,740	1.4	10,811,392	1856.7	-	-	-	1.4	10,813,440	**1201.3**
		9,615,769	2.8	9,615,360	2242.9	-	-	-	2.8	9,613,312	**303.9**
RW 3	(574, 1304)	10,539,417	-	-	-	-	-	-	0.8	10,539,008	**1802.4**
		9,368,371	-	-	-	-	-	-	1.6	9,367,552	**1802.8**
CM 1	(73, 149)	11.3 GB	0.0	11.1 GB	6.3	0.0	11.4 GB	6.8	0.0	11.1 GB	**3.1**
		10.0 GB	0.1	9.65 GB	5.6	0.1	10.8 GB	6.7	0.1	9.9 GB	**3.1**
CM 2	(353, 751)	31.9 GB	**0.1**	31.6 GB	434.1	0.1	31.5 GB	505.2	0.2	31.9 GB	**65.2**
		28.4 GB	0.3	28.3 GB	485.3	0.5	27.8 GB	1065.4	0.3	28.4 GB	**69.3**

Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021.

Briggs, P., Cooper, K. D., and Torczon, L. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp. 311–321, 1992.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Colombet, Q., Brandner, F., and Darte, A. Studying optimal spilling in the light of ssa. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

Diamond, S. and Boyd, S. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

Gagrani, M., Rainone, C., Yang, Y., Teague, H., Jeon, W., Bondesan, R., van Hoof, H., Lott, C., Zeng, W. W., and Zappi, P. Neural topological ordering for computation graphs. In *Advances in Neural Information Processing Systems*, 2022.

Gilbert, J. R., Lengauer, T., and Tarjan, R. E. The pebbling problem is complete in polynomial space. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pp. 237–248, 1979.

Griewank, A. and Walther, A. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

Griewank, A. and Walther, A. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL `https://www.gurobi.com`.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jain, P. Checkmate repository, 2020. URL `https://github.com/parasj/checkmate/tree/mlsys20_artifact`.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=Vfs_2RnOD0H`.

Kubota, K. A fortran77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. *Optimization Methods and Software*, 10(2):319–335, 1998.

Kumar, R., Purohit, M., Svitkina, Z., Vee, E., and Wang, J. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Kusumoto, M., Inoue, T., Watanabe, G., Akiba, T., and Koyama, M. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

Lozano, R. C. and Schulte, C. Survey on combinatorial register allocation and instruction scheduling. *ACM Computing Surveys (CSUR)*, 52(3):1–50, 2019.

Lozano, R. C., Carlsson, M., Blindell, G. H., and Schulte, C. Combinatorial register allocation and instruction scheduling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(3):1–53, 2019.

Mostafa, H. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. *Proceedings of Machine Learning and Systems*, 4:265–275, 2022.

Ohrimenko, O., Stuckey, P. J., and Codish, M. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

Patil, S. G., Jain, P., Dutta, P., Stoica, I., and Gonzalez, J. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pp. 17573–17583. PMLR, 2022.

Perron, L. and Furnon, V. Or-tools, 2022. URL `https://developers.google.com/optimization/`.

Schuler, M., Membarth, R., and Slusallek, P. Xengine: Optimal tensor rematerialization for neural networks in heterogeneous environments. *ACM Transactions on Architecture and Code Optimization*, pp. 5, 2022.

Schutt, A., Feydy, T., Stuckey, P. J., and Wallace, M. G. Solving rcpsp/max by lazy clause generation. *Journal of scheduling*, 16(3):273–289, 2013.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

tinyML Foundation. tinyML On-Device Learning Forum, 2022. URL `https://www.tinyml.org/event/on-device-learning/`.

A. Appendix

A.1. Additional Numerical Experiments

Figure 6 shows the time until best solution as a function of number of nodes n in log-log scale. The reason why the blue curve stops after $n = 250$ is that CHECKMATE does not produce a feasible solution within the given time limit of 1 hour. Figure 6 shows that our method demonstrates better scalability. In instances where both methods return feasible solutions, the objective function values are the same. For graphs with more than $n = 250$ nodes, for the feasible solutions that our method finds, the corresponding total duration increase is consistently less than 5%.

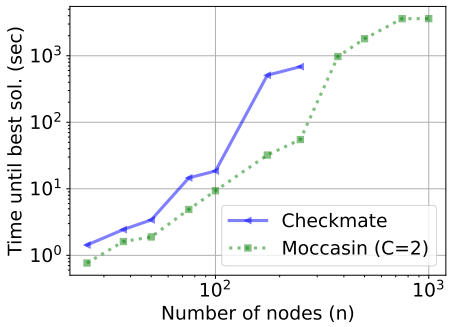

Figure 6. Time until the best solution as a function of number of nodes n for random layered graphs. The memory budget is selected for each graph as the 90% of the peak memory for the input topological order.

Table 3 shows the extended version of Table 2, where results for additional graphs are included.

Table 3. Extended version of Table 2.

			CHECKMATE MILP			CHECKMATE LP+Rounding			MOCCASIN		
Graph	(n, m)	M	TDI %	Peak mem	Time (s)	TDI %	Peak mem	Time (s)	TDI %	Peak mem	Time (s)
RL G_1	(100, 236)	41,687	0.8	41,360	18.5	0.1	44,939	16.0	0.8	40,820	9.3
		37,055	2.3	36,830	22.7	7.3	43,370	20.1	2.3	36,810	9.5
RL G_2	(250, 944)	132,156	0.9	132,130	685.1	93.0	178,200	401.7	0.9	131,831	55.0
		117,472	-	-	-	328.6	181,200	696.9	4.9	117,400	639.5
RL G_3	(500, 2461)	255,995	-	-	-	-	-	-	0.7	255,959	1803.3
		227,551	-	-	-	-	-	-	3.4	227,429	1804.8
RL G_4	(1000, 5875)	547,757	-	-	-	-	-	-	0.7	547,660	3612.9
		486,895	-	-	-	-	-	-	3.4	486,880	3611.8
RW 1	(358, 947)	20,227,276	2.3	20,226,048	1340.0	-	-	-	2.3	20,226,048	123.5
		17,979,801	4.5	17,977,344	1605.4	-	-	-	4.5	17,977,344	122.0
RW 2	(442, 1247)	10,817,740	1.4	10,811,392	1856.7	-	-	-	1.4	10,813,440	1201.3
		9,615,769	2.8	9,615,360	2242.9	-	-	-	2.8	9,613,312	303.9
RW 3	(574, 1304)	10,539,417	-	-	-	-	-	-	0.8	10,539,008	1802.4
		9,368,371	-	-	-	-	-	-	1.6	9,367,552	1802.8
RW 4	(698, 1436)	24,328,396	-	-	-	-	-	-	1.7	24,328,192	1803.9
		21,625,241	-	-	-	-	-	-	3.4	21,624,832	1314.0
CM 1	(73, 149)	11.3 GB	0.0	11.1 GB	6.3	0.0	11.4 GB	6.8	0.0	11.1 GB	3.1
		10.0 GB	0.1	9.65 GB	5.6	0.1	10.8 GB	6.7	0.1	9.9 GB	3.1
CM 2	(353, 751)	31.9 GB	0.1	31.6 GB	434.1	0.1	31.5 GB	505.2	0.2	31.9 GB	65.2
		28.4 GB	0.3	28.3 GB	485.3	0.5	27.8 GB	1065.4	0.3	28.4 GB	69.3

A.2. Examples of Graphs Used in Experiments

Figure 7 shows visualizations of two of the graphs that we use for our experiments to give a better understanding of the structure of these graphs. The Checkmate FCN8 graph (left) is a training computation graph and the 100-node random layered graph (right) models an inference graph. We study much larger graphs, but these are quite difficult to visualize in two dimensions.

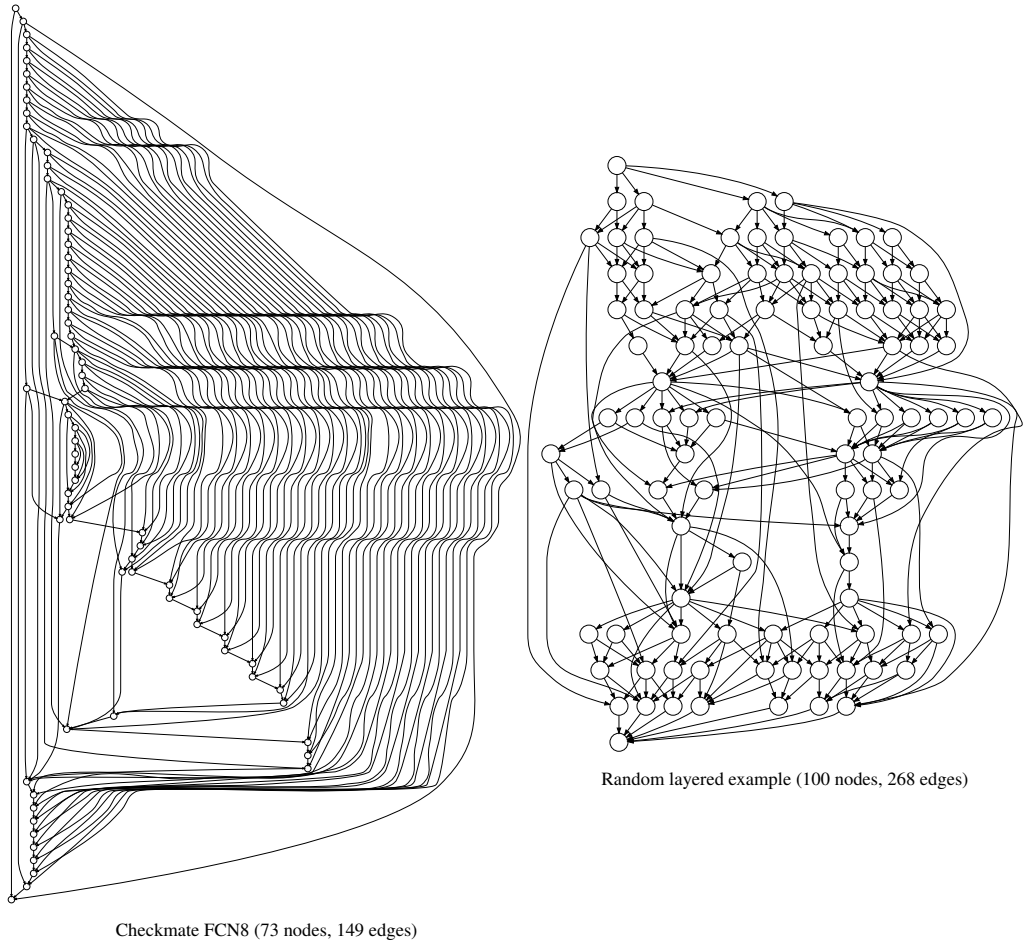

Figure 7. The example graphs used for our experiments are shown using a simple visualization showing only the structure of the graphs.