Optimizing DNN Computation Graph using Graph Substitutions

Jingzhi Fang* Yanyan Shen† Yue Wang‡ Lei Chen*
*The Hong Kong University of Science and Technology
†Shanghai Jiao Tong University
‡Shenzhen Institute of Computing Sciences, Shenzhen University
jfangak@connect.ust.hk, shenyy@sjtu.edu.cn, yuewang@sics.ac.cn, leichen@cse.ust.hk

ABSTRACT

Deep learning has achieved great success in various real-world applications. As deep neural networks (DNNs) are getting larger, the inference and training cost of DNNs increases significantly. Since one round of inference or one iteration in the training phase of a DNN is typically modeled as a computation graph, existing works propose to optimize computation graphs by performing a sequence of functionally equivalent graph substitutions, leading to higher inference and training efficiency. In this work, we formally define the Optimizing Computation Graph using Graph Substitutions (OCGGS) problem, and prove it to be NP-hard and Poly-APX-complete. We develop two exact and efficient methods to the OCGGS problem. The pruning-based algorithm eliminates the examination of redundant graph substitution sequences, and the dynamic programming with pruning algorithm makes use of the explored graph substitutions. To further speed up the search process, we propose a sampling heuristic which is effective to optimize complex computation graphs with polynomial time and space complexity. Extensive experiments on various DNN architectures and sizes are conducted to verify the effectiveness and efficiency of our proposed solutions compared with existing techniques.

PVLDB Reference Format:
Jingzhi Fang, Yanyan Shen, Yue Wang, Lei Chen. Optimizing DNN computation graph using graph substitutions. *PVLDB*, 13(11): 2734-2746, 2020.
DOI: https://doi.org/10.14778/3407790.3407857

1. INTRODUCTION

Deep learning has revolutionized many practical applications including video analytics, natural language processing, etc. For example, practitioners use DNNs to identify objects in videos or images [8, 14, 17], translate one language to another [6, 11, 19], and recommend commodities to customers [7, 16, 18]. With the increasing availability of data and computing power, modern DNNs are getting larger in order to deliver high performance for complex problems. The ILSVRC 2015 classification challenge winner ResNet [8] involves up to 152 layers, and BERT-large [6] has 340 million parameters. As a result, the inference and training of such DNNs become time-consuming. Since one round of inference or one iteration in the training phase of a DNN is typically modeled as a computation graph consisting of inputs, outputs, and parametric operations, there exists an opportunity to optimize the computation graph towards higher inference and training efficiency.

A common practice to optimize a DNN computation graph is to perform *graph substitutions* following a set of predefined rules, where each rule describes how to replace a computation subgraph with another functionally equivalent one. For example, one rule used by TensorRT [3] is to replace the convolution, bias and ReLU operations in a computation graph with a new equivalent operation which fuses them all, and the fused operation can take less time to be executed. By conducting a sequence of graph substitutions over the original computation graph, it is possible that the optimized computation graph consumes less time in training and inference compared with the original one. In this paper, we would like to investigate how to optimize DNN computation graphs effectively based on a given set of substitution rules.

In fact, it is a non-trivial task to optimize DNN computation graphs with graph substitutions. The challenge originates from the huge search space of all possible graph substitution sequences. A graph substitution, which does not contribute the biggest efficiency improvement or even degrades the efficiency of a computation graph, may allow the subsequent substitutions to be performed and achieve the optimal efficiency improvement in the end. For example, in Figure 1a, the computation graph contains two convolution operations (one has 256 kernels of size (3×3) and the other has 256 kernels of size (1×1)) followed by a concatenation operation. One possible graph substitution sequence is to first enlarge the kernel size of the convolution in v_3 to (3×3) and then merge the two convolutions in v_2 and v_3 into one. The inference time of the graph with an input tensor of size $(1 \times 256 \times 14 \times 14)$ on an NVIDIA Tesla P100 GPU would first increase by 0.04 ms due to the kernel enlarging and then decrease by 0.07 ms thanks to the convolution merging, leading to an overall 0.03 ms runtime reduction per iteration of the graph. Many of the existing systems such as TVM [5] and TensorRT [3] only select the rules that can continuously improve the efficiency, which may get stuck at local optima. While some methods [9, 10] take efficiency-degrading rules

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407857

Table 1: Notation table.

Variable	Description
$G = (V, E)$	a computation graph
$\phi = (G_s, G_t, f)$	a substitution rule
$\tau = (\phi, g : V_s \to V_m)$	a graph substitution
$P = (\tau_1, ..., \tau_k)$	a graph substitution sequence
K	a length constraint in OCGGS
$\Gamma_v = (\tau, j)$	a label of node v
Δ	a set of graph substitutions
$\mathcal{P}$	graph substitutions sequence set
Q	sample size
η	further exploration parameter

into account and adopt cost-based backtracking search algorithms, they fall short in providing guarantee on search time and space complexity (in the worst case, they would explore the whole search space), and the search efficiency can be unacceptable when optimizing complex DNNs (e.g., taking tens of hours or more to optimize NasNet-A [20]).

In order to study the problem of finding the optimal graph substitution sequence, we formulate it as the Optimizing Computation Graph using Graph Substitutions (OCGGS) problem. Specifically, given the original computation graph G and a set of substitution rules Φ, the OCGGS problem aims to find the optimal graph substitution sequence within a maximum length K. We theoretically prove the OCGGS problem is NP-hard by reducing from the maximum flow problem with conflict graph [12] and further prove it is Poly-APX-complete. To avoid enumerating all the possible sequences, we design an efficient pruning-based method by eliminating the examination of redundant graph substitution sequences. We also develop a dynamic programming algorithm, which makes use of the explored graph substitutions to speed up the search process. The above two algorithms are exact solutions. To further improve the search efficiency for complex models, we introduce a sampling-based heuristic with polynomial time complexity and a good optimization effect. To summarize, we have made the following contributions in this paper:

(1) We formally define the OCGGS problem that aims to reduce the computation cost of any DNN computation graph by performing a sequence of graph substitutions based on a set of substitution rules. We prove that the problem is NP-hard and even Poly-APX-complete.

(2) We develop two exact and efficient methods for the OCGGS problem: a pruning-based algorithm and a dynamic programming with pruning algorithm. We provide analysis on the time and space complexity of the two methods.

(3) We propose an effective sampling algorithm with polynomial time complexity to accelerate the search process especially for complex models.

(4) We conduct extensive experiments to verify the effectiveness and efficiency of our proposed methods on optimizing DNNs, compared with various baselines.

The remainder of this paper is organized as follows. We define the OCGGS problem and provide its hardness results in Section 2. Section 3 and 4 present two exact algorithms. Section 5 introduces an efficient sampling heuristic. Experiment results are provided in Section 6. We review the related works in Section 7 and conclude this paper in Section 8. Section 9 is the acknowledgments.

2. PRELIMINARIES

In this section, we first introduce some basic concepts and then formally define the OCGGS problem. Table 1 summarizes all the symbols used throughput this paper.

2.1 Computation Graph and Cost Function

We model one round of inference process or one iteration in the training phase of a DNN by a directed acyclic computation graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. A node $v \in V$ represents a constant value or a parametric operation (e.g., convolution), and an edge $e(v_s, v_t) \in E$ describes the data dependence between two nodes, i.e., the output of v_s is an input of v_t.

There are two special types of nodes in G, *input and output nodes*. The input and output nodes only have outgoing and incoming edges, respectively. The result of an output node is an output of G. For the nodes in G that are neither input nor output nodes, we refer to them as the *inner nodes* of G. There is no constraint on the number of input and output nodes in a computation graph.

EXAMPLE 1. *Consider the computation graph $G = (V, E)$ in Figure 1b. The node set V is $\{v_1, \cdots, v_5\}$, and the edge set E is $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5)\}$. v_1 and v_5 are the input and output node respectively (the output of G is the output of v_5, which is the same as the output of v_4), and v_2-v_4 are the inner nodes. v_2, v_3 represent two convolution operations, each of which has 256 kernels of size (3×3). v_4 is an inner node denoting the concatenation operation.*

A computation graph is associated with a cost to measure its runtime efficiency. The cost function is $cost : \mathcal{G} \to \mathbb{R}^+$, where $\mathcal{G}$ is the set of all computation graphs and $\mathbb{R}^+$ is the positive real number set. We assume that the cost of a computation graph G can be calculated in $O(|G|)$ time, where $|G| = |V| + |E|$. In this paper, we define the cost of a node to be the time of computing its outputs, and the cost of a computation graph is measured by the total cost of all the nodes in it. The cost function can also be defined using other metrics, e.g., the total number of FLOPs of computing the outputs of all nodes, which are all feasible in our solutions.

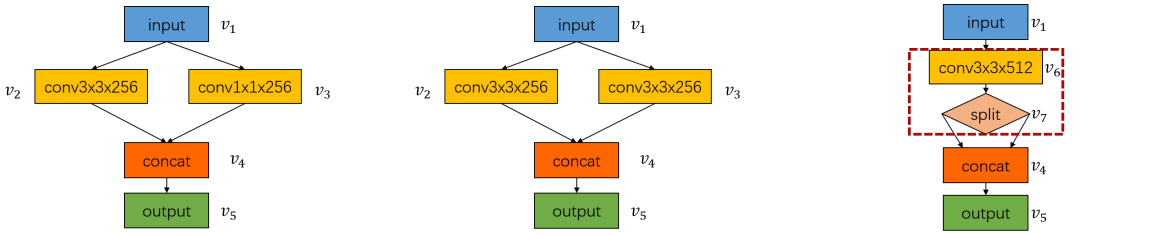
2.2 Definitions and Problem

We first define the substitution rules which specify how to optimize a computation graph by replacing its subgraph with a functionally equivalent one.

DEFINITION 1 (SUBSTITUTION RULE ϕ). *A substitution rule is denoted by $\phi = (G_s, G_t, f)$. G_s and G_t are both computation graphs, where G_s is the source graph and G_t is the target graph. f is a bijective function from the input and output nodes of G_s to the input and output nodes of G_t. G_s and G_t should always compute the same outputs given the same inputs w.r.t f, which is known as functionally equivalent.*

In this paper, we assume the rules are correct (i.e., the source graph and target graph of a rule are functionally equivalent) and given in advance. How to discover useful and correct rules has been studied in [9].

EXAMPLE 2. *Figure 2 shows an example of substitution rule ϕ, which fuses two convolution nodes into one, without changing the outputs. The left computation graph is the source graph G_s of ϕ, while the right one is the target graph G_t. v_1 is the input node of G_s, and v_4, v_5 are*



(a) Original computation graph. **(b)** Intermediate computation graph **(c)** Final computation graph.

Figure 1: An example of optimizing computation graphs using a sequence of graph substitutions from [10].

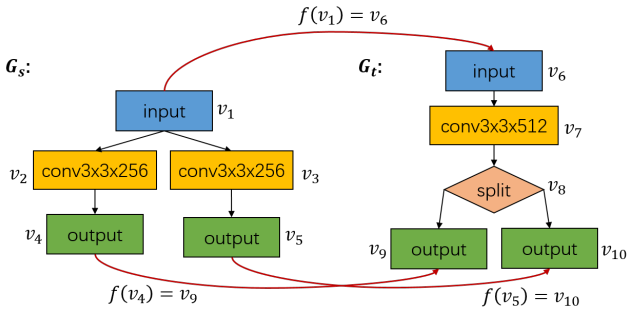


Figure 2: An example substitution rule.

the output nodes of G_s. v_6 is the input node of G_t, and v_9, v_{10} are the output nodes of G_t. The bijection f satisfies: $f(v_1) = v_6, f(v_4) = v_9, f(v_5) = v_{10}$. Given the same value of v_1 and v_6, the output of v_4 is the same as that of v_9, and the outputs of v_5 and v_{10} are equivalent.

The source and target graph of a substitution rule ϕ are functionally equivalent. If we find a subgraph of the computation graph G that is isomorphic to the source graph in ϕ and replace it with a new one according to the target graph in ϕ, the outputs of G will not be changed. Following this intuition, we formally define the graph substitution below.

Definition 2 (Graph Substitution τ). *Given a computation graph $G = (V, E)$ and a substitution rule $\phi = (G_s, G_t, f)$, graph substitution τ is denoted by (ϕ, g, h), where $g : V_s \to V_m$ is a bijection from the inner node set V_s of G_s to the node set $V_m \subseteq V$ satisfying:*
(1) for $v \in V_s$ and $v' = g(v)$, the node types (e.g., convolution operation) and parameters of v and v' are the same;
(2) the subgraph $G_s[V_s]$ of G_s induced by V_s, and the subgraph $G[V_m]$ of G induced by V_m are isomorphic;
(3) for each input node v_{in} of G_s, G has a node v'_{in} s.t. for any edge in G_s from v_{in} to an inner node v, G contains an edge from v'_{in} to $g(v)$; for any edge in G from a node $v' \in V_m$ to another node $u \notin V_m$, G_s involves an edge from $g^{-1}(v')$ to an output node v_{out}.
$h : V_t \to V'_m$ is a bijection from the inner node set V_t of G_t to a new node set V'_m that replaces nodes in V_m following f.

Example 3. *We use the substitution rule ϕ in Figure 2 to optimize the computation graph G in Figure 1b. We find that v_2 and v_3 in G can match v_2 and v_3 in G_s of ϕ respectively. For the input node v_1 in G_s, we can find v_1 in G such that for the edges (v_1, v_2) and (v_1, v_3) in G_s, G contains edges (v_1, v_2) and (v_1, v_3) to match them respectively. For edges $(v_2, v_4), (v_3, v_4)$ in G, G_s has the corresponding edges $(v_2, v_4), (v_3, v_5)$ ended at output nodes. Therefore, by replacing two convolution nodes (v_2 and v_3) in G with a convolution node with 512 kernels of size (3×3) (v_6) and a split node (v_7), we get the optimized computation graph G' in Figure 1c. Edge (v_6, v_7) is generated according to (v_7, v_8) in the target graph G_t of ϕ. The dotted rectangle in G' contains the new subgraph. According to f in ϕ, the external edges (v_1, v_2) and (v_1, v_3) in G are replaced by the new edge (v_1, v_6) in G', while (v_2, v_4) and (v_3, v_4) are replaced by two new edges from v_7 to v_4 in G'. Hence, this graph substitution is denoted by $\tau = (\phi, g, h)$, where g maps v_2 in G_s to v_2 in G, and v_3 in G_s to v_3 in G, and h maps v_7 in G_t to v_6 in G', and v_8 in G_t to v_7 in G'.*

We can optimize a computation graph by performing a sequence of graph substitutions, which is defined as follows.

Definition 3 (Graph Substitution Sequence P). *Given a computation graph G, a graph substitution sequence with length k is denoted by $P = (\tau_1, ..., \tau_k)$, where τ_i is the i-th graph substitution, and the graph substitutions in P are applied to G sequentially.*

Different from the prior work [9, 10], we set a length constraint for graph substitution sequences in this paper, in order to limit the search space of all possible graph substitution sequences. Now we formally define our problem.

Definition 4 (OCGGS Problem Statement). *Given an initial computation graph G, a set of substitution rules Φ, an integer K, and a cost function cost : $\mathcal{G} \to \mathbb{R}^+$, the problem of Optimizing Computation Graph using Graph Substitutions (OCGGS) is to find a graph substitution sequence P with length no more than K such that the cost of the optimized graph G' after applying P is minimized.*

Next we give the theorem to show the NP-hardness of the OCGGS problem.

Theorem 1. *The OCGGS problem is NP-hard.*

Proof. The maximum flow problem with conflict graph (MFCG) [12] is NP-hard on networks consisting of disjoint paths between two nodes v_b and v_e, even if the conflict graph is a 2-ladder (a 2-ladder is an undirected graph whose components are paths of length one). An instance of the MFCG problem is defined as follows. Given a directed graph H with a source node v_b and a sink node v_e, there are several disjoint paths from v_b to v_e. Each edge in the graph has a positive integer capacity. There are some conflicting edge pairs, denoted by the set Ψ, such that at most one edge

in the pair can carry flow at the same time. The decision problem of MFCG is to decide whether the total flow from v_b to v_e can be at least Θ. We construct an instance of the OCGGS problem from the above MFCG instance as follows.

• Nodes in H correspond to inner nodes in the computation graph G. Each inner node has a unique type without parameters and always outputs 1 regardless of its input. All the edges in H correspond to all the edges between inner nodes in G. We additionally create an input node v_{in} and an output node v_{out} for G, adding two edges: (v_{in}, v_b), (v_e, v_{out}).

• For every conflicting pair $(a = (v_1, v_2), \bar{a} = (v_3, v_4)) \in \Psi$, we construct 2 rules, ϕ and $\bar{\phi}$. G_s of ϕ contains nodes $\{v_1, v_2, v_3, v_4, v_{in_1}, v_{in_2}, v_{out_1}, v_{out_2}\}$, where v_{in_1}, v_{in_2} are two input nodes and v_{out_1}, v_{out_2} are two output nodes. G_s has edges $a, \bar{a}$, and four additional edges, (v_{in_1}, v_1),(v_{in_2}, v_3), (v_2, v_{out_1}), (v_4, v_{out_2}). G_t of ϕ is the same as G_s of it, except that it deletes edge a and then replaces v_2 with a node v_2' of constant value 1 (i.e., edge (v_2, v_{out_1}) becomes (v_2', v_{out_1})). f of ϕ maps v_{in_1}, v_{in_2} of G_s to v_{in_1}, v_{in_2} of G_t respectively, and v_{out_1}, v_{out_2} of G_s to v_{out_1}, v_{out_2} of G_t respectively. For $\bar{\phi}$, G_s and f of it are the same as those of ϕ. G_t of $\bar{\phi}$ is the same as G_s of it, except that it deletes edge $\bar{a}$ and then replaces v_4 with a node v_4' of constant value 1 (i.e., edge (v_4, v_{out_2}) becomes (v_4', v_{out_2})).

• The length constraint K is set to $|\Psi|$.

• The cost function $cost : \mathcal{G} \to \mathbb{R}^+$ is the following. The cost of a computation graph is positive infinity if two edges corresponding to a pair in Ψ both exist in it. Otherwise, the cost is $\frac{1}{F}$, where F is the maximum flow on the computation graph. To compute the maximum flow, the cost function specifies the capacity of each edge as the capacity of the corresponding edge in H (for a node v' which replaces an original inner node v, the capacity of edge (v', u) is the same as that of (v, u) in H), and the capacity of (v_{in}, v_b), (v_e, v_{out}) as positive infinity. The cost can be calculated in polynomial time by checking conflicting edge pairs and summing over the maximum flows of all paths.

The above construction can be done in $O(|H| + |\Psi|)$ time. The decision problem of the OCGGS problem is to decide whether there is a graph substitution sequence no longer than K such that the cost of the optimized computation graph is no more than $\frac{1}{\Theta}$. It is easy to see that the OCGGS instance is YES if and only if the MFCG instance is YES, and the decision problem of OCGGS is in NP. Therefore, the OCGGS problem is NP-hard. □

Further, no polynomial time approximation algorithm exists for the OCGGS problem due to the following theorem.

THEOREM 2. *OCGGS is Poly-APX-complete.*

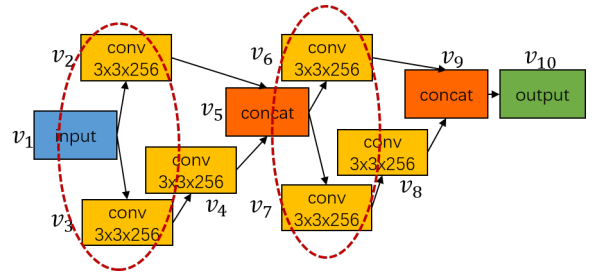
PROOF. MFCG on networks consisting of disjoint paths between v_b and v_e with conflict graph being a 2-ladder is Poly-APX-complete. If there is an α-approximation algorithm to the OCGGS problem, we can find a $\frac{1}{\alpha}$-approximation algorithm to the MFCG problem, which contradicts the fact that it is Poly-APX-complete. Therefore, the OCGGS problem is also Poly-APX-complete. □

3. PRUNING-BASED ALGORITHM

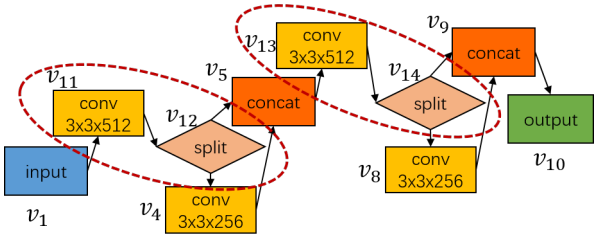
In this section, we introduce our pruning-based algorithm. The main idea is to find equivalent substitution sequences and eliminate examining the redundant ones. We define two substitution sequences are equivalent as follows:

DEFINITION 5 (EQUIVALENT SEQUENCES). *Given a computation graph G, and two substitution sequences P, P', we call P and P' are equivalent, if the final optimized computation graphs by applying P and P' are the same.*

To find equivalent substitution sequences, we define a partial order over graph substitutions in a sequence. This order ensures that all graph substitution sequences can be reordered (sorted) without changing their optimization effect on the computation graph G. Therefore, for any unordered graph substitution sequence, there always exists an equivalent ordered graph substitution sequence. To this end, we can prune the graph substitution sequences which are out of order and only check the ordered ones.



(a) Original computation graph.



(b) Optimized computation graph.

Figure 3: An illustration of equivalent substitution sequences.

EXAMPLE 4. *The computation graph G in Figure 3a has 6 convolution operations (each with 256 kernels of size (3×3)), 2 concatenation operations, an input and an output. Suppose we are given only the rule in Figure 2, which fuses two convolutions into one. Then there are two graph substitution sequences of length 2, denoted by P_1 and P_2. P_1 first fuses v_2 and v_3, and then fuses v_6 and v_7, while P_2 first fuses v_6 and v_7, and then fuses v_2 and v_3. The final optimized computation graphs by applying by P_1 and P_2 respectively are the same (as shown in Figure 3b). Therefore, we only need to check P_1 or P_2, instead of both.*

Since DNNs are generally large, there can be a number of equivalent substitution sequences. This calls for a pruning rule to avoid examining equivalent substitution sequences.

3.1 Partial Order

In the following, we first introduce the label for a node in any computation graph. Then we discuss how to order two graph substitutions in a graph substitution sequence using the node labels, and prove the order is a partial order, and finally present the pruning rule.

The node label is defined in the substitution sequence context, which consists of two parts: (1) the node is generated

by which graph substitution in the sequence; and (2) the node is generated by which node in the target graph of the substitution rule. Since the source and target graphs of a substitution rule are both computation graphs, we label nodes in them as well. For convenience, we define a dummy substitution N. The nodes in source/target graphs of substitution rules, or in the initial computation graph can be regarded as being generated by the dummy substitution N.

For ease of understanding, we define the labels of nodes (1) in source/target graphs of substitution rules and (2) in initial/optimized computation graphs separately.

Definition 6 (Node Label). *(1) Given a source/target graph G of a substitution rule, the label of a node v in G is $\Gamma_v = (N, j)$. N is the dummy substitution, and j is any integer s.t. every two nodes in G have different values of j.*

(2) Given a computation graph G_0 and a graph substitution sequence $P = (\tau_1, ..., \tau_k)$, let G_i be the intermediate computation graph optimized by applying substitutions from τ_1 to τ_i. The label of a node v in G_0 is $\Gamma_v = (N, j)$, as defined in (1). For a node v in G_i $(i = 1, ..., k)$, if v is also in G_{i-1}, the label of v is the same as that in G_{i-1}. Otherwise, v is generated by $\tau_i = (\phi_i, g_i, h_i)$. Supposing $h_i^{-1}(v) = v_t$ and label of v_t is $\Gamma_{v_t} = (N, l)$, the label of v is $\Gamma_v = (\tau_i, l)$.

To order substitutions in a sequence, we first define the dependence between substitutions as follows:

Definition 7 (Substitution Dependence). *Given a substitution sequence $P = (\tau_1, ..., \tau_k)$ and an initial computation graph G_0, we define that τ_i in P depends on τ_l (τ_l is N or is in P), if τ_i replaces a node v and the label of v is $\Gamma_v = (\tau_l, j)$ (i.e., v is generated by τ_l).*

Since the label of a node contains the information of which substitution generates it, for a substitution, we know all the substitutions it depends on from the labels of the nodes it replaces. Therefore, we can compare two substitutions in the same sequence by the respective latest substitutions they depend on. The comparison is done recursively. Following this intuition, we formally define the order over substitutions in a sequence as follows.

Definition 8 (Order over Graph Substitutions). *Given a computation graph G and a graph substitution sequence $P = (\tau_1, ..., \tau_k)$, the order between two graph substitutions in P is defined in a recursive way:*

(1) For any substitution τ_i in P, we have $N \preceq \tau_i$.

(2) For any two substitutions τ_i and τ_j in P ($\tau_i, \tau_j \neq N$), let V_i, V_j be the respective set of nodes τ_i and τ_j replace, Δ_i and Δ_j be the respective sets of substitutions they depend on, and the latest substitutions of Δ_i and Δ_j be τ_m and τ_n. That is, for $\forall \tau_l \in \Delta_i$, $\tau_l \preceq \tau_m$, and for $\forall \tau_l \in \Delta_j$, $\tau_l \preceq \tau_n$. Let $\alpha_i = \max_{v \in V_i, \Gamma_v = (\tau_m, j_v)} j_v$, $\alpha_j = \max_{v \in V_j, \Gamma_v = (\tau_n, j_v)} j_v$. We have $\tau_i \preceq \tau_j$ iff (a) $\tau_m \preceq \tau_n$, but $\tau_n \not\preceq \tau_m$; or (b) $\tau_m = \tau_n$, and $\alpha_i \leq \alpha_j$.

Below we use an example to explain the node label and the substitution order.

Example 5. *Consider the example in Figure 3. ϕ denotes the substitution rule in Figure 2. The label of node $v_i (i = 1, ..., 10)$ in the original computation graph (denoted by G) can be initialized as $\Gamma_{v_i} = (N, i)$, and the label of node $v_j (j = 6, ..., 10)$ in G_t of ϕ can be initialized as $\Gamma'_{v_j} = (N, j)$. Suppose we fuse v_2 and v_3 in G first, and then v_6 and v_7. Let τ_1, τ_2 denote the two graph substitutions, so the substitution sequence is $P = (\tau_1, \tau_2)$. Then the label of $v_{11}, v_{12}, v_{13}, v_{14}$ in the optimized computation graph are $\Gamma_{v_{11}} = (\tau_1, 7), \Gamma_{v_{12}} = (\tau_1, 8), \Gamma_{v_{13}} = (\tau_2, 7), \Gamma_{v_{14}} = (\tau_2, 8)$ respectively. To compare τ_1 and τ_2, we first find the substitutions they depend on. Since τ_1 replaces v_2 and v_3, which are "generated" by N, τ_1 only depends on N. Similarly, since τ_2 replaces v_6 and v_7, it only depends on N as well. Therefore, we need to check the second condition in Definition 8 to compare τ_1 and τ_2. The respective α values in Definition 8 for τ_1 and τ_2 are 3 and 7, and $3 \leq 7$, therefore, $\tau_1 \preceq \tau_2$.*

Theorem 3. *The order over graph substitutions defined in Definition 8 is a partial order.*

Proof. By mathematical induction, we can prove the order is a partial order. Appendix A shows more details. □

Based on the substitution order, we say a graph substitution sequence $P = (\tau_1, ..., \tau_k)$ is *ordered* if $\tau_1 \preceq ... \preceq \tau_k$.

3.2 Pruning Rule

Given the property that a graph substitution sequence can be sorted using the partial order over graph substitutions, we prove that sorting a graph substitution sequence will not change its optimization effect. For convenience, we define the child-father relationship of graph substitution sequences, the child substitution of a substitution sequence and the substitution sequence descendant below.

Definition 9 (child-father relationship). *Given a computation graph G and graph substitution sequences, $P = (\tau_1, ..., \tau_k)$, $P' = (\tau_1, ..., \tau_k, \tau'_{k+1})$, we dub P' as a child graph substitution sequence of P and P as a father graph substitution sequence of P'. If P is empty and $P' = (\tau'_1)$, the child-father relationship still holds for P and P'.*

Definition 10 (child substitution). *Given two substitution sequences P and P', if P' is a child graph substitution sequence of P, we say the last substitution in P' is a child graph substitution of P.*

Definition 11 (sequence descendant). *Given a computation graph G and a graph substitution sequence $P = (\tau_1, ..., \tau_k)$, any graph substitution sequence P' for G satisfying that $P' = (\tau_1, ..., \tau_k, \tau'_{k+1}, ..., \tau'_{k'})$ $(k' > k)$ is a descendant of P.*

Theorem 4. *Given a computation graph G and a graph substitution sequence P_1, let P_2 be the substitution sequence after we sort P_1 according to the order over graph substitutions (P_2 is ordered and unique), then the computation graphs optimized by P_1 and P_2 respectively are the same.*

Proof. To prove Theorem 4, we first define the swapping operation for graph substitutions: for two adjacent graph substitutions τ_1, τ_2 in a graph substitution sequence P satisfying that τ_1 is ahead of τ_2 but $\tau_2 \preceq \tau_1$, the swapping operation exchanges τ_1 and τ_2 in P. Then we can prove Theorem 4 easily by showing (1) the swapping operation will not change the optimization effect of a graph substitution sequence, and (2) we can sort a graph substitution sequence by doing swapping operations sequencially. □

Theorem 4 shows the effect-preserving property of sorting. We now provide the pruning rule as follows.

COROLLARY 1 (PRUNING RULE). *Given a computation graph G and a graph substitution sequence P, if P is not ordered, by Theorem 4, there exists an ordered P' equivalent to P, and therefore, we only need to check P' and its descendants, instead of checking P and descendants of P, when searching the solution to the OCGGS problem.*

The algorithm using the pruning rule to solve the OCGGS problem is described in Algorithm 1. When we first invoke Algorithm 1, the input substitution sequence P is an empty sequence. For a substitution sequence P, if its length is K, we do not search its descendants (line 2). Otherwise, the descendant set of P is the union of the descendant sets of all its child substitution sequences (line 9). For each child substitution τ of P, we check whether the last substitution of P (if any) $\preceq \tau$ (line 6). If so, we append τ to P (line 7); otherwise we ignore τ. As such, we only keep ordered graph substitution sequences and prune unordered ones.

Algorithm 1: Pruning

Input: A computation graph $G = (V, E)$, a set of rules Φ, an integer K, a cost function $cost : \mathcal{G} \to \mathbb{R}^+$, an initial substitution sequence P.
Output: An optimal graph substitution sequence, the cost of the graph optimized by the sequence.

```
1: P_best ← P, C_best ← cost(G)
2: if K = 0 then
3:    return P_best, C_best
4: for φ_j ∈ Φ do
5:    for every graph substitution τ using φ_j on G do
6:       if P is empty or the last substitution τ* of P ⪯ τ then
7:          P' ← a new substitution sequence by adding τ after P
8:          G' ← apply τ to G
9:          P'_best, C'_best ← Pruning(G', Φ, K − 1, cost, P')
10:         if C'_best < C_best then
11:            P_best ← P'_best, C_best ← C'_best
12: return P_best, C_best
```

Time Complexity. Suppose there are $\mathcal{T}$ ordered graph substitution sequences checked by Algorithm 1. Assume the degree of each node in the computation graphs (including the source/target graphs of rules) is $O(D)$, and the number of nodes in the source/target graph of a rule is $O(S)$. If we regard D and S as small constants (like 10 for D in NasNet-A and 4 for S in our experiments), then after K graph substitutions, there are $O(|V|+K)$ nodes in the computation graph, where $|V|$ is the number of nodes in the original computation graph. There can be $O(|V| + K)$ graph substitutions to be checked in line 5 of Algorithm 1 (if the source graph of any rule is weakly connected), and we need $O(|V| + K)$ time to find them all. Line 6 costs $O(1)$ time to get the comparison result. Applying P' to G (line 8) and calculating the cost of a computation graph (line 1) require $O(|E| + K)$ time. In total, Algorithm 1 takes $O(\mathcal{T}|\Phi|(|V| + K) + \mathcal{T}(|E| + K))$ time, where $|\Phi|$ is the number of rules. If the node number and the edge number in the computation graph are bounded by $O(|V|)$ and $O(|E|)$ respectively due to the given rules in the substituting process, the overall time complexity is $O(\mathcal{T}(|\Phi||V| + |E|))$. If the source graphs in the rules are not weakly connected graphs, the time complexity based on the aforementioned assumptions is $O(\mathcal{T}(|\Phi||V|^S + |E|))$.

Space Complexity. Algorithm 1 needs to store $O(K)$ substitution sequences (each needs $O(K)$ space) and $O(K)$ computation graphs in the search process at the same time. Similar to the analysis of time complexity, if the node number and the edge number are bounded by $O(|V|)$ and $O(|E|)$, the space complexity of Pruning is $O(|\Phi|+K(|V|+|E|+K))$, which is polynomial in terms of the input size and K.

4. DYNAMIC PROGRAMMING

While the pruning-based algorithm can avoid examining redundant graph substitution sequences effectively, performing subgraph matching during the search of graph substitutions is still costly. The main idea of the dynamic programming algorithm is to store the already explored substitutions and reuse them when we are looking for the child substitutions of a substitution sequence. In this way, we can avoid the subgraph matching cost of the reused substitutions.

To find reusable substitutions, we observe that, given a computation graph G and the set Δ of all the possible graph substitutions on G, there may exist two substitutions $\tau_1, \tau_2 \in \Delta$ that replace different nodes in G without interfering with each other. Suppose we apply τ_1 on G, i.e., the substitution sequence is $P = (\tau_1)$. We are sure that τ_2 is one child substitution of P, and hence the sequence $P' = (\tau_1, \tau_2)$ can be applied on G sequentially. In the following, we provide a concrete example to illustrate the above observation.

EXAMPLE 6. *Recall the example in Figure 3. Given the rule of fusing two convolutions, we can find two graph substitutions τ_1 and τ_2 at first (τ_1: fusing v_2, v_3; τ_2: fusing v_6, v_7), and each forms a substitution sequence of length 1, denoted by $P_1 = (\tau_1)$ and $P_2 = (\tau_2)$. After applying P_1, τ_2 can still be applied, leading to a substitution sequence $P_3 = (\tau_1, \tau_2)$ of length 2. Since P_2 and P_3 both have τ_2, we can make use of the matching result of τ_2 in P_2 when trying to find P_3.*

4.1 Dynamic Programming Algorithm

To search all possible graph substitution sequences for a computation graph, the dynamic programming algorithm iteratively appends substitutions to the existing substitution sequences. Before we present the state transition rules, for convenience, we first define some variables.

Given a computation graph G, a graph substitution sequence P_1 and its father graph substitution sequence P_2, let Δ_1, Δ_2 be the respective child substitution sets in P_1 and P_2. τ^* denotes the last substitution of P_1, and hence $\tau^* \in \Delta_2$. Let $\Delta_3 \subseteq \Delta_2$ be the set of all the child substitutions of P_2 which replace nodes different from the ones replaced by τ^*. Let $\Delta_4 \subseteq \Delta_1$ be the set of all the child substitutions of P_1 which depend on τ^*. $\mathcal{B} : (P, \Delta) \to \Delta$ is a function which takes a substitution sequence P and a set of substitutions Δ as input, and outputs the substitutions such that appending each of them to P leads to an ordered sequence.

We define the state of a substitution sequence as its child substitution set. Theorem 5 shows the state transition rules.

THEOREM 5 (STATE TRANSITION RULES). *Given an initial computation graph G, for a substitution sequence P_1:*

(1) if P_1 is empty, its child substitutions are searched directly on G;

(2) if P_1 is not empty, whose father substitution sequence is P_2, then its child substitution set $\Delta_1 = \Delta_3 \cup \Delta_4$ (the variables have the same meaning as aforementioned).

It is easy to prove Theorem 5. After we find the child substitutions of a substitution sequence, we can append them to the sequence to get new substitution sequences. Therefore, we can use dynamic programming based on Theorem 5 to find all possible graph substitution sequences for a computation graph. Furthermore, the following theorem shows the pruning rule in Corollary 1 can be combined with the dynamic programming method. The correctness of Theorem 6 is guaranteed by the transitivity of the partial order over graph substitutions.

THEOREM 6. *Given an initial computation graph G, for a substitution sequence P_1:*

(1) if P_1 is empty, its child substitutions are searched directly on G;

(2) if P_1 is not empty, whose father substitution sequence is P_2, then the set of its child substitutions τ' such that $\tau^* \preceq \tau'$ is $\mathcal{B}(P_1, \Delta_1) = \mathcal{B}(P_1, \mathcal{B}(P_2, \Delta_3)) \cup \Delta_4$ (the variables have the same meaning as aforementioned).

Algorithm 2 provides the pseudocode of DP with pruning according to Theorem 6. For the first time of invoking Algorithm 2, the input substitution sequence P is an empty sequence and the input substitution set Δ is $\emptyset$. If a substitution sequence P is of length K, we do not search its child substitution sequences (line 2). Otherwise, we search the two parts of its child substitutions according to Theorem 6 and store them in Δ' (line 6, 11). By appending the child substitutions to P, we get its child substitutions sequences (line 13). The descendant set of P is the union of the descendant sets of all its child substitution sequences (line 15).

Time Complexity. Suppose there are $\mathcal{T}$ ordered graph substitution sequences being checked. Let $\mathcal{T}'$ be $\sum_{\Delta \in \mathbb{S}} |\Delta|^2$, where $\mathbb{S}$ is the set of child substitution sets Δ of all ordered substitution sequences ($\mathcal{T}' > \mathcal{T}$). Other variables used in this analysis are the same as those in the analysis of Algorithm 1. We also regard D and S as constants. Line 6 takes $O(\mathcal{T}')$ time in total, There can be $O(1)$ (if τ^* exists) or $O(|V|)$ (otherwise) graph substitutions in line 10 (if the source graph of any rule is weakly connected), each taking $O(1)$ time to be checked whether it can be matched. Line 1 and line 14 take $O(|E| + K)$ time, where $|E|$ is the number of edges in the original computation graph. Therefore, the total time complexity of Algorithm 2 is $O(\mathcal{T}' + |\Phi||V| + \mathcal{T}(|\Phi| + |E| + K))$. If the edge number is bounded by $O(|E|)$ during the substituting process, the overall time complexity is $O(\mathcal{T}' + |\Phi||V| + \mathcal{T}(|\Phi| + |E|))$, and the time complexity is $O(\mathcal{T}' + |\Phi||V|^S + \mathcal{T}(|\Phi||V|^{S-1} + |E|))$ when the source graphs of rules are not weakly connected graphs.

Space Complexity. Algorithm 2 needs to store $O(K\mathcal{U})$ substitutions (each needs $O(1)$ space), $O(K)$ substitution sequences and $O(K)$ computation graphs in the process at the same time, where $\mathcal{U}$ is the maximum size of a child substitution set of a sequence and is polynomial in terms of the input size according to the time complexity analysis. If the node number and the edge number are bounded by $O(|V|)$ and $O(|E|)$ in the substituting process, the space complexity of DPP is $O(|\Phi| + K(\mathcal{U} + K + |V| + |E|))$, which is polynomial in terms of the input size and K.

Algorithm 2: DP with pruning (DPP)

Input: A computation graph $G = (V, E)$, a set of rules Φ, an integer K, a cost function $cost : \mathcal{G} \to \mathbb{R}^+$, an initial substitution sequence P, the child substitution set Δ of the father substitution sequence of P.

Output: An optimal graph substitution sequence, the cost of the graph optimized by the sequence.

1: $P_{best} \leftarrow P$, $C_{best} \leftarrow cost(G)$
2: **if** $K = 0$ **then**
3: **return** P_{best}, C_{best}
4: **if** P is not empty **then**
5: τ^* denotes the last graph substitution of P
6: $\Delta' \leftarrow$ reusable substitutions τ from Δ according to Section 4.1 s.t. $\tau^* \preceq \tau$
7: **else**
8: $\Delta' \leftarrow \emptyset$, $\tau^* \leftarrow$ None
9: **for** $\phi_j \in \Phi$ **do**
10: **for** every graph substitution τ using ϕ_j on G and depending on τ^* (if $\tau^* \neq$ None) **do**
11: $\Delta' = \Delta' \cup \{\tau\}$
12: **for** $\tau \in \Delta'$ **do**
13: $P' \leftarrow$ a new sequence by adding τ after P
14: $G' \leftarrow$ apply τ to G
15: $P'_{best}, C'_{best} \leftarrow \text{DPP}(G', \Phi, K - 1, cost : \mathcal{G} \to \mathbb{R}^+, P', \Delta')$
16: **if** $C'_{best} < C_{best}$ **then**
17: $P_{best} \leftarrow P'_{best}$, $C_{best} \leftarrow C'_{best}$
18: **return** P_{best}, C_{best}

5. SAMPLING-BASED APPROXIMATION ALGORITHM

Algorithm 1 and 2 provide exact solutions. They fully explore the search space of all possible graph substitution sequences, and their time complexity is determined by the total number of ordered graph substitution sequences, which can be exponential in terms of the input size. To improve the search efficiency by sacrificing little accuracy, we develop a sampling-based approximation algorithm to the OCGGS problem with polynomial time and space complexity.

The pseudocode is illustrated in Algorithm 3. Overall, it starts with an empty graph substitution sequence and tries to find the best substitution sequence iteratively until we cannot find any feasible substitution sequences ($P_{k-1} = \emptyset$) due to the length constraint or the sampling rule described below (line 3). During the k-th iteration, we take no more than Q already sampled substitution sequences in set $\mathcal{P}_{k-1}$ as input, where Q is the given sample size. We search all the child substitution sequences with length no longer than K for the sequences in $\mathcal{P}_{k-1}$ and keep them in $\mathcal{P}_{tmp}$ (line 4). This step is accomplished using the dynamic programming technique (line 4-11 of Algorithm 2 but without order checking). We then evaluate the sequences in $\mathcal{P}_{tmp}$ and some of their descendants to obtain $\mathcal{P}_k$ via sampling (line 5-14). For all the iterations, we use P_{best} to maintain the best graph substitution sequence being searched so far that has the lowest cost C_{best} of the optimized computation graph.

The challenge for the sampling-based approximation algorithm is how to effectively sample possibly good substitution sequences without enumerating all the sequences. A simple heuristic is to append one graph substitution each time to the sequence that leads to maximal reduction in the cost.

However, while an optimal sequence can derive a computation graph with lowest cost, the involved graph substitutions may not always decrease the cost. Recall the example in Figure 1. By first enlarging the kernel size of one convolution node and then merging two convolution nodes into one, the cost of the computation graph increases before eventually decreases. The search and sampling heuristic should grant credits to such sequences rather than greedily pursue the sequences that reduce the cost continuously. Since the source graph of any substitution rule typically has a small number of nodes, a reasonable assumption is that for any graph substitution in a sequence, the number of the graph substitutions it depends on in the sequence that increase the cost cannot be very large (like 1 in the example of Figure 1).

To introduce the sampling rule, we first define further exploration sequences as a graph substitution sequence that needs further exploration, and the potential of a further exploration substitution sequence. For a graph substitution sequence $P = (\tau_1, ..., \tau_k)$, we dub P as a *further exploration substitution sequence* if (1) τ_k increases the computation graph cost, and (2) P does not have more than η successive cost-increasing substitutions, where η is a given constant (the value of η is selected based on the assumption mentioned above).

DEFINITION 12 (POTENTIAL). *Given a further exploration substitution sequence $P = (\tau_1, ..., \tau_k)$, we find all its descendants $P' = (\tau_1, ..., \tau_k, \tau'_{k+1}, ..., \tau'_{k'})$ such that:*
(1) τ_i depends on τ_{i-1}, for $i = k+1, ..., k'$;
(2) $\tau'_{k'}$ decreases the computation graph cost;
(3) $P'' = (\tau_1, ..., \tau_k, \tau'_{k+1}, ..., \tau'_{k'-1})$ is a further exploration substitution sequence.
The potential of P is the lowest cost of the optimized computation graphs by applying the above descendants.

We propose to perform search and sampling in the following way. In the k-th iteration of Algorithm 3, we divide $\mathcal{P}_{tmp}$ into two parts, $\mathcal{P}_{des}$ and $\mathcal{P}_{inc}$. $\mathcal{P}_{inc}$ contains further exploration sequences (line 6), and $\mathcal{P}_{des}$ is $\mathcal{P}_{tmp} - \mathcal{P}_{inc}$ (line 7). For $\mathcal{P}_{des}$, we select $Q/2$ sequences with the best optimized computation graph costs and add them into $\mathcal{P}_k$ to be evaluated in the next iteration (line 14), because their descendants are very likely to include the optimal graph substitution sequence. For all the sequences in $\mathcal{P}_{inc}$, we do not sample them directly but search their descendants and sample the descendants (line 8-12). Specifically, we first assign $\mathcal{P}_{inc}$ to $\mathcal{P}$ (line 6), which contains further exploration sequences. We then run in iterations until there is no further exploration sequences left (line 8). In each iteration, we select $Q/2$ sequences from $\mathcal{P}$ with the best potentials (line 9), search the child substitution sequences of the sampled ones and store them in $\mathcal{P}$ (line 10, using line 9-11 of Algorithm 2), add $\mathcal{P}$ to $\mathcal{P}_{inc}$ (line 11), and then only keep the further exploration sequences in $\mathcal{P}$ (line 12). For the new $\mathcal{P}_{inc}$, we delete sequences in it ending with a cost-increasing substitution and sample $Q/2$ of them with the best optimized computation graph costs to add to $\mathcal{P}_k$.

Time Complexity. Similar to the analysis of Algorithm 2, we regard D and S as constants. For the simplicity of expression, let $\mathcal{H}$ denote $(Q|\Phi||V|)$, $\mathcal{H}'$ denote $(Q|\Phi||V|^S)$, and $\mathcal{M}$ denote $(QK|\Phi|)$. If the source graph of any rule is weakly connected, line 4-5 in Algorithm 3 take $O(\mathcal{M}(|V|+K)(|E|+K))$ time in total. Line 8-12 take $O(K|\mathcal{P}|\log|\mathcal{P}|+\mathcal{M}(|E|+K))$ time, and line 14 takes $O(K|\mathcal{P}|\log|\mathcal{P}|)$ time,

Algorithm 3: Sampling

Input: A computation graph $G = (V, E)$, a set of rules Φ, an integer K, a cost function $cost : \mathcal{G} \to \mathbb{R}^+$, a sample size Q, a constant η for further exploration need check.
Output: An optimal graph substitution sequence.
1: $P_{best} \leftarrow$ None, $C_{best} \leftarrow cost(G)$
2: $\mathcal{P}_0 \leftarrow \{P_{best}\}$, $k \leftarrow 1$
3: **while** $\mathcal{P}_{k-1} \neq \emptyset$ **do**
4: $\quad\mathcal{P}_{tmp} \leftarrow$ all child substitution sequences of $\mathcal{P}_{k-1}$ no longer than K
5: $\quad$update P_{best}, C_{best}.
6: $\quad\mathcal{P}, \mathcal{P}_{inc} \leftarrow$ further exploration sequences $\in \mathcal{P}_l$
7: $\quad\mathcal{P}_{des} \leftarrow \mathcal{P}_{tmp} - \mathcal{P}_{inc}$
8: $\quad$**while** $\mathcal{P} \neq \emptyset$ **do**
9: $\quad\quad\mathcal{P} \leftarrow$ sample $Q/2$ from $\mathcal{P}$ according to their potentials
10: $\quad\quad\mathcal{P} \leftarrow$ child substitution sequences of $\mathcal{P}$ no longer than K (line 9-11 of Algorithm 2)
11: $\quad\quad\mathcal{P}_{inc} \leftarrow \mathcal{P}_{inc} \cup \mathcal{P}$, update P_{best}, C_{best}.
12: $\quad\quad\mathcal{P} \leftarrow$ further exploration sequences in $\mathcal{P}$
13: $\quad$delete sequences ending with cost-increasing substitutions from $\mathcal{P}_{inc}$
14: $\quad\mathcal{P}_k \leftarrow$ sample $Q/2$ graph substitution sequences from $\mathcal{P}_{des}$ and $\mathcal{P}_{inc}$ respectively
15: $\quad K \leftarrow K + 1$
16: **return** P_{best}

Table 2: The details of the DNNs.

Model	Details
Inception-v3	11 blocks of 5 types, 138 operators
ResNet	8 blocks of 1 type, 40 operators
ResNext-50	16 blocks of 2 types, 86 operators
NasNet-A	18 blocks of 2 types, 293 operators
SRU	1 block of 1 type, 11 operators
NasRNN	5 blocks of 1 type, 230 operators
BERT	8 blocks of 1 type, 113 operators

where $|\mathcal{P}| = O(Q|\Phi|(|V|+K))$. So the total time complexity is $O(\mathcal{M}(|V|+K)(|E|+K+\log(Q|\Phi|(|V|+K))))$. If the numbers of nodes and edges are bounded by $O(|V|)$ and $O(|E|)$ respectively due to the given rules in the substituting process, the time complexity is $O(K\mathcal{H}(|E|+\log\mathcal{H}))$, and the time complexity is $O(K\mathcal{H}'(|E|+\log\mathcal{H}'))$ when the source graphs of rules are not weakly connected graphs.

Space Complexity. Similar to the analysis of time complexity, if the node number and the edge number are bounded by $O(|V|)$ and $O(|E|)$, the space complexity of Sampling is $O(|\Phi| + Q\mathcal{U}(K+|V|+|E|))$, where $\mathcal{U}$ is the maximum size of a child substitution set of a sequence and is polynomial in terms of the input size.

6. EXPERIMENTS

6.1 Experimental Setting

DNNs. We use 7 DNNs in the experiments, and the details of the models are provided in Table 2. **Inception-v3** [14] is a deep convolutional neural network (CNN) proposed for image classification. **ResNet** [8] is also a CNN and it can be very deep thanks to the residual connections. **ResNeXt-50** [17] explores the split-transform-merge strategy of Incep-

tion models and introduces a new grouped convolution operator to the original structure of ResNet. **NasNet-A** [20] is a state-of-the-art CNN model discovered by neural architecture search. **SRU** [11] is a recurrent neural network (RNN) architecture which simplifies the computation and exploits more parallelism compared with the conventional RNN architectures. **NasRNN** [19] is also an RNN discovered by neural architecture search and outperforms the widely-used LSTM. **BERT** [6] is a powerful language model which obtains the state-of-the-art results on many natural language processing tasks.

Substitution rules. We use the 157 substitution rules identified by TASO [9], whose correctness has been verified by TASO. Applying the rules may either increase or decrease the cost of a computation graph.

Comparison methods. We compare six algorithms in the experiments: (1) **Pruning** (Algorithm 1), (2) **DPP** (Algorithm 2), (3) **Sampling** (Algorithm 3), and three baselines, (4) the brute-force enumeration algorithm Enumeration (**ENU**), (5) MetaFlow (**MF**) [10], a cost-based backtracking search algorithm accelerated by using a graph splitting strategy, and (6) TASO (**TS**) [9], using the same cost-based backtracking search algorithm as MF without graph splitting. For Sampling, we set $Q = 20$ and $\eta = 1$. For MF and TS, the hyperparameter α is set to 1.05, which is the same as in [9, 10]. The threshold used in graph splitting of MF is 30, following the same setting as in [10]. For some complex models, the two exact algorithms and three baselines would run for more than hours or even days, which are much slower than Sampling that finishes in seconds. In the experiments, if an algorithm does not terminate within 1 hour, we kill the corresponding process. To provide baseline results in spite of the process kill, we report the optimized graph costs by the best substitution sequences MF and TS find, the search time, and the memory consumption when they are stopped. For Pruning, DPP and ENU we report their result as "-" if they cannot finish within 1 hour, because they are exact algorithms and only the results obtained when they terminate normally are meaningful.

Measurements. We measure the performance of different algorithms using three metrics: (1) the optimized cost, i.e., the cost of the computation graphs after optimization as mentioned in Section 2 (the cost of each operator is estimated by its execution time on a machine, which is also used in [10]); (2) the search time (algorithm running time); and (3) the memory consumption. All the experiments were repeated for 100 times, and we report the average results.

We implemented the algorithms in C++[1] on top of the code provided by Jia et al. [9], and conducted experiments on a Ubuntu 16.04.6 machine with a 12-core Intel E5-2690 CPU, 2 NVIDIA Tesla P100 GPUs and 220 GB of RAM.

6.2 Evaluation on Different DNNs

We evaluate the performance of all the algorithms over (1) different DNN architectures in Section 6.2.1 and (2) different DNN sizes in Section 6.2.2.

6.2.1 *Different DNN Architectures*

We consider 7 different DNN architectures listed in Table 2. The length constraint K is set to 10. Table 3 shows the results. Specifically, Table 3a shows the costs of the computation graphs optimized by the algorithms, Table 3b shows the search time for each algorithm, and Table 3c shows the memory consumption results of the algorithms.

[1]The code is available on https://github.com/Experiment-code/OCGGS.

Table 3: Results on different DNN architectures.

(a) Optimized cost.

DNN	Optimized cost (ms)					
	ENU	Pruning	DPP	MF	TS	Sampling
Inception-v3	-	-	-	9.48	**8.80**	**8.80**
ResNet	-	**1.73**	**1.73**	**1.73**	**1.73**	**1.73**
ResNeXt-50	-	-	-	8.54	**7.80**	**7.80**
NasNet-A	-	-	-	20.97	**18.31**	**18.31**
BERT	-	-	-	**1.21**	**1.21**	**1.21**
SRU	-	**0.15**	**0.15**	**0.15**	**0.15**	**0.15**
NasRNN	-	-	-	1.82	**1.80**	**1.80**

(b) Search time.

DNN	Search time (s)					
	ENU	Pruning	DPP	MF	TS	Sampling
Inception-v3	-	-	-	11.39	> 3600	**4.38**
ResNet	-	20.39	7.89	1.35	2.82	**0.26**
ResNeXt-50	-	-	-	5.25	108.83	**1.14**
NasNet-A	-	-	-	> 3600	> 3600	**233.34**
BERT	-	-	-	11.69	790.28	**2.02**
SRU	-	21.96	33.12	0.03	**0.02**	0.28
NasRNN	-	-	-	> 3600	> 3600	**20.59**

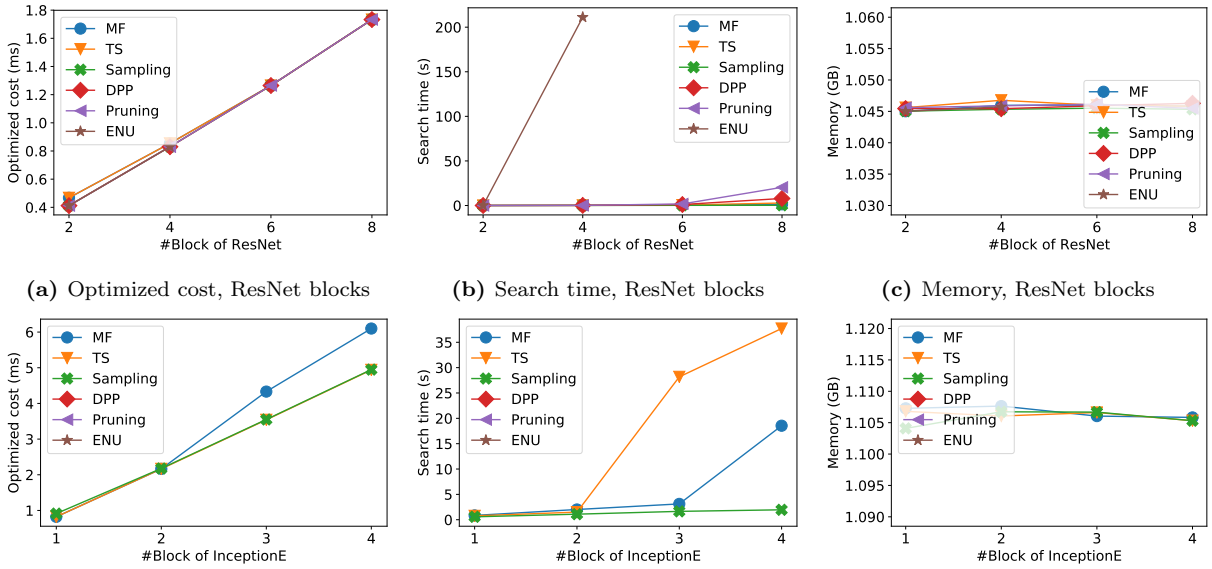
(c) Memory.

DNN	Memory (GB)					
	ENU	Pruning	DPP	MF	TS	Sampling
Inception-v3	-	-	-	**1.10**	> 1.30	1.15
ResNet	-	**1.05**	**1.05**	**1.05**	**1.05**	**1.05**
ResNeXt-50	-	-	-	**1.13**	**1.13**	**1.13**
NasNet-A	-	-	-	> 34.08	> 12.67	**5.03**
BERT	-	-	-	**1.07**	1.09	1.09
SRU	-	**1.07**	**1.07**	**1.07**	**1.07**	**1.07**
NasRNN	-	-	-	**1.07**	1.38	1.70

From Table 3, we can see that ENU cannot finish within 1 hour for all models. Pruning and DPP have results for ResNet and SRU, because the search space of these two models are the smallest among all 7 models.

According to Table 3a, Sampling finds the optimal solution on ResNet and SRU. On other models, Sampling also always finds the best graph substitution sequence compared with MF and TS. Compared with MF, On NasNet-A, the optimized graph cost by Sampling is 1.15 times better, and on Inception-v3 and ResNeXt-50, Sampling can also achieve about 8% improvement. On NasRNN, the optimized cost by Sampling is 1% smaller than MF. Compared with TS, Sampling has the same optimization effect.

For search efficiency, according to Table 3b, DPP is about 3 times faster than Pruning on ResNet. This is because Pruning spends about 18 seconds on finding substitutions, while DPP only spends 1.69 seconds on it, due to the reusing technique. However, on SRU, DPP is slower than Pruning, because SRU is a rather small model and there are not many substitutions to reuse. In this case, DPP only saves 5 seconds on finding substitutions, but spends about 20 seconds on additional computation for the reusing technique. Sampling shows great advantage over other algorithms in terms of the search time. It can finish within seconds for most models. For the two most complex models, NasNet-A and NasRNN, the respective search time of it is about 233 and 20 seconds. No any other algorithm in Table 3b can finish in 1 hour for all models. Although on SRU, the search time of Sampling is longer than MF and TS, the absolute value of



(a) Optimized cost, ResNet blocks **(b)** Search time, ResNet blocks **(c)** Memory, ResNet blocks

(d) Optimized cost, InceptionE blocks **(e)** Search time, InceptionE blocks **(f)** Memory, InceptionE blocks

Figure 4: Results on ResNet blocks and InceptionE blocks, varying the block number.

the search time is less than 0.3 seconds, which is very small. On other models, Sampling is the fastest.

As for memory consumption, all algorithms on most models cost less than 2 GB. On ResNet and SRU, Pruning, DPP and Sampling perform similarly compared with MF and TS. On Inception-v3, Sampling costs 0.05 GB memory more than MF, but saves at least 12% of the memory TS costs. On BERT, Sampling performs the same as TS, and costs 0.02 GB more than MF. On NasRNN, Sampling respectively costs 0.63 GB and 0.32 GB more memory than MF and TS. The extreme case is on NasNet-A, where MF and TS consume tens of GBs memory, and Sampling saves 85% and 60% of the memory MF and TS cost respectively.

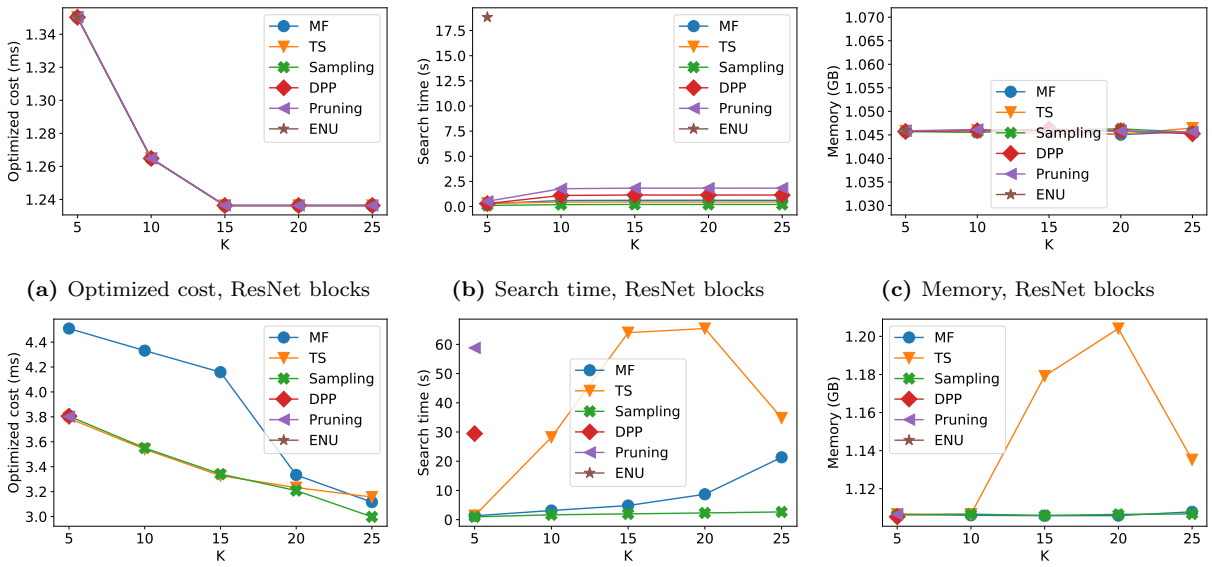
In a nutshell, we get the following observations: (1) Pruning and DPP are much more efficient than ENU in search time. DPP can save the time for finding substitutions compared with Pruning, but for small models where there are not many substitution reusing opportunities (like SRU), the addition computation may make DPP run slower. (2) MF uses graph split strategy to accelerate searching, but its effectiveness is harmed on some models (like Inception-v3). (3) MF and TS are able to find optimal graph substitution sequences efficiently on DNNs with relatively simpler architectures. Their performance suffer in face of larger or more complex computation graphs. (3) Sampling is effective and efficient to deal with different types of computation graphs.

Summary. From Table 3, we can find that the search time of algorithms in general is influenced by two factors: the input substitution rules and the DNN structures. For example, Sampling runs slowest on NasNet-A. This is because: (1) NasNet-A is a CNN and many input substitution rules deal with convolution operators; (2) NasNet-A has a complex architecture. Hence, many substitution sequences can be found for NasNet-A (the number of substitution sequences with only one substitution for NasNet-A is already 346), and there can be causality among substitutions in a substitution sequence as well, i.e., different substitutions can lead to different substitution sequences following them. As a result, the search space of all the possible substitution sequences is much larger than other DNNs. While ResNet and ResNeXt-50 are also CNNs, their structures are simpler than NasNet-A, and therefore Sampling runs faster on them. Among models other than CNNs, compared with BERT and SRU, NasRNN has more substitution possibilities due to the complex structure and the substitution rules as well (the number of substitution sequences of length 1 is already 371 for NasRNN) and Sampling reports longer search time on it. But the causality among substitutions does not affect the search space as significantly as for NasNat-A, so Sampling runs faster on NasRNN than on NasNet-A.

6.2.2 *Different DNN Sizes*

Since DNNs usually consist of repetitive blocks, in order to analyse the effect of DNN sizes, we fix the block structure and vary the number of blocks in the DNNs. We present the evaluation results on DNNs consisting of 2, 4, 6, 8 ResNet blocks (each consists of 5 nodes) and DNNs with 1, 2, 3, 4 InceptionE blocks [14] (each consists of 13 nodes). K is set to 10. Figure 4a-4c show the results on ResNet blocks, while Figure 4d- 4f provide the results on InceptionE blocks.

On ResNet blocks, by referring to the optimized cost of our two exact algorithms, we can see that Sampling finds the optimal substitution sequences for all the block numbers, while TS and MF report slightly higher optimized costs when the number of blocks is 2 and 4. The optimized graph by Sampling is 1.0-1.13 times faster than that by TS and MF. In terms of the search time, DPP and Pruning are much more efficient than ENU, they are about 1515 and 1681 times faster than ENU respectively on the model of 4 ResNet blocks. ENU cannot finish within 1 hour on models of 6, 8 ResNet blocks. DPP achieves superior performance than Pruning on large computation graphs thanks to the reusing substitution technique. The exceptions are the models with 2, 4 ResNet blocks, where DPP runs slower



(a) Optimized cost, ResNet blocks **(b)** Search time, ResNet blocks **(c)** Memory, ResNet blocks

(d) Optimized cost, InceptionE blocks **(e)** Search time, InceptionE blocks **(f)** Memory, InceptionE blocks

Figure 5: Results on ResNet blocks and InceptionE blocks, varying K.

than Pruning. This is because DPP needs additional computation cost for the reusing part, and this overhead becomes insignificant on larger models involving 6, 8 blocks where DPP runs 1.60-2.58 times faster than Pruning. The difference between DPP and TS, MF in search time becomes larger as the block number increases. On the model with 8 ResNet blocks, the search time ratio of DPP against TS and MF is 2.79 and 5.84 respectively. Sampling can be up to 30.56, 10.94, 5.24 times faster than DPP, TS and MF respectively. In terms of the memory consumption, there is not many differences among our algorithms and the baselines, ENU, TS and MF. The memory consumption of all the algorithms is quite stable as the block number varies.

On InceptionE blocks, we exclude the results of the three exact algorithms due to long search time (tens of hours). This is because InceptionE blocks are much more complex than ResNet blocks, resulting in a huge search space. For the remaining algorithms, in terms of the optimized cost, Sampling performs similar to TS, but is up to 1.23 times better than MF. As for the search time, Sampling runs up to 19.26 and 9.47 times faster than TS and MF, respectively. The memory consumption of Sampling, TS, and MF is similar, and the trends are stable as the number of blocks increases.

Summary. Figure 4 shows for the same block type, all the algorithms takes longer search time with the increasing block number in a model. This is reasonable because more blocks means more substitutions to be explored, resulting in a larger search space. Furthermore, there can be substitutions across blocks rather than within a single block, and hence the number of substitutions that can be performed on a computation graph does not simply increase linearly with the block number. It is worth mentioning that the search time of Sampling is consistently small over all the block numbers. We also notice that the memory consumption does not change a lot according to the experiments. This observation is consistent with the space complexity of our algorithms, and the fact that the absolute increase in substitutions with the block number on an original or intermediate graph is typically small and cannot affect the memory consumption significantly for ResNet and InceptionE blocks.

6.3 Evaluation on Length Constraint

To evaluate the effect of length constraint, we consider two DNNs: one consisting of 6 ResNet blocks and the other with 3 InceptionE blocks. K is chosen from the set {5, 10, 15, 20, 25}. Figure 5a- 5c and Figure 5d- 5f show the results on the ResNet and InceptionE models, respectively.

On the ResNet model, we only report the results of ENU when $K = 5$ because of the long search time. All the other algorithms report the optimal substitution sequence as K varies. The search time of our proposed algorithms do not increase too much when $K \geq 10$, because the optimal substitution sequence is of length 11. DPP runs 64.65 times faster than ENU (when $K = 5$), 1.60-1.78 times faster than Pruning, and only up to 2.56 and 1.87 times slower than TS and MF (but DPP has accuracy guarantee), respectively. Sampling runs up to 5.78, 3.22, 3.14 times faster than DPP, TS and MF, respectively. The memory consumption among algorithms are close and stable with different values of K.

On the InceptionE model, we only report the results of Pruning and DPP for $K = 5$ as they take tens of hours for larger K. We exclude the results of ENU also because of its long search time. When $K = 5$, DPP is 2 times faster than Pruning and 30.44 times slower than Sampling. For the remaining algorithms, in terms of the optimized cost, Sampling performs similar to TS, but is 1.05 times better than TS when $K = 25$. Compared with MF, Sampling always finds a much better substitution sequence. The optimized cost reported by Sampling is 1.04-1.24 times better than that by MF. For the search time, the growth rate of Sampling is significantly slower than that of TS and MF, and Sampling can be 32.52 and 8.09 times faster than TS and MF, respectively. Sampling also outperforms TS on the memory consumption, which saves upto 8% memory space.

Summary. As the length constraint increases, the search space involving all the possible substitution sequences becomes larger. For complex models and large length constraints, our exact algorithms are inefficient to produce results in reasonable time (they would last for days or longer). Thanks to the time and space complexity guarantee, the search time and memory consumption of Sampling increase sublinearly as the length constraint becomes larger.

7. RELATED WORK

Optimizing DNN efficiency has attracted great attention. Our study is closely related to two categories of research: graph-level and operator-level optimization for DNNs.

Graph-level optimization. This kind of optimization tries to optimize computation graphs by applying rule-based transformations. A number of works leverage manually designed rules. TVM [5] recognizes four categories of graph operators, and provides generic rules to fuse these operators so that the intermediate results do not need to be maintained in memory, thus reducing the execution time. TensorRT [3] uses predefined rules to combine layers (operators) and eliminate unnecessary operatiors, e.g., fusing convolution, bias and ReLU operations together. MetaFlow [10] considers rules with increasing cost in the intermediate computation graphs to seek more optimization opportunities. Different from the above works, TASO [9] introduces a method to automatically generate rules (not limited to performance-improving ones) for a given set of operators, and then verify their correctness. It also jointly optimizes data layout when applying rules to computation graphs based on MetaFlow. All the existing works focus on the design of substitution rules and adopt heuristic algorithms to search substitution sequences for optimizing computation graphs. However, none of them have paid attention to the theoretical analysis of the computation graph optimization problem. The effectiveness or efficiency of existing heuristics may suffer in face of complex models due to the large search space. In contrast, our work formally provides the hardness results of the computation graph optimization problem via graph substitution sequences, and introduces two exact algorithms and an efficient sampling-based approximation method with polynomial time complexity.

Operator-level optimization. This kind of optimization aims to make operators run faster. There are libraries for basic operations like GEMM (General matrix multiplication), including cuBLAS, Neon, and OpenAI [1, 2, 4]. Besides these libraries, TVM [5] optimizes low-level programs according to hardware characteristics via a learning-based cost modeling method. While Tensor Comprehensions [15] uses black-box auto-tuning and polyhedral optimizations. Astra [13] is a compilation and execution framework, and performs an amplified variant of multi-version compilation. TensorRT [3] runs operators on dummy data to select the fastest from its kernel catalog to make adaptive optimization for different environments. Our work focuses on accelerating the process of graph-level optimization and hence the above approaches are orthogonal to our problem.

8. CONCLUSIONS

In this paper, we formally define the OCGGS problem and prove it to be NP-hard and Poly-APX-complete. We introduce a partial order among graph substitutions to identify redundant substitution sequences. We develop the pruning-based search algorithm and the dynamic programming method that reuses the information of explored graph substitutions, with detailed analysis on the time and space complexity of the two exact algorithms. To deal with complex computation graphs with large search space, we propose an efficient sampling-based approximation algorithm with polynomial time and space complexity. Extensive experiments demonstrate that our Sampling algorithm is effective and efficient to find good substitution sequences, compared with existing techniques. Recently, Huawei has made its deep learning framework, Mindspore, open sourced[2]. Mindspore has achieved great performance on training DNNs. We would like to investigate applying our proposed methods on Mindspore IR to further reduce the training cost.

9. ACKNOWLEDGEMENTS

This work is partially supported by the Hong Kong RGC GRF Project 16202218, CRF Project C6030-18G, C10 31-18G, C5026-18G, AOE Project AoE/E-603/18, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001 and 2019A1515110473, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, Didi-HKUST joint research lab project, and Wechat and Webank Research Grants. Yanyan Shen is the corresponding author.

APPENDIX

A. PROOF OF THEOREM 3

PROOF. Before showing that the order defined in Definition 8 is a partial order, we introduce a concept called "depth" for graph substitutions. Specifically, the depth of the dummy substitution N is 0. For a graph substitution $\tau = (\phi, g : V_s \to V_m, h) \neq N$, its depth is the maximum depth of all the substitutions it depends on plus 1.

In order to prove the order in Definition 8 is a partial order, we need to prove three properties of it: reflexivity, antisymmetry and transitivity. By definition, for graph substitutions of depth 0, these three properties hold. Suppose the above three properties hold for graph substitutions of depth $\leq n$ ($n \geq 0$). Consider substitutions of depth $\leq n+1$. let τ_1, τ_2, τ_3 be any three graph substitutions of depth $\leq n+1$ in a graph substitution sequence ($\tau_1, \tau_2, \tau_3 \neq N$), and the respective biggest substitutions they depend on are $\tau_1^*, \tau_2^*, \tau_3^*$, whose depths are $\leq n$. $\tau_1^*, \tau_2^*, \tau_3^*$ must exist because the order over substitutions of depth $\leq n$ is a partial order.

• Reflexivity. Suppose τ_1 is of depth $n+1$. By definition, $\tau_1 \preceq \tau_1$.

• Antisymmetry. For any substitution τ, $(N \preceq \tau, \tau \preceq N) \Rightarrow N = \tau$. Further, if $\tau_1 \preceq \tau_2, \tau_2 \preceq \tau_1$, then $\tau_1^* = \tau_2^*$, and τ_1, τ_2 replace some common nodes. Therefore, $\tau_1 = \tau_2$.

• Transitivity. For any three substitutions either being N or in the same sequence, it is easy to see this property holds. Further, $(\tau_1 \preceq \tau_2, \tau_2 \preceq \tau_3) \Rightarrow (\tau_1^* \preceq \tau_2^*, \tau_2^* \preceq \tau_3^*) \Rightarrow \tau_1^* \preceq \tau_3^*$. If $\tau_1^* = \tau_3^*$, then $\tau_1^* = \tau_2^* = \tau_3^*$; otherwise, $\tau_1^* \neq \tau_3^*$. In both cases, by Definition 8, $\tau_1 \preceq \tau_3$.

By mathematical induction, we prove that the order in Definition 8 is a partial order. □

[2]https://github.com/mindspore-ai/

10. REFERENCES

[1] Cuda basic linear algebra subroutine library. `https://developer.nvidia.com/cuda-toolkit`.

[2] Neon. `https://github.com/NervanaSystems/neon`.

[3] Nvidia tensorrt: Programmable inference accelerator. `https://developer.nvidia.com/tensorrt`.

[4] Open single and half precision gemm implementations. `https://github.com/openai/openai-gemm`.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 578–594, 2018.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, 2019.

[7] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 1725–1731, 2017.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[10] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, 2019.

[11] T. Lei, Y. Zhang, and Y. Artzi. Training rnns as fast as cnns. *CoRR*, abs/1709.02755, 2017.

[12] U. Pferschy and J. Schauer. The maximum flow problem with disjunctive constraints. *Journal of Combinatorial Optimization*, 26(1):109–119, 2013.

[13] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[14] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[15] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[16] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 346–353, 2019.

[17] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[18] L. Zheng, V. Noroozi, and P. S. Yu. Joint deep modeling of users and items using reviews for recommendation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 425–434, 2017.

[19] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations*, 2017.

[20] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.