Redundancy-Free Computation for Graph Neural Networks

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Sina Lin
Microsoft
silin@microsoft.com

Rex Ying
Stanford University
rexying@stanford.edu

Jiaxuan You
Stanford University
jiaxuan@stanford.edu

Jure Leskovec
Stanford University
jure@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Graph Neural Networks (GNNs) are based on repeated aggregations of information from nodes' neighbors in a graph. However, because nodes share many neighbors, a naive implementation leads to repeated and inefficient aggregations and represents significant computational overhead. Here we propose *Hierarchically Aggregated computation Graphs* (HAGs), a new GNN representation technique that explicitly avoids redundancy by managing intermediate aggregation results hierarchically and eliminates repeated computations and unnecessary data transfers in GNN training and inference. HAGs perform the same computations and give the same models/accuracy as traditional GNNs, but in a much shorter time due to optimized computations. To identify redundant computations, we introduce an accurate cost function and use a novel search algorithm to find optimized HAGs. Experiments show that the HAG representation significantly outperforms the standard GNN by increasing the end-to-end training throughput by up to 2.8× and reducing the aggregations and data transfers in GNN training by up to 6.3× and 5.6×, with only 0.1% memory overhead. Overall, our results represent an important advancement in speeding-up and scaling-up GNNs without any loss in model predictive performance.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Graph neural network, redundancy-free computation

ACM Reference Format:
Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020. Redundancy-Free Computation for Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394486.3403142

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403142

1 INTRODUCTION

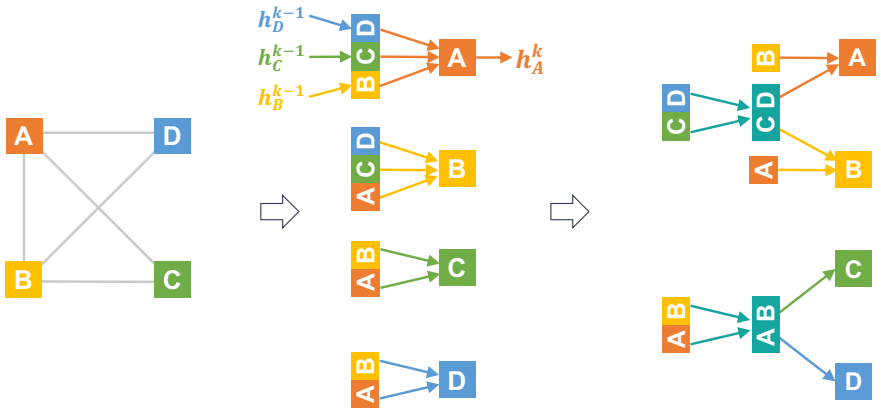
Graph Neural Network models (GNNs) generalize deep representation learning to graph data [3, 9, 23] and have achieved state-of-the-art performance across a number of graph-based tasks, such as node classification, link prediction, and graph classification and recommender systems [8, 14, 24, 27].

GNNs are based on a recursive neighborhood aggregation scheme, where within a single layer of a GNN each node aggregates its neighbors' activations and uses the aggregated value to update its own activation [23]. Such updated activations are then recursively propagated multiple times (multiple layers). In the end, every node in a GNN collects information from other nodes that are in its k-hop network neighborhood [8]. The activations of the final GNN layer are then used for downstream prediction tasks, such as node classification, graph classification, or link prediction.

Scaling GNNs to efficiently model very large graph datasets remains a challenge [6]. In particular, many implementations aim to work with the entire graph Laplacian matrix, which cannot fit into main memory even for medium-size graphs [14]. The framework GraphSAGE [8] improves over this approach by first performing sampling of small graph neighborhoods over individual nodes in each minibatch and then aggregating the neighborhood messages for these nodes. However, sampling individual neighborhoods is an expensive process and many hand-tuned heuristics are used to limit the sampling complexity and select the neighborhood graph [26]. Several works [4, 5, 12, 29] improve the efficiency of GNNs by focusing on the sampling step of GraphSAGE.

Present work. In this paper, we focus on speeding up the aggregation step of GNN models while preserving model performance. We propose a new GNN representation called *Hierarchically Aggregated computation Graphs* (HAGs), which allows us to make GNNs more efficient and more scalable.

We start with an observation that when there is significant overlap between the network neighborhoods of two nodes, the nodes' GNN graphs will have significant overlap and simply computing activation propagation on each one separately leads to redundant computations (Figure 1). In particular, we observe that existing GNN representations use a *computation graph* (referred to as a GNN-graph, Figure 1b) to define computation in a GNN layer. The GNN-graph includes a tree structure for each node u in the input graph describing how to compute u's activations by aggregating the previous-layer activations of u's neighbors. Figure 1b shows just a single layer of the GNN-graph of the input graph in Figure 1a;



(a) Input graph (b) GNN-graph (c) HAG

Figure 1: Comparison between a GNN-graph and an equivalent HAG. (a) Input graph; (b) 1-layer GNN computation graph (GNN-graph); (c) HAG that avoids redundant computation. The GNN-graph computes new activations $h_v^{(k)}$ by aggregating the previous-layer activations of v's neighbors. Because nodes in the input graph share common neighbors, the GNN-graph performs redundant computation (e.g., both $\{A, B\}$ and $\{C, D\}$ are aggregated twice). (c) By identifying common computational patterns, the HAG avoids repeated computation and leads to significant speed-ups. For example, rather than needing 10 aggregations (b), the HAG needs only 6 aggregations for the same calculation (c).

for example, for node A, its neighbors' activations $h_B^{(k-1)}$, $h_C^{(k-1)}$ and $h_D^{(k-1)}$ from the layer $k-1$ are aggregated to compute new activation $h_A^{(k)}$ of A at layer k. The new activations of the other nodes are computed similarly using the previous activations of their neighbors and the whole process repeats for multiple layers. Notice that this representation results in redundant computation and data transfers. In this small example, both $\{A, B\}$ and $\{C, D\}$ are aggregated twice. Furthermore, due to triadic closure and clustering of real-world graphs, we expect such overlaps and redundancies to be rather significant [21]. Furthermore, as GNNs grow wider and multi-layer, the redundancies in GNN-graphs account for a significant fraction of all computation. For example, our experiments show that in modern GNNs up to 84% of the aggregations are redundant and can be avoided to speed-up training as well as inference of GNNs.

Contributions. To avoid redundant computations we propose a new GNN representation called *Hierarchically Aggregated computation Graphs* (HAGs). Figure 1c shows the HAG for the input graph in Figure 1a, which eliminates redundant computation and unnecessary data transfers in GNN computation graphs by hierarchically managing and reusing intermediate aggregation results.

HAGs are functionally equivalent to standard GNN-graphs in the sense that they produce the same models and same predictions as the original GNN, but HAGs represent common neighbors across different nodes using aggregation hierarchies, which eliminates redundant computation and unnecessary data transfers in both GNN training as well as inference. In addition, a HAG is agnostic to any particular GNN model, and can be used to eliminate redundancy for arbitrary GNNs. HAG applies to both order invariant aggregation functions (GCN [14], GraphSAGE [8], PinSage [26], GIN [24], P-GNN [28]), as well as to GNN architectures where ordering of nodes in the aggregation matters (GraphSAGE-LSTM [8], Tree-LSTM [18], Ego-CNN [19]).

To identify redundancies in GNNs and compose HAGs, we introduce an accurate cost function to estimate the cost of a given HAG. Then we develop a linear-time HAG search algorithm to automatically find a HAG with small computational cost. We prove that the search algorithm finds HAGs with strong performance guarantees: (1) for GNN models whose neighborhood aggregations require a specific ordering on a node's neighbors, the algorithm finds a *globally optimal* HAG under the cost function; and (2) for other GNN models, the algorithm finds HAGs whose runtime performance is at least a $(1 - 1/e)$ approximation ($\approx 63\%$) of globally optimal HAGs using the submodularity property [17]. Most importantly, our HAG abstraction maintains the predictive performance of GNNs but leads to much faster training and inference.

We evaluate the runtime performance of HAGs on three node classification datasets and two graph classification datasets using three different GNN architectures (GCN, GraphSAGE, SimpleGC). Our evaluation focuses on three dimensions: (a) end-to-end training and inference runtime; (b) number of aggregations; and (c) size of data transfers. The experiments show that HAGs increase the end-to-end training and inference runtime performance of GNNs on node as well as graph classification by up to 2.8× and 2.9×, respectively. In addition, compared to standard GNN-graphs, HAGs reduce the number of aggregations and the size of data transfers by up to 6.3× and 5.6×, respectively, across the three predictive tasks, five datasets, and three GNN architectures.

2 RELATED WORK

We briefly survey related work in graph neural networks.

Graph neural networks have been used to solve various real-world tasks with relational structures: GCN [14], GraphSAGE [8],

Algorithm 1 An abstraction for GNNs. $\mathcal{V}$ is the set of nodes in an input graph, and $\mathcal{N}(v)$ denotes the set of neighbors for node v.

1: $h_v^{(0)} = x_v, \forall v \in \mathcal{V}$
2: **for** $k = 1$ to K **do**
3: **for** $v \in \mathcal{V}$ **do**
4: $a_v^{(k)} \leftarrow \textsc{Aggregate}(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\})$
5: $h_v^{(k)} \leftarrow \textsc{Update}(a_v^{(k)}, h_v^{(k-1)})$
6: **Goal:** minimize $\mathcal{L}(\{h_v^{(K)} | v \in \mathcal{V}\})$

DiffPool [27], P-GNN [28] and GIN [24]. To improve training efficiency on large graphs, FastGCN [4] and SGC [22] accelerate GNN training using importance sampling and removing nonlinearilities, and ClusterGCN[6] approximates the graph by focusing on intra-cluster edges. Our paper solves the orthogonal problem of optimizing GNN efficiency while maintaining network accuracy by removing redundant aggregations. Furthermore, our method can be easily applied to the models above to speed them up without loss of predictive accuracy, model training time or generalization ability. Although our model does not apply to GAT[20], GAT cannot scale well on large datasets such as REDDIT due to memory constraints, and is not considered in existing scalable GNNs[4, 5, 29].

Join-trees are a tree decomposition technique that maps a graph into a corresponding tree structure to solve optimization problems on the graph, such as query optimization [7]. Although a join-tree provides a possible way to find optimal HAGs for a GNN-graph, its time complexity is exponential in the *treewidth* of a graph [2], and real graphs tend to have very large treewidths. For example, Adcock et al. [1] shows that the treewidth of real-world social networks grows linearly with the network size, making it infeasible to use join-trees to find optimal HAGs.

Computation reduction in DNNs. Recent work has proposed several techniques to reduce computation in DNNs, including pruning weights [11] and quantization [10]. For example, Han et al. [11] presents a weight pruning algorithm to iteratively remove weak connections in a network. As another example, Han et al. [10] proposes a deep compression technique to reduce network computation by training on low precision weights. These techniques reduce computation at the cost of modifying networks, resulting in decreased accuracy (as reported in these papers). In contrast, we propose a new GNN representation that accelerates GNN training by eliminating redundancy in GNN-graphs while maintaining the original network accuracy.

3 HIERARCHICALLY AGGREGATED COMPUTATION GRAPHS (HAGS)

Existing GNN-graph representation. An input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has nodes $\mathcal{V}$ and edges $\mathcal{E}$. For each node $v \in \mathcal{V}$, $\mathcal{N}(v)$ denotes the set of neighbors of v, and x_v denotes the input node features. A GNN iteratively learns representations for individual nodes over the entire graph through a number of GNN layers, as shown in Algorithm 1. The learned activations of node v at layer k is $h_v^{(k)}$, and we initialize $h_v^{(0)}$ with x_v. At the k-th layer, $a_v^{(k)}$ denotes the aggregated activations of v's neighbors, which is combined with $h_v^{(k-1)}$ to compute an updated activation $h_v^{(k)}$. The learned node activations of the final layer (i.e., $h_v^{(K)}$) are used for downstream learning tasks, and a GNN model generally minimizes a loss function $\mathcal{L}$ that takes the final node activations as inputs (line 6).

Existing GNN models use a GNN *computation graph* (GNN-graph) to describe the computation in each GNN layer, as shown in Figure 1b. For each node v in the input graph, the GNN-graph includes an individual tree structure to define how to compute the activations $h_v^{(k)}$ of node v by aggregating the previous-layer activations of v's neighbors (i.e., $\{h_u^{(k-1)}, u \in \mathcal{N}(v)\}$). GNN-graphs are efficient at expressing direct neighborhood relations between nodes, but are not capable of capturing common neighbors across multiple nodes, leading to redundant computation in GNN training and inference.

3.1 HAG Representation

We propose a new graph representation called *Hierarchically Aggregated computation Graphs* (HAGs) for GNNs. HAGs eliminate redundancy in the GNN-graph representation by hierarchically managing and reusing intermediate aggregation results. A HAG $\widehat{\mathcal{G}} = (\widehat{\mathcal{V}}, \widehat{\mathcal{E}})$ has nodes $\widehat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}_A$ and edges $\widehat{\mathcal{E}}$, where $\mathcal{V}$ is the set of nodes in the original graph, and $\mathcal{V}_A$ is a new set of *aggregation* nodes. Each aggregation node in $\mathcal{V}_A$ represents the intermediate aggregations result for a subset of nodes (i.e., aggregation on a subset of $h_v^{(k-1)}$) . For the HAG example in Figure 1c, the new nodes *AB* and *CD* denote the aggregation results of $\{A, B\}$ and $\{C, D\}$, respectively. A HAG can contain a multi-level aggregation hierarchy. For example, Figure 1c can also have a third aggregation node *ABCD* that depends on *AB* and *CD*. Similar to edges in GNN-graphs, an edge (u, v) in a HAG denotes an aggregation relation — computing v's activations requires aggregating u's activations.

The standard GNN-graph representation can be considered a special case in the HAG representation with no intermediate aggregation nodes (*i.e.*, $\mathcal{V}_A = \emptyset$). Our HAG abstraction is general and applicable to many existing GNN models. Table 1 shows how to use our abstraction to define existing GNNs, which can be further divided into two categories:

- **Set Aggregate**. Most GNNs assume the neighbors of a node have *no ordering*, and the aggregations are *associative* and *commutative* operations that are invariant to the order in which the aggregations are performed. Examples include GCN with summation aggregations and GraphSAGE-P with element-wise pooling aggregations (Table 1). Note that set aggregations in GNNs are designed to be order invariant and thus can be performed in a hierarchical fashion as we do in HAGs.
- **Sequential Aggregate**. Another class of GNNs require a specific ordering of a node's neighbors and the aggregations are not commutative. Examples include N-ary Tree-LSTM [18] and the LSTM variant of GraphSAGE [8]. HAGs can be applied in the case of sequential aggregations as well. Rather than identifying common subsets of neighbors, we identify the common prefixes of the sequence of aggregated nodes, which can then be reused among nodes.

We further define two properties for the aggregation nodes $\mathcal{V}_A$. First, for each $v \in \mathcal{V}_A$, $\widehat{a}_v$ denotes its intermediate aggregation result, and $\widehat{\mathcal{N}}(v)$ denotes the in-neighbors of node v. To capture the aggregation hierarchy in a HAG, we use a recursive function to

GNN	AGGREGATE($\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}$)	UPDATE($a_v^{(k)}, h_v^{(k-1)}$)
Set AGGREGATE		
GCN [14]	$a_v^{(k)} = \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$	$h_v^{(k)} = \sigma(W^{(k)} \cdot \frac{a_v^{(k)} + h_v^{(k-1)}}{\lvert\mathcal{N}(v)\rvert+1})$
GIN [24]	$a_v^{(k)} = \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$	$h_v^{(k)} = \sigma\big(W \cdot \big((1+\epsilon^{(k)})h_v^{(k-1)} + a_v^{(k)}\big)\big)$
GraphSAGE-P [8]	$a_v^{(k)} = max_{u \in \mathcal{N}(v)}\{\sigma(W_1^{(k)} \cdot h_u^{(k-1)})\}$	$h_v^{(k)} = \sigma\big(W_2^{(k)} \cdot (a^{(k)}, h_v^{(k-1)})\big)$
SGC [22]	$a_v^{(k)} = \sum_{(u,v) \in S^k} h_u^{(0)}$	$h_v^{(k)} = \sigma\big(W \cdot a_v^{(k)}\big)$
Sequential AGGREGATE		
GraphSAGE-LSTM [8]	$a_v^{(k)} = LSTM(h_{v_1}^{(k-1)}, ..., h_{v_N}^{(k-1)})$	$h_v^{(k)} = \sigma\big(W^{(k)} \cdot (a_v^{(k)}, h_v^{(k-1)})\big)$
N-ary Tree-LSTM [18]	$a_v^{(k)} = Tree\text{-}LSTM\text{-}Agg(h_{v_1}^{(k-1)}, ..., h_{v_N}^{(k-1)})$	$h_v^{(k)} = Tree\text{-}LSTM\text{-}Update(a_v^{(k)}, h_v^{(k-1)})$

Table 1: Existing GNNs described in our abstraction. GraphSAGE-P and GraphSAGE-LSTM are the pooling and LSTM variants of GraphSAGE, respectively. σ and max indicate element-wise non-linear activation and max functions. For sequential AGGREGATE, v_i denotes the i-th in-neighbor of node v.

Algorithm 2 A GNN abstraction with HAGs. We exclude layer index superscripts in $\widehat{a}_v$ to denote that $\widehat{a}_v$ does not need to be memoized for back propagation, and its memory can be reused across all layers.

1: $h_v^{(0)} = x_v, \forall v \in \mathcal{V}$
2: **for** $k = 1$ to K **do**
3: **for** $v \in \mathcal{V}_A$ **do**
4: compute $\widehat{a}_v$ using Equation (1)
5: **for** $v \in \mathcal{V}$ **do**
6: $a_v^{(k)} \leftarrow$ AGGREGATE($\{\widehat{a}_u \mid u \in \widehat{\mathcal{N}}_v\}$)
7: $h_v^{(k)} \leftarrow$ UPDATE($a_v^{(k)}, h_v^{(k-1)}$)

define $\widehat{a}_v$.

$$\widehat{a}_v = \text{AGGREGATE}\left(\left\{\begin{matrix} h_u^{(k-1)} & u \in \mathcal{V} \\ \widehat{a}_u & u \in \mathcal{V}_A \end{matrix}\right. \middle| \, u \in \widehat{\mathcal{N}}(v)\right) \tag{1}$$

Second, $\widehat{\mathcal{C}}(v)$ denotes the set of input activations $h_u^{(k-1)}$ used to compute $\widehat{a}_v$ in the recursive procedure.

$$\widehat{a}_v = \text{AGGREGATE}(\{h_u^{(k-1)} \mid u \in \widehat{\mathcal{C}}(v)\}) \tag{2}$$

Intuitively, $\widehat{\mathcal{C}}(v)$ defines the coverage of node v in a HAG. For the HAG example in Figure 1c, $\widehat{\mathcal{C}}(A) = \{B, C, D\}$ because $h_B^{(k-1)}$, $h_C^{(k-1)}$, and $h_D^{(k-1)}$ are used as inputs to compute $h_A^{(k)}$.

For a set AGGREGATE, $\widehat{\mathcal{C}}(\cdot)$ is an unordered set:

$$\widehat{\mathcal{C}}(v) = \bigcup_{u \in \widehat{\mathcal{N}}(v)} \begin{cases} u & u \in \mathcal{V} \\ \widehat{\mathcal{C}}(u) & u \in \mathcal{V}_A \end{cases} \tag{3}$$

And for a sequential AGGREGATE, $\widehat{\mathcal{C}}(\cdot)$ is an ordered list:

$$\widehat{\mathcal{C}}(v) = \big(\widehat{\mathcal{C}}(u_1), ..., \widehat{\mathcal{C}}(u_m)\big) \tag{4}$$

where $u_1, ..., u_m$ are the ordered in-neighbors of v.

3.2 GNNs with HAGs

Next we show the equivalence between the GNN-graph abstraction and HAGs, as shown in Algorithm 2. As GNN-graph can be abstracted by a HAG, HAG can also be represented by a GNN. The only difference is how to compute neighborhood aggregations (i.e., $a_v^{(k)}$) in each GNN layer. Before the original aggregations are processed, we add one step that precomputes commonly used intermediate aggregation results to reduce redundant computation. In Algorithm 2, we first compute $\widehat{a}_v$ for all aggregation nodes $\mathcal{V}_A$ (line 3-4). This is performed recursively following the aggregation hierarchy of the HAG. We then compute the neighborhood aggregations $a_v^{(k)}$ (line 5-6) using the precomputed intermediate aggregations $\widehat{a}_v$.

Equivalence between GNN-graphs and HAGs. We then define a GNN-graph and a HAG to be *equivalent* if they produce the exact same outputs and gradients for a GNN model. Formally, a GNN-graph $\mathcal{G}$ and a HAG $\widehat{\mathcal{G}}$ are equivalent for a GNN model if (1) the GNN model outputs the same activations (i.e., $h_v^{(k)}$) at each GNN layer, and (2) the GNN model computes the same gradients for all trainable parameters in back propagation. Equivalent graphs guarantee the same predictive performance, and therefore can be used interchangeably for both GNN training and inference. Theorem 1 provides the equivalence between the two representations. We prove the theorem in the Appendix.

THEOREM 1. *A GNN-graph with nodes $\mathcal{V}$ and a HAG with nodes $(\mathcal{V}, \mathcal{V}_A)$ are equivalent if and only if $\mathcal{N}(v) = \widehat{\mathcal{C}}(v)$ for all $v \in \mathcal{V}$, where $\mathcal{N}(v)$ is v's neighbors in the input graph and $\widehat{\mathcal{C}}(\cdot)$ is defined in Equation 3 and 4.*

Memory overhead. Although Algorithm 2 includes new intermediate variables $\widehat{a}_v$, the memory overhead for storing $\widehat{a}_v$ is negligible since $\widehat{a}_v$ is not used for back propagation and can be saved in a constant memory across all GNN layers. In the experiments, we show that HAGs can increase the training throughput by 2.8×, while maintaining the original model accuracy at the cost of 0.1% memory overhead to store $\widehat{a}_v$. Meanwhile, storing a HAG representation requires less memory than the original GNN-graph, as HAGs generally contain much fewer edges.

4 HAG SEARCH ALGORITHM

For a GNN model and an input GNN-graph, there exists a large space of equivalent HAGs with the same model accuracy but various runtime performance. Our goal is to explore the search space to

discover a HAG with optimized runtime performance. First, we define a realistic cost function to quantitatively evaluate the runtime performance of different HAGs (Section 4.1). Second, we introduce an efficient search algorithm (Section 4.2) that finds an optimized HAG with the following guarantees :

- For GNNs with sequential AGGREGATE function, the HAG search algorithm can find *globally optimal* HAGs under this cost function.
- For GNNs with set AGGREGATE, finding an optimal HAG is NP-hard by a reduction from the NP-hard *maximum coverage problem* (see Appendix for the proof). The search algorithm finds at least a $(1-1/e)$-*approximation* of globally optimal HAGs based on the submodularity property [17].

4.1 Cost Function

We introduce a cost function that allows us to quickly estimate the runtime of a HAG by measuring the computation cost to perform one epoch of GNN propagation on the HAG.

The computation cost of a GNN model includes aggregating the neighbors of each node by calling AGGREGATE and updating the activations of each node via UPDATE, as shown in Algorithm 2. For a GNN model $\mathcal{M}$, we need to define two constants: the cost of performing AGGREGATE on two elements $\alpha_{\mathcal{M}}$, and the cost of computing an UPDATE $\beta_{\mathcal{M}}$. In Algorithm 2, computing $\widehat{a}_v$ with $|\widehat{\mathcal{N}}_v|$ neighbors requires performing $(|\widehat{\mathcal{N}}_v|-1)$ binary aggregations, whose cost is $\alpha_{\mathcal{M}} \times (|\widehat{\mathcal{N}}_v|-1)$. Therefore, the total computation cost of training a GNN model $\mathcal{M}$ on a HAG $\widehat{\mathcal{G}}$ is

$$\begin{aligned} cost(\mathcal{M}, \widehat{\mathcal{G}}) &= \sum_{v \in \mathcal{V} \cup \mathcal{V}_A} \alpha_{\mathcal{M}}(|\widehat{\mathcal{N}}_v| - 1) + \sum_{v \in \mathcal{V}} \beta_{\mathcal{M}} \\ &= \alpha_{\mathcal{M}}\big(|\widehat{\mathcal{E}}| - |\mathcal{V}_A|\big) + (\beta_{\mathcal{M}} - \alpha_{\mathcal{M}})|\mathcal{V}| \end{aligned}$$

$|\mathcal{V}|$ is determined by the input graph. $\alpha_{\mathcal{M}}$ and $\beta_{\mathcal{M}}$ only depend on the GNN model $\mathcal{M}$, and is independent to the HAG used for training. Therefore, our goal is to minimize $\big(|\widehat{\mathcal{E}}| - |\widehat{\mathcal{V}}_A|\big)$, where $|\widehat{\mathcal{E}}|$ and $|\widehat{\mathcal{V}}_A|$ are the number of edges and aggregation nodes in a HAG, respectively.

4.2 Search Algorithm

Given the cost function $cost(\mathcal{M}, \widehat{\mathcal{G}})$ our next goal is to find a HAG that minimizes the cost. Here we present a HAG search algorithm that finds a globally optimal HAG for GNNs with sequential AGGREGATE and a $(1-1/e)$-approximation of globally optimal HAGs for GNNs with set AGGREGATE. In addition to an input GNN-graph and a GNN model, the algorithm also takes a hyper-parameter *capacity*, defining an upper limit on the number of intermediate aggregation nodes (i.e., $|\mathcal{V}_A|$).

Algorithm 3 shows the pseudocode of the HAG search algorithm. We start with an input GNN-graph, and iteratively insert aggregation nodes into the current HAG to merge highly redundant aggregations and remove unnecessary computation and data transfers.

The REDUNDANCY function (line 3-8) evaluates the degree of redundancy for aggregating each node pair. By iteratively eliminating aggregations with the highest redundancy we lower the cost of the HAG, as defined in Section 4.1. More specifically, in each iteration

Algorithm 3 A HAG search algorithm to automatically find an equivalent HAG for a GNN-graph with optimized runtime performance. $\widehat{\mathcal{E}}$ and $\mathcal{V}_A$ are the set of edges and aggregation nodes in the HAG. REDUNDANCY$(v_1, v_2, \widehat{\mathcal{E}})$ calculates the number of nodes aggregating both v_1 and v_2. Recall that $\widehat{C}(u)$ is an ordered list for sequential AGGREGATE (see Equation 4).

```
1:  Input: A GNN-graph G and a GNN model M.
2:  Output: An equivalent HAG with optimized performance
3:  function REDUNDANCY(v1, v2, Ê)
4:      if M has a set AGGREGATE then
5:          R = {u | (v1, u) ∈ Ê ∧ (v2, u) ∈ Ê}
6:      else
7:          R = {u | v1 = Ĉ(u)[1] ∧ v2 = Ĉ(u)[2]}
8:      return |R|
9:
10: V_A ← ∅, Ê ← E
11: while |V_A| < capacity do
12:     (v1, v2) = arg max_{v1,v2} REDUNDANCY(v1, v2, Ê)
13:     if REDUNDANCY(v1, v2, Ê) > 1 then
14:         V_A ← V_A + {w}                ▷ where w is a new node
15:         Ê ← Ê + (v1, w) + (v2, w)
16:         for u ∈ V do
17:             if (v1, u) ∈ Ê ∧ (v2, u) ∈ Ê then
18:                 Ê ← Ê − (v1, u) − (v2, u) + (w, u)
19: return (V_A ∪ V, Ê)
```

we identify a binary aggregation with the highest *redundancy* and insert a new aggregation node w in $\mathcal{V}_A$ to represent the binary aggregation results (line 12-15). All nodes containing this binary aggregation can directly use the output of w without recomputing the aggregation (line 16-18). The search algorithm iteratively reduces the computation cost of the HAG by eliminating the most redundant aggregation in each iteration. The redundancy scores are maintained in a *heap* structure.

For a GNN model with a sequential AGGREGATE, Theorem 2 shows that our search algorithm finds an equivalent HAG with globally optimal computation cost. We prove the theorem in the appendix.

THEOREM 2. *For any GNN-graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and any GNN model $\mathcal{M}$ with a sequential AGGREGATE, Algorithm 3 returns an equivalent HAG with globally minimum cost as long as capacity $\geq |\mathcal{E}|$.*

For a GNN model with a set AGGREGATE, Theorem 3 shows that our search algorithm finds a HAG that is at least a $(1-1/e)$-approximation of the globally optimal HAGs (see the proof in the appendix).

THEOREM 3. *For any GNN-graph $\mathcal{G}$ and GNN model $\mathcal{M}$ with a set AGGREGATE, Algorithm 3 gives a $(1-1/e)$-approximation of globally optimal HAGs under the cost function. Formally, let $\widehat{\mathcal{G}}$ be the HAG returned by Algorithm 3, and $\widehat{\mathcal{G}}_o$ is a globally optimal HAG under the capacity constraint,*

$$cost(\mathcal{M}, \widehat{\mathcal{G}}) \leq \frac{1}{e} cost(\mathcal{M}, \mathcal{G}) + \frac{e-1}{e} cost(\mathcal{M}, \widehat{\mathcal{G}}_o)$$

HAGs produced by Algorithm 3 have the following three major advantages.

Time and space complexity. Our HAG algorithm achieves low theoretical complexity and has negligible runtime overhead. In particular, the overall time complexity of Algorithm 3 is $O(capacity \times |\mathcal{V}| + |\mathcal{E}| \times \log |\mathcal{V}|)$, and the space complexity is $O(capacity \times |\mathcal{V}| + |\mathcal{E}|)$ (see the appendix for the proof). One key optimization is a heap data structure for maintaining the redundancy scores of the highest $O(|V|)$ node pairs. Finding the most redundant node pair over the entire graph thus only takes $O(1)$ time, and updating the redundancy scores (i.e., line 15 and 18) each takes $O(\log |V|)$ time.

Fast GPU implementation. Real-world graphs have non-uniform edge distributions, leading to unbalanced workload among different nodes. Previous work [13, 16] has proposed different strategies to explicitly balance workload distributions among nodes at the cost of synchronization overhead among GPU threads. In contrast, Algorithm 3 produces HAGs whose aggregation nodes (i.e., $\mathcal{V}_A$) have uniform edge distributions (each has exactly two in-edges). This eliminates any synchronization overheads to balance workload among aggregation nodes and results in faster GPU implementations.

High reusability. For a given GNN-graph, the HAG produced by Algorithm 3 only depends on the capacity and aggregation type (set or sequential AGGREGATE) and is agnostic to any particular GNN models. This allows us to only run the search algorithm once for each aggregation type, and any GNN models can directly reuse the generated HAGs without any additional analysis of the graph.

5 EXPERIMENTS

The HAG representation maintains the predictive performance of GNNs but has much better runtime performance. This section evaluates the runtime performance of HAGs on five real-world graph datasets. We evaluate HAGs along three dimensions: (a) end-to-end training and inference performance; (b) number of aggregations; and (c) size of data transfers.

5.1 Implementation

Existing deep learning frameworks such as TensorFlow and PyTorch are designed for spatial data structures (e.g., images and text), and have limited support for irregular data structures such as graphs. As a result, GNN models in existing frameworks translate graph structures to sparse adjacent matrices and use matrix operations to perform GNN training.

We implemented the following operations in TensorFlow r1.14 to support GNN training with HAGs.

- First, `graph_to_hag` automatically transforms an input GNN-graph to an equivalent HAG with optimized performance.
- Second, `hag_aggregate` takes a HAG and nodes' activations as inputs, and computes the aggregated activations of all nodes.
- Finally, `hag_aggregate_grad` computes the gradients of `hag_aggregate` for back propagation.

Our implementation minimizes changes to existing GNN programs: a GNN application can directly use all HAG optimizations by only modifying a few lines of code.

Name	# Nodes	# Edges
Node Classification		
BZR	6,519	137,734
PPI	56,944	1,612,348
REDDIT	232,965	114,615,892
Graph Classification		
IMDB	19,502	197,806
COLLAB	372,474	12,288,900

Table 2: Datasets used in the experiments.

5.2 Experimental Setup

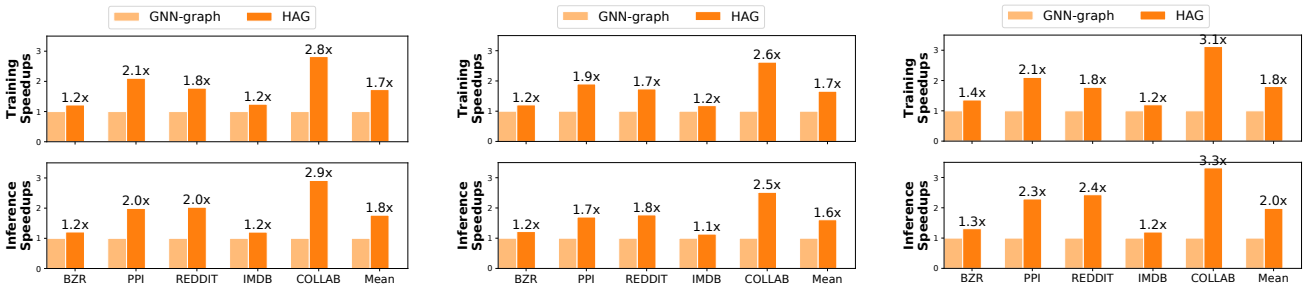
Datasets. Table 2 summarizes the public datasets used in our experiments. BZR is a chemical compound dataset, where each node is an atom and an edge is a chemical bond between two atoms [15]. PPI contains a number of protein-protein interaction graphs, each of which corresponds to a different human tissue [30]. REDDIT is an online discussion forum dataset, with each node being a Reddit post and each edge being commenting relations. For both PPI and REDDIT, we directly use preprocessed data from Hamilton et al. [8]. IMDB and COLLAB are two collaboration datasets for graph classification [25]. IMDB is a movie collaboration dataset, with each node representing an actor/actress, while COLLAB is a scientific collaboration dataset, with each node representing a researcher.

All experiments were performed running TensorFlow r1.14 on NVIDIA Tesla V100 GPUs. Following previous work [8, 14], each GNN model has two GNN layers and one SoftMax layer. For graph classification datasets, each GNN model also includes a mean-pooling layer to gather graph-level activations. For all experiments, we set the maximum *capacity* of $|\mathcal{V}_A|$ in a HAG to be $|\mathcal{V}|/4$, which achieves high performance on real-world graphs. In all experiments, the memory overhead to save intermediate aggregation results is negligible: intermediate nodes consume 6MB of memory in the worst case while GNN training requires more than 7GB of memory (~0.1% memory overhead).

5.3 End-to-End Performance

Per-epoch performance. We first measure the per-epoch training time and inference latency of GCN [14], GIN [24], and SGC [22] on different graph datasets. We follow previous work [8, 15, 25] to split the datasets into training, validation, and test sets, and use the testing sets to measure the average inference latency.

We perform our experiments on five different datasets, two different tasks (node classification, graph classification), and three different GNN architectures (GCN, GIN, SGC). Figure 2 compares the per-epoch training time and inference latency between GNN-graphs and HAGs across all these experimental configurations. Compared to GNN-graphs, HAGs can improve the training and inference performance by up to 3.1× and 3.3×, respectively, while maintaining the same model accuracy. We note this improvement is achieved completely automatically, and computing a HAG is inexpensive. Thus, because the improvement provided by HAGs maintains the original model accuracy and is essentially for free, we believe there is no reason not to use HAGs in preference to GNN-graphs.



(a) GCN (b) GIN (c) SGC

Figure 2: End-to-end runtime performance comparison between GNN-graphs and HAGs on two prediction tasks, five datasets, and three different GNN architectures. We measure the per-epoch training time and inference latency on GCN [14], GIN [24], and SGC [22]. The performance numbers are normalized by the GNN-graph numbers (higher is better). Note that across all experimental configurations HAGs consistently provide significant speed-ups.

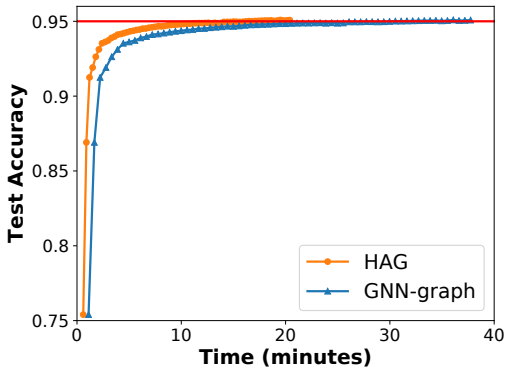


Figure 3: Time-to-accuracy comparison between HAG and GNN-graph for training a 2-layer GCN model on the Reddit dataset.

Time-to-accuracy performance. We compare the time-to-accuracy performance between HAG and GNN-graph. We train a 2-layer GCN model (with 64 hidden dimensions in each layer) on the Reddit dataset until the test accuracy exceeds 95%. We follow previous work [14] in setting all hyper-parameters and the split of the dataset.

Figure 3 shows the results. Each dot indicates the training time and test accuracy of each epoch. Ax expected, the GCN model using the HAG representation achieves exactly the same training and test accuracy at the end of each epoch. It takes 55 training epochs to achieve a test accuracy of 95% for both HAG and GNN-graph, and HAG improves the end-to-end training time by 1.8×.

5.4 Aggregation Performance

We further compare the aggregation performance of GNN-graphs and HAGs on the following two metrics: (1) the number of binary aggregations performed in each GNN layer; and (2) the size of data transfers between GPU threads to perform the aggregations. Note that aggregating a neighbor's activations requires transferring the activations from GPU global memory to a thread's local memory.

Figure 4 shows the comparison results. For GNNs with set aggregations, HAGs reduce the number of aggregations by 1.5-6.3× and the size of data transfers by 1.3-5.6×. For GNNs with sequential aggregations, HAGs reduce aggregations and data transfers by up to 1.8× and 1.9×, respectively.

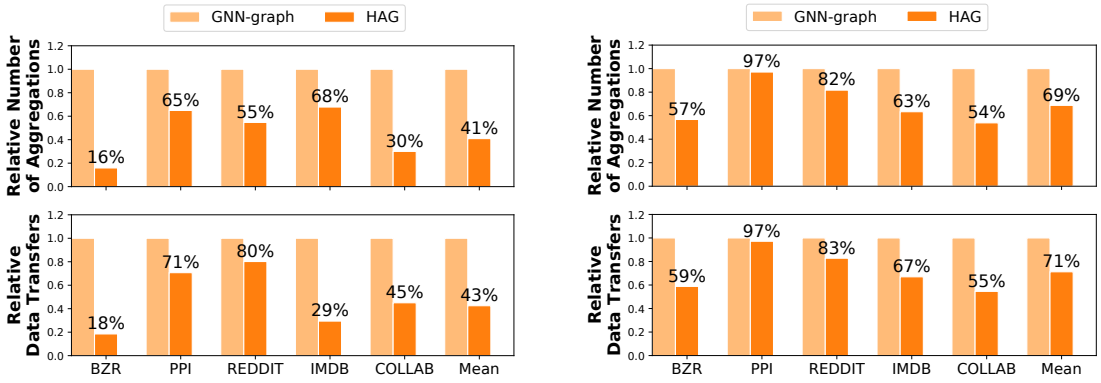
Although the search algorithm finds a globally optimal HAG for sequential aggregations (Theorem 2) and a $(1-1/e)$-approximation of globally optimal HAGs for set aggregations (Theorem 3), we observe the performance improvement is more significant for set aggregations. Optimality for HAGs with set aggregation involves more potential redundancy compared to sequential aggregations, due to permutation invariance of set aggregation. Thus higher performance can be achieved with HAGs for set aggregations, though optimal solutions are more difficult to compute.

It is also worth noting that the HAG search algorithm can find highly optimized HAGs even on very sparse graphs. For example, on the COLLAB dataset with a graph density of 0.01%, our algorithm reduces the number of aggregations and data transfers by 3.3× and 2.2×, respectively.

5.5 HAG Search Algorithm

We evaluate the performance of the HAG search algorithm. Recall that the search algorithm uses a hyper-parameter *capacity* to control the number of aggregation nodes in a HAG. A larger *capacity* allows the algorithm to eliminate more redundant aggregations and achieves lower cost.

Figure 5 shows the end-to-end GCN training time on the COLLAB dataset using HAGs with different capacities. A larger value of capacity can consistently improve the training performance, which indicates that the cost function is an appropriate metric to evaluate and compare the performance of different HAGs. By gradually increasing the capacity, the search algorithm eventually finds a HAG with ∼100K aggregation nodes, which consume 6MB of memory (0.1% memory overhead) while improving the training performance by 2.8×. In addition, the HAG search time is negligible compared to the end-to-end training time.



(a) Set Aggregations

(b) Sequential Aggregations

Figure 4: Comparing the number of aggregations and the total data transferred between GPU threads to perform aggregations (lower is better). The y-axes are normalized by GNN-graphs, and the last column in each figure is the geometric mean over all datasets. Notice that HAG reduces the number of required aggregation operations by up to 84%.

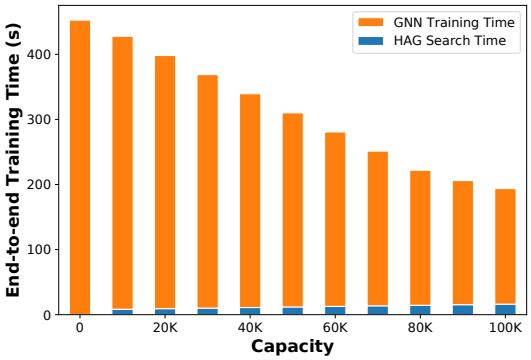


Figure 5: End-to-end GCN training time on the COLLAB dataset using HAGs with different capacities. We train GCN for a maximum of 350 epochs by following prior work [14].

6 CONCLUSION

We introduce HAG, a new graph representation to eliminate redundancy in many GNNs. We propose a cost function to estimate the performance of different HAGs and use a search algorithm to find optimized HAGs. We show that HAGs outperform existing GNN-graphs by improving the end-to-end training performance and reducing the aggregations and data transfers in GNN training.

ACKNOWLEDGMENTS

We thank Alexandra Porter, Sen Wu, and the anonymous KDD reviewers for their helpful feedback. This work was supported by NSF grants CCF-1160904 and CCF-1409813.

REFERENCES

[1] Aaron B Adcock, Blair D Sullivan, and Michael W Mahoney. 2016. Tree decompositions and social graphs. *Internet Mathematics* 12, 5 (2016), 315–361.

[2] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284.

[3] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[4] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. *ICLR* (2018).

[5] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic training of graph convolutional networks with variance reduction. In *ICML*.

[6] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*.

[7] Jörg Flum, Markus Frick, and Martin Grohe. 2002. Query Evaluation via Tree-decompositions. *J. ACM* (2002).

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*.

[9] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin* (2017).

[10] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* (2016).

[11] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *NeurIPS*.

[12] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. In *NeurIPS*.

[13] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *PVLDB* (2017).

[14] Thomas N Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. *ICLR* (2017).

[15] Nils Kriege and Petra Mutzel. 2012. Subgraph matching kernels for attributed graphs. *ICML* (2012).

[16] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2018. Towards Efficient Large-Scale Graph Neural Network Computing. *CoRR* (2018).

[17] Elchanan Mossel and Sebastien Roch. 2007. On the submodularity of influence in social networks. In *STOC*.

[18] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *ACL* (2015).

[19] Ruo-Chun Tzeng and Shan-Hung Wu. 2019. Distributed, Egocentric Representations of Graphs for Detecting Critical Structures. In *ICML*.

[20] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. *ICLR* (2018).

[21] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of small-world networks. *Nature* 393, 6684 (1998), 440.

[22] Felix Wu, Tianyi Zhang, Amauri H. Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. *ICML* (2019).
[23] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. 2019. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596* (2019).
[24] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *ICLR*.
[25] Pinar Yanardag and S.V.N. Vishwanathan. 2015. Deep Graph Kernels. In *KDD*.
[26] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *KDD*.
[27] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*.
[28] Jiaxuan You, Rex Ying, and Jure Leskovec. 2018. Position-aware graph neural networks. *ICML* (2018).
[29] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. Graphsaint: Graph sampling based inductive learning method. In *ICLR*.
[30] Marinka Zitnik and Jure Leskovec. 2017. Predicting multicellular function through multi-layer tissue networks. *ISMB* (2017).