On Greedy Approaches to Hierarchical Aggregation

Alexandra Porter
Department of Computer Science
Stanford University
Stanford, CA
amporter@stanford.edu

Mary Wootters
Departments of Computer Science
and Electrical Engineering
Stanford University
Stanford, CA
marykw@stanford.edu

2021 IEEE International Symposium on Information Theory (ISIT) | 978-1-5386-8209-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISIT45174.2021.9517814

***Abstract*—We analyze greedy algorithms for the *Hierarchical Aggregation* (HAG) problem, a strategy introduced in [Jia et al., KDD 2020] for speeding up learning on Graph Neural Networks (GNNs). The idea of HAG is to identify and remove redundancies in computations performed when training GNNs. The associated optimization problem is to identify and remove the *most* redundancies.**

Previous work introduced a greedy approach for the HAG problem and claimed a 1-1/e approximation factor. We show by example that this is not correct, and one cannot hope for better than a 1/2 approximation factor. We prove that this greedy algorithm does satisfy some (weaker) approximation guarantee, by showing a new connection between the HAG problem and maximum matching problems in hypergraphs. We also introduce a second greedy algorithm which can out-perform the first one, and we show how to implement it efficiently in some parameter regimes. Finally, we introduce some greedy heuristics that are much faster than the above greedy algorithms, and we demonstrate that they perform well on real-world graphs.

A full version of this paper is accessible at: https://arxiv.org/abs/2102.01730

I. Introduction

In this work, we analyze an optimization problem that arises from *Hierarchical Aggregration* (HAG), a strategy that was recently introduced in [4] for speeding up learning on *Graph Neural Networks* (GNNs).

At a high level, HAG identifies redundancies in the computations performed in training GNNs and elimates them. This gives rise to an optimization problem, the *HAG problem,* which is to find and eliminate the most redundancies possible. In this paper, we study greedy algorithms for this optimization problem.

Our contributions are as follows.

1) The work [4] proposed a greedy algorithm, which we call FullGreedy, for the HAG optimization problem, and claimed that it gives a $1 - 1/e$ approximation. Unfortunately, this is not true, and we show by example that one cannot hope for a better than a $1/2$ approximation. We prove a new approximation guarantee for FullGreedy in Theorem 9. In more detail, we are able to establish a $\frac{1}{d}(1-1/e)$ approximation ratio for a related objective function, where d is a parameter of the problem ($d = 2$ is a reasonable value).

2) We propose a second greedy algorithm, PartialGreedy, for the HAG optimization problem. We show by example that this algorithm can obtain strictly better results than FullGreedy mentioned above. It is not obvious that PartialGreedy is efficient, and in Theorem 8 we show that it can be implemented in polynomial time in certain parameter regimes.

3) While both of the greedy algorithms we study are "efficient," in the sense that they are polynomial time, they can still be slow on massive graphs. To that end, we introduce greedy heuristics and demonstrate that they perform well on real-world graphs.

Our approach is based on a new connection between the HAG problem and a problem related to maximum hypergraph matching. We use this connection both in our approximation guarantees for FullGreedy and in our efficient implementation of PartialGreedy. We note that there are somewhat similar results for the problems of network computation [5] and computing over sensor networks [2] using different tools. Our work may also be useful for these applications.

II. Preliminaries and Problem Definition

A. Abstraction of Graph Neural Networks

Let $G = (V, E)$ be a directed graph that represents some underlying data. For example, G could arise from a social network, a graph of transactions, and so on. The goal of a GNN defined on G is to learn a *representation* $\mathbf{h}_v \in \mathbb{R}^s$ for each $v \in V$, with the goal of minimizing some loss function $\mathcal{L}(\{\mathbf{h}_v : v \in V\})$.[1] GNNs were

[1] A typical set-up for a GNN might be the following. The representations $\mathbf{h}_v$ are some function f of the features $\mathbf{x}_u$ and representations $\mathbf{h}_u$ of the nodes u in the neighborhood of v; a prediction $\mathbf{o}_v$ is a function g of the $\mathbf{x}_v$ as well as of $\mathbf{h}_v$; and both f and g are fully connected feed-forward neural networks. However, the details of GNNs will not actually matter for this work.

introduced by [11], and have numerous extensions and applications [1], [3], [6], [12], [13].

Learning these representations $\mathbf{h}_v$ follows the following abstract process (see the full version for pseudocode). First, each node v calls a function AGGREGATE on the values $\mathbf{h}_w$ for $w \in \Gamma_{in}(v)$, resulting in an aggregated value a_v. Here, $\Gamma_{in}(v)$ represents the set of nodes $w \in V$ so that $(w, v) \in E$. Next, the node v calls a function UPDATE on a_v and the current value of $\mathbf{h}_v$ to obtain an updated $\mathbf{h}_v$. Then this repeats. Here, the function AGGREGATE can be as simple as a summation (e.g. in GCN [6]), or it can be more complicated (e.g. in GraphSAGE-P [3]). In this work, we assume that AGGREGATE does not depend on the order of its inputs and can be applied hierarchically. This is often the case in GNNs (see [4] for more details).

B. Hierarchical Aggregation

The starting point for our work is the paper [4], which showed that there are significant improvements to be made (up to 2.8x, empirically), by cutting redundant computations. To see where redundant computations might arise, suppose that two nodes $u, v \in G$ have a large shared out-neighborhood $\Gamma_{out}(u) \cap \Gamma_{out}(v)$. In the aggregation procedure described above, we would call AGGREGATE on the nodes u and v many times, once for each node in this shared out-neighborhood. However, we can save computation by introducing an *intermediate node* m so that $\Gamma_{in}(m) = \Gamma_{in}(u) \cap \Gamma_{in}(v)$ and $\Gamma_{out}(m) = \Gamma_{out}(u) \cap \Gamma_{out}(v)$, and then disconnecting u and v from its original shared out-neighborhood. Then, we only call AGGREGATE on u and v once, and we can use the computation stored at m many times.

The Hierarchical Aggregation (HAG) problem is to find the best way to introduce such intermediate nodes.

Definition 1 (GNN Computation Graph)**.** *Given a directed graph $G = (V, E)$, the **GNN Computation Graph** $\mathcal{G}$ for G is a bipartite graph $(L, R, \mathcal{E})$, where L and R are copies of V, and, for $u \in L$ and $v \in R$, $(u_L, v_R) \in \mathcal{E}$ if and only if $(u, v) \in E$. We use $\Gamma_{in}(v)$ to denote the set of in-neighbors of a vertex v in $\mathcal{G}$, and we use $\Gamma_{out}(v)$ to denote the set of out-neighbors of v in $\mathcal{G}$.*

Definition 2 (HAG Computation Graph)**.** *Given a directed graph $G = (V, E)$, a **HAG Computation Graph** $\hat{\mathcal{G}}$ for G is a graph $(\hat{\mathcal{V}}, \hat{\mathcal{E}})$, where $\hat{\mathcal{V}} = L \cup M \cup R$ and L and R are copies of V. $\hat{\mathcal{E}}$ contains directed edges from L to M, from M to R, and possibly within M, and the following property holds. For every directed edge $(u, v) \in E$, there is a unique directed path from u_L to v_R in $\hat{\mathcal{G}}$. We use $\hat{\Gamma}_{in}(v)$ to denote the set of in-neighbors of a vertex v in $\hat{\mathcal{G}}$, and we use $\hat{\Gamma}_{out}(v)$ to denote the set of out-neighbors edges of v in $\hat{\mathcal{G}}$. When no edges in $\hat{\mathcal{E}}$ have both endpoints in M, so $\hat{\mathcal{G}}$ is tripartite, we call this a* single-layer *HAG computation graph. When there exists integer d such that for all $w \in M$, $|\hat{\Gamma}_{in}(w)| = d$, then we call $\hat{\mathcal{G}}$ a* d-HAG computation graph*.*

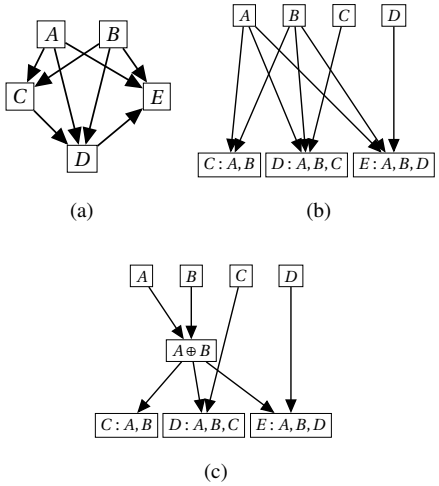


Fig. 1: Example of hierarchical aggregation. (a) A directed graph $G = (V, E)$, (b) the GNN computation graph $\mathcal{G}$, and (c) a possible HAG computation graph $\hat{\mathcal{G}}$, equivalent to $\mathcal{G}$. The notation $[D : A, B, C]$ means that node D is requesting information from nodes A, B, C. The notation $A \oplus B$ means that this intermediate node computes AGGREGATE(A, B).

See Figure 1 for an example of a GNN computation graph and a HAG computation graph arising from a directed graph G. We say that a GNN computation graph $\mathcal{G}$ and a HAG computation graph $\hat{\mathcal{G}}$ are **equivalent** if they are both computation graphs for the same underlying graph G. For a vertex v in $\hat{\mathcal{G}}$, we define the **cover** of v, $\text{cover}(v)$, to be the set of all nodes $u \in L$ so that there is a path from u to v in $\hat{\mathcal{G}}$.

Given a HAG computation graph $\hat{\mathcal{G}}$, we can reorganize the GNN computation described above in order to aggregate computations at the intermediate nodes in M. That is, we order the vertices of M so that for any $v \in M$, the vertices in $\hat{\Gamma}_{in}(v) \cap M$ appear before v (this is possible since $\hat{\mathcal{G}}$ is a DAG). Then, in this order, each node $v \in M$ calls AGGREGATE on the values $\mathbf{h}_u$ for $u \in \hat{\Gamma}_{in}(v)$, and finally each vertex $r \in R$ can call AGGREGATE on the values $\mathbf{h}_u$ from the nodes $u \in \hat{\Gamma}_{in}(r)$, and then UPDATE on the result. (See the full version for pseudocode).

In [4], the following cost function was considered. We say that the cost of a computation graph $\mathcal{G}$ with vertices $\mathcal{V}$ and right-hand side R (either a HAG computation graph or a GNN computation graph) is

$$\text{cost}(\mathcal{G}) = c_{Agg} \sum_{w \in \mathcal{V}} (|\Gamma_{in}(w)| - 1) + c_{Up} \cdot |R|$$

where c_{Agg} and c_{Up} are some constants representing the cost of an aggregation and an update respectively. We define the value of a HAG computation graph $\hat{\mathcal{G}}$ to be proportional to the amount of cost that it saves.

Definition 3. *The **value** of a HAG Computation Graph $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$ is given by*

$$\text{value}(\hat{\mathcal{G}}) = \frac{1}{c_{Agg}} \left(\text{cost}(\mathcal{G}) - \text{cost}(\hat{\mathcal{G}}) \right)$$

$$= \sum_{v \in M} \left[|\hat{\Gamma}_{out}(v) \cap R| (|\text{cover}(v)| - 1) - (|\hat{\Gamma}_{in}(v)| - 1) \right].$$

The equality above is verified in [10].

C. The HAG Problem

Given the above setup, we can formally define the HAG problem. We additionally take two parameters d and k. The parameter d is a bound on the left-degree of the aggregation nodes (for example, the work [4] considered $d = 2$ in their algorithm). The parameter k is a budget on the number of intermediate nodes allowed.

Definition 4 (HAG problem)**.** *Let d be an integer. The d-HAG problem is the following. Given a graph G and a node budget k, find a HAG computation graph $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$ for G with the largest value, so that $|M| \leq k$ and $|\hat{\Gamma}_{in}(w)| = d$ for all $w \in M$.*

We also define a *single-layer* variation of the problem. The single layer variation is faster to compute and we show empirically that single-layer solutions achieve almost as much value as general multi-layer solutions.

Definition 5 (single-layer HAG problem)**.** *Let d be an integer. The single-layer d-HAG problem is defined as the d-HAG problem with the additional constraint that $\hat{\mathcal{G}}$ be tripartite.*

We note that if $\hat{\mathcal{G}}$ is a single-layer d-HAG computation graph, value can be simplified: $value(\hat{\mathcal{G}}) = \sum_{v \in M} (|\hat{\Gamma}_{out}(v)| - 1)(|\hat{\Gamma}_{in}(v)| - 1)$.

III. Greedy Algorithms

We study two natural greedy algorithms for the HAG problem. We call these two algorithms FullGreedy and PartialGreedy. Intuitively, FullGreedy greedily choose an internal node, with all of its incoming and outgoing edges, and fixes it. On the other hand, PartialGreedy greedily chooses an internal node with all of its incoming edges, but re-optimizes the outgoing edges when each new internal node is added.

Definition 6. *Given HAG computation graph $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$ with $\hat{\mathcal{V}} = L \cup M \cup R$, for $X, Y \in \{L, M, R\}$ let $T_{\hat{\mathcal{G}}}(X, Y)$ denote the set edges in $\hat{\mathcal{G}}$ that either connect X and Y in $\hat{\mathcal{G}}$, or connect Y to Y in $\hat{\mathcal{G}}$:*

$$T_{\hat{\mathcal{G}}}(X, Y) := (\hat{\mathcal{E}} \cap (X \times Y)) \cup (\hat{\mathcal{E}} \cap (Y \times Y)).$$

We begin with the algorithm FullGreedy, proposed by [4]. This is shown in Algorithm 1. Note that in Line 5, Algorithm 1 is finding the set C of size d that maximizes the number of nodes in R that request all of the nodes in C. This C is then used as the in-neighborhood for the new internal node at that step.

Algorithm 1 Greedy Algorithm FullGreedy

Require: GNN Computation Graph $\mathcal{G} = (L, R, \mathcal{E})$; aggregation node limit k, aggregation in-degree d.

1: $M_0 \leftarrow \emptyset$
2: $\hat{\mathcal{E}}_0 \leftarrow \mathcal{E}$
3: $\hat{\mathcal{G}}_i \leftarrow (L \cup M_0 \cup R, \hat{\mathcal{E}}_0)$
4: **for** $i = 1, ..., k$ **do**
5: $\quad C \leftarrow \arg\max_{C \subseteq L \cup M_{i-1}, \text{ s.t. } |C| = d} |\bigcap_{v \in C} \hat{\Gamma}_{out}(v) \cap R|$
6: $\quad R_C \leftarrow \bigcap_{v \in C} \hat{\Gamma}_{out}(v) \cap R$
7: $\quad M_i \leftarrow M_{i-1} \cup \{v_i\}$ ▷ add a new vertex v_i to M
8: $\quad$ Construct the new edge set $\hat{\mathcal{E}}_i$:
- $\hat{\mathcal{E}}_i \leftarrow \hat{\mathcal{E}}_{i-1}$
- Add edge (ℓ, v_i) to $\hat{\mathcal{E}}_i$ for all $\ell \in C$.
- Add edge (v_i, r) to $\hat{\mathcal{E}}_i$ for all $r \in R_C$.
- Remove any edges (ℓ, r) from $\hat{\mathcal{E}}_i$ with $\ell \in C$ and $r \in R_C$.

9: $\quad \hat{\mathcal{G}}_i \leftarrow (L \cup M_i \cup R, \hat{\mathcal{E}}_i)$

We next consider a greedy algorithm, PartialGreedy, in which the edges between the intermediate nodes and receiving nodes are re-assigned at each iteration. In particular, at the i^{th} step the edge set $T_{\hat{\mathcal{G}}_i}(M_i, R)$ is chosen to be optimal given M_i and $T_{\hat{\mathcal{G}}_i}(L, R)$, rather than constructed by adding edges to the set $T_{\hat{\mathcal{G}}_{i-1}}(M_{i-1}, R)$ from the previous step. Algorithm 2 describes this process. Note that in Line 8 of Algorithm 2, $\mathcal{S}_C$ is the set of all graphs $\hat{\mathcal{G}}^{(C)}$ that extend the left-hand side $T_{\hat{\mathcal{G}}_{i-1}}(L, M)$ of $\hat{\mathcal{G}}_{i-1}$ by adding an intermediate node v with $\Gamma_{in}(v) = C$.

Remark 7. *Note that both Algorithms 1 and 2 can be easily modified to find a single-layer solution.*

IV. Efficiency

We note that FullGreedy (Algorithm 1) is clearly polynomial time if d is constant. In particular, the argmax can be naively implemented in time $O(n^d)$.

On the other hand, it is not clear that PartialGreedy (Algorithm 2) is even polynomial time (in n), because it is not clear how to solve the optimization problem in Line 9. However, we show that in fact this can be re-cast as a matching problem in hypergraphs, which is efficient in certain parameter regimes.

Theorem 8. *Suppose that either:*

- *$d = 2$, and Alg. 2 is restricted to a single layer (see Remark 7); or*

Algorithm 2 Greedy Algorithm PARTIALGREEDY

Require: GNN Computation Graph $\mathcal{G} = (L, R, \mathcal{E})$; aggregation node limit k, aggregation in-degree d.

$M_0 \leftarrow \emptyset$
$\hat{\mathcal{E}}_0 \leftarrow \mathcal{E}$
$\hat{\mathcal{G}}_0 \leftarrow (L \cup M_0 \cup R, \hat{\mathcal{E}}_0)$
for $i = 1, ..., k$ **do**
  Suppose that $\hat{\mathcal{G}}_{i-1}$ has vertices $\hat{\mathcal{V}}_{i-1} = L \cup M_{i-1} \cup R$.
  for $C \subseteq L \cup M_{i-1}$ s.t. $|C| = d$ **do**
    $M_i \leftarrow M_{i-1} \cup \{v_i\}$ ▷ Add a new vertex v_i

$$\mathcal{S}_C = \left\{ \begin{array}{c} \hat{\mathcal{G}}^{(C)} = (L \cup M_i \cup R, \hat{\mathcal{E}}^{(C)}) : \\ \hat{\mathcal{G}}^{(C)} \text{ is a d-HAG computation graph} \\ \text{equivalent to } \mathcal{G} \text{ and } T_{\hat{\mathcal{G}}^{(C)}}(L, M_i) \\ = T_{\hat{\mathcal{G}}_{i-1}}(L, M_{i-1}) \cup (C \times \{v_i\}) \end{array} \right\}$$

    $\hat{\mathcal{G}}^{(C)}_{opt} \leftarrow \arg\max_{\hat{\mathcal{G}}^{(C)} \in \mathcal{S}_C} \text{value}(\hat{\mathcal{G}}^{(C)})$
  $\hat{\mathcal{G}}_i \leftarrow \arg\max_C \text{value}(\hat{\mathcal{G}}^{(C)}_{opt})$

- $d = O(1)$ *and* $\deg(G) = O(1)$*, where* $\deg(G)$ *is the maximum degree of the original graph* G*.*

Then Alg. 2 can be implemented in polynomial time.

To prove Theorem 8 (full proof in [10]), we first show that for some HAG computation graph $\hat{\mathcal{G}}$, there is an underlying bijection between the possible edge sets $T_{\hat{\mathcal{G}}}(M, R)$ and matchings on a related weighted hypergraph H. From this, we get a direct relationship between the value of $\hat{\mathcal{G}}$ and the value of the matching on H that corresponds to $T_{\hat{\mathcal{G}}}(M, R)$. We then show that when $d = O(1)$ is a constant, PARTIALGREEDY (Algorithm 2) can be implemented using a polynomial number of maximum weighted-hypergraph matching problems. In particular, when $d = 2$ or when $\deg(G)$ (the degree of the underlying graph) is constant, we can implement Algorithm 2 in polynomial time.

V. APPROXIMATION GUARANTEES

We begin with FULLGREEDY. The work [4] introduced FULLGREEDY and claimed that it gives a $1 - 1/e$ approximation, in the sense that $\text{value}(\hat{\mathcal{G}}_{greedy}) \geq \left(1 - \frac{1}{e}\right) \text{value}(\hat{\mathcal{G}}_{opt})$. Unfortunately, it can be shown by example (see full version [10]) that this is not correct, and we cannot hope for better than a $1/2$ approximation.

We analyze FULLGREEDY (Algorithm 1) in the single-layer case. In Section VI, we show that the single-layer HAGs are nearly as good as multi-layer HAGS on real-world graphs. Our main theorem is the following.

Theorem 9. *For a* d*-HAG computation graph* $\hat{\mathcal{G}}$ *with* k *internal nodes, define*

$$\widetilde{\text{value}}(\hat{\mathcal{G}}) := \text{value}(\hat{\mathcal{G}}) + k(d-1).$$

Then the single-layer d*-HAG computation graph returned by* FULLGREEDY *(Algorithm 1, with a restriction to single-layer; see Remark 7)* $\hat{\mathcal{G}}_{greedy}$ *satisfies*

$$\widetilde{\text{value}}(\hat{\mathcal{G}}_{greedy}) \geq \frac{1}{d}\left(1 - \frac{1}{e}\right) \widetilde{\text{value}}(\hat{\mathcal{G}}^*),$$

where $\hat{\mathcal{G}}^*$ *is the* d*-HAG computation graph with the largest value (and also the largest* $\widetilde{\text{value}}$*).*

We are not able to establish an approximation ratio for the function $\text{value}(\cdot)$ itself, although we conjecture that a similar result holds.

The idea of the proof is as follows (see [10] for details). It is a standard result that greedy algorithms for submodular functions achieve a $1 - 1/e$ approximation ratio; this was the approach taken by [4]. However, the FULLGREEDY objective function is not technically submodular, since the order of the inputs matters, and this prevents the $1-1/e$ approximation result from being true. However, we can use the connection to hypergraph matching developed in the proof of Theorem 8 (see [10]) in order to translate the objective function of FULLGREEDY to an objective function where the order does not matter, at the cost of a factor of d. This results in a $\frac{1}{d}(1 - 1/e)$ approximation ratio for $\widetilde{\text{value}}$.

Like FULLGREEDY, PARTIALGREEDY also cannot achieve an approximation ratio better than $1/2$. We prove this by example in [10]. (The example also shows that the objective function that PARTIALGREEDY is greedily optimizing is not submodular.) However, it can do strictly better than FULLGREEDY on certain graphs.

Proposition 10. *There are graphs for which* PARTIALGREEDY *is strictly better than* FULLGREEDY.

This is also shown by example in [10]. Thus, the algorithm that runs both FULLGREEDY and PARTIALGREEDY and takes the better of the two achieves at least the approximation guarantee of Theorem 9, and can sometimes do strictly better than FULLGREEDY.

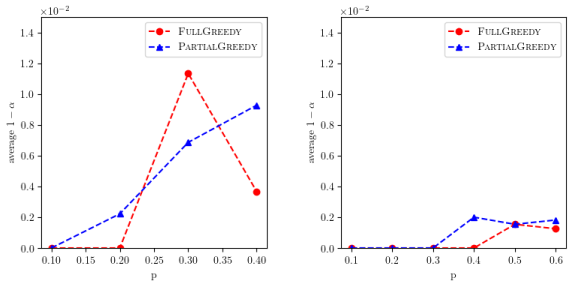
VI. EXPERIMENTAL RESULTS

We first show that multi-layer HAG graphs do not have a significantly higher value for small k compared to single-layer HAG graphs; this justifies our focus on single-layer HAG graphs in Theorem 9. In Table I we show a comparison of FULLGREEDY single-layer and multi-layer results for three datasets: a Facebook dataset [9], an Amazon co-purchases dataset [7] (the subset from March 2nd, 2003), and the Email-EU dataset [8][2]. For all three, the relative benefit of multi-

[2]All three of these datasets can be found at snap.stanford.edu/data

Dataset	Facebook	Amazon	Email-EU
$\mu^{(1)}$ (single-layer)	8636.09	1800.73	3088.73
$\mu^{(m)}$ (multi-layer)	8945.83	1806.29	3260.11
μ (% improvement)	3.2%	0.22%	4.9%
σ (std. dev. of improvement)	1.02782	0.216026	1.674153

TABLE I: The improvement of multi-layer over single-layer for FULLGREEDY on real-world datasets averaged over $k = 1, ..., 100$. We compute $v_1^{(1)}, ..., v_{100}^{(1)}$ and $v_1^{(m)}, ..., v_{100}^{(m)}$. $v_i^{(1)}$ is the value achieved by single-layer FULLGREEDY with $k = i$; v_j^m is the value achieved by multi-layer FULLGREEDY for $k = j$. We compute $\mu^{(m)} = \frac{1}{100}\sum_{i=1}^{k} v_i^{(1)}$, $\mu^{(m)} = \frac{1}{100}\sum_{i=1}^{k} v_i^{(m)}$, $\mu = \frac{1}{100}\sum_{i=1}^{100} \frac{v_i^{(m)} - v_i^{(1)}}{v_i^{(1)}}$ and $\sigma = \sqrt{\frac{1}{100}\sum_{i=1}^{100}(\frac{v_1^{(m)} - v_1^{(1)}}{v_1^{(1)}} - \mu)^2}$.



(a) (b)

Fig. 2: We compare FULLGREEDY and PARTIALGREEDY to the optimal HAG computation graph on a set of 50 Erdős-Rényi graphs $G(n, p)$ with $n = 15$. The y-axis plots average values of $1 - \alpha$, where α is the approximation ratio. The x-axis plots the parameter p. Shown are (a) $k = 2$ and (b) $k = 3$.

layer HAG is low.

We next compare single-layer FULLGREEDY and PARTIALGREEDY to the optimal single-layer solution (computing the optimum is only tractable for limited graph parameters even in the single-layer case, so we did not consider multi-layer). Figure 2 shows the quantity $1 - \alpha$, where α is the approximation ratio $\text{value}(\hat{\mathcal{G}}_{greedy})/\text{value}(\hat{\mathcal{G}}_{opt})$, for Erdős-Rényi graphs $G(n, p)$ with $n = 15$ and various values of p. Higher values of p result in approximation ratios slightly further from 1 for both $k = 2$ and $k = 3$, although in all experiments the approximation ratios are quite close to 1 for both algorithms.

While FULLGREEDY and PARTIALGREEDY are much faster in practice than computing the optimal solution, they are still computationally intensive for large values of k and large datasets. In this section we describe two alternative heuristics, DEGREEHEURISTIC and HUBHEURISTIC, which only achieve a fraction of the value of FULLGREEDY, but compute the HAG computation graph significantly faster.

	DEGREEHEURISTIC vs. FULLGREEDY		HUBHEURISTIC vs FULLGREEDY	
Dataset	Value Ratio	Runtime Ratio	Value Ratio	Runtime Ratio
Amazon	0.0699	0.123	0.629	0.124
Email-EU	0.558	0.0548	0.410	0.107
Facebook	0.376	0.0408	0.313	0.0894

TABLE II: For each dataset, FULLGREEDY, DEGREEHEURISTIC and HUBHEURISTIC were run 10 times with $k = 100$. Value Ratio is computed as the value of the DEGREEHEURISTIC (resp. HUBHEURISTIC) result divided by the value of the FULLGREEDY result for the first (resp. third) column. Runtime Ratio is analogous.

DEGREEHEURISTIC starts by ranking all of the vertices of the input graph $G = (V, E)$ by degree: $\{v_i\}_{i=1}^n$ with $\Gamma_{out}(v_i) \geq \Gamma_{out}(v_{i+1})$ for $i = 1, ..., n$. It then takes the top k adjacent pairs of the sequence (i.e., $(v_1, v_2), (v_2, v_3), \ldots, (v_{2k-1}, v_{2k})$) as the covers of the k aggregation nodes and constructs a single-layer 2-HAG computation graph. The out-edges of the aggregation nodes are assigned greedily in the same cover order $(v_1, v_2), (v_2, v_3), ...$ based on degree. We compare this heuristic to FULLGREEDY for value and runtime in Table II. This method performs decently on the Facebook and Email-EU datasets, and significantly worse on the Amazon purchasing network. We conjecture that this is because the Amazon network has has a significantly lower average degree (about 2.8) than the other two sets (about 22 for Facebook and 25 for Email-EU).

HUBHEURISTIC is based on searching for "good" intermediate aggregation nodes around high-degree nodes of G. This algorithm is motivated by the frequency with which triangles appear in real-datasets. HUBHEURISTIC also starts by ranking the vertices from highest to lowest degree as $\{v_i\}_{i=1}^n$. Then for $v_1, ..., v_k$ the heuristic does the following: for each $u \in \Gamma_{in}(v_i)$, compute the value of adding aggregation node with cover $\{v_i, u\}$. Then a new node m is added with cover $\{v_i, u\}$ using the u that allows for maximal out-edges from m. This process is repeated for $v_1, .., v_k$ in order, so it is greedy in the sense that out neighbors of previous aggregation nodes remain the same during subsequent iterations. We compare HUBHEURISTIC to FULLGREEDY for value and runtime, shown in Table II.

ACKNOWLEDGEMENTS

We thank Zhihao Jia, Rex Ying, and Jure Leskovec for helpful conversations.

REFERENCES

[1] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[2] Arvind Giridhar and Praveen R Kumar. Computing and communicating functions over sensor networks. *IEEE Journal on selected areas in communications*, 23(4):755–764, 2005.

[3] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[4] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. Redundancy-free computation graphs for graph neural networks. *arXiv preprint arXiv:1906.03707*, 2019.

[5] Nikhil Karamchandani, Rathinakumar Appuswamy, and Massimo Franceschetti. Distributed computation of symmetric functions with binary inputs. In *2009 IEEE Information Theory Workshop on Networking and Information Theory*, pages 76–80. IEEE, 2009.

[6] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[7] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5–es, 2007.

[8] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[9] Julian J McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *NIPS*, volume 2012, pages 548–56. Citeseer, 2012.

[10] Alexandra Porter and Mary Wootters. On greedy approaches to hierarchical aggregation. *arXiv preprint arXiv:2102.01730*, 2020.

[11] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[12] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[13] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.