Graph Neural Networks for Selection of Preconditioners and Krylov Solvers [*]

Ziyuan Tang
Department of Computer Science
University of Minnesota
tang0389@umn.edu

Hong Zhang
Mathematics and Computer Science Division
Argonne National Laboratory
hongzhang@anl.gov

Jie Chen
MIT-IBM Watson AI Lab
IBM Research
chenjie@us.ibm.com

Abstract

Solving large sparse linear systems is ubiquitous in science and engineering, generally requiring iterative solvers and preconditioners since many problems cannot be solved efficiently by using direct solvers. The practical performance of solvers and preconditioners is sometimes beyond theoretical analysis, however, and an optimal choice calls for intuition from domain experts and knowledge of the hardware, as well as trial and error. In this work we propose a new method for optimal solver-preconditioner selection using graph neural networks as a complementary solution to laborious expert efforts. The method is based on the graph representation of the problem and the idea of integrating node features with graph features via graph convolutions. We show that our models outperform traditional machine learning models investigated by a margin of 25% under the NDCG evaluation metric.

1 Introduction

Solving linear systems of the form

$$A\mathbf{x} = \mathbf{b}\ , \tag{1}$$

where A is a large and sparse matrix, is a core computing task in numerous scientific and engineering problems, such as piezoelectric crystal vibration simulation [1], superconductor modeling [2], and airborne electromagnetic problems [3]. These problems typically scale poorly for direct solvers because of constraints of memory and computational resources. Therefore, iterative methods, as well as preconditioners, are continuously being developed to improve effectiveness and accelerate the computations [4]. Because of the wide variety of algorithms, data structures, and hardware architectures, however, selecting the optimal combination of a solver and a preconditioner for a given linear system is difficult for application developers and researchers and may require extensive background and expertise in numerical analysis and high-performance computing. For example, the scientific computing library PETSc [5, 6, 7] already includes 17 iterative methods, 17 preconditioners, and 6 matrix formats (not counting support for third-party packages). The growing pool of new

[*]This material was based upon work funded by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This work was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0016140.

techniques becomes a barrier to productivity. Therefore, a long-standing goal has been to automate the process and ease the reliance on numerical analysis and hardware knowledge. It becomes more crucial when the problem size is enormous and its solution requires parallel computation resources.

Achieving this goal is a challenging task, however, because the decision cannot be made based on the algorithm analysis alone: the problem characteristics also must be taken into account. While most existing machine learning methods rely on finding features that are relevant for the performance prediction but often expensive to compute, we propose a machine learning method that uses graph neural networks (GNNs) for fast selection of preconditioners and sparse iterative solvers based upon features that are easy to compute on graphs. Our work is motivated by the observations that matrices of sparse linear systems entail a graph interpretation and GNNs has shown promises for learning and encapsulating the domain knowledge required for efficient solution of sparse linear systems [8].

For a proof of concept, only selected built-in Krylov solvers and preconditioners in PETSc are considered here for the performance benchmark. Users with access to other software libraries can follow a similar procedure and create a suitable training dataset to obtain specific predictions for solver and preconditioner selection in their computing environment.

Our main contributions are listed below.

- We introduce a general graph representation of large sparse matrices so that the local structure of the matrix is exploited by the graph convolution process.
- We formulate a score-based metric that accounts for the budget of running time and accuracy requirement of the target problem, thus allowing for a trade-off between these two factors.
- We analyze four known graph convolution models through the proposed GNN architecture that combines node features with graph features.
- We introduce a two-level pooling strategy to simulate the mechanism by which human experts select iterative solvers and preconditioners.

2 Related work

Over the years, the literature has reported automatic methodologies for selecting preconditioned linear solvers for sparse linear systems. In 1996, Barrett et al. [9] proposed a poly-iterative approach that executed several Krylov methods simultaneously for the same problem to predict the best solver. Bhowmick et al. [10, 11] presented ideas for applying alternating decision trees to choose linear solvers adaptively. Holloway and Chen [12] extended the methodology to neural networks, while Kuefler and Chen [13] evaluated the effectiveness of reinforcement learning. Bhowmick et al. [14] subsequently investigated other machine learning (ML) methods, including k-nearest neighbor, naive Bayes, and support vector machines. Similar research includes that of Eller et al. [15], who introduced a dynamic model especially for adaptive hydraulics problems during transient simulations. Kotthoff et al. [16] compared ML techniques to address the more general problem of algorithm selection.

Software and systems for tuning and recommending numerical solvers have also been developed. In particular, PYTHIA [17] is a knowledge-based recommender system for elliptic partial differential equations. In 2010, Eijkhout and Fuentes [18] developed a self-adapting large-scale solver architecture (SALSA) for multistage solver selection, with the help of their matrix feature extraction software named AnaMod [19]. Later, a comprehensive taxonomy and framework were given in the project Lighthouse [20, 21, 22], which targets preconditioned solvers in PETSc and Trilinos libraries [23].

These prior works rely heavily on a number of carefully chosen features to predict the performance of preconditioners and solvers. However, useful features such as condition numbers are expensive to compute, and inexpensive features such as average node degree are not very effective in distinguishing matrices with different sparse structures. It is difficult to balance the effectiveness and cost of feature extraction with traditional methods. In this work, we consider GNN mainly for two reasons. First, it can efficiently utilize features related to spatial structure, which are usually easy to compute. Second, GNNs allow us to explore popular techniques in graph learning, such as virtual nodes [24]. These techniques offer opportunities to improve graph classification performance, thus potentially beneficial to the preconditioners and solvers selection problem.

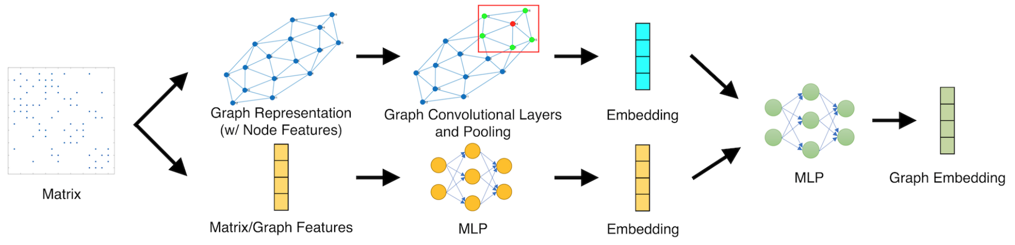

Matrix
Graph Representation (w/ Node Features)
Graph Convolutional Layers and Pooling
Embedding
Matrix/Graph Features
MLP
Embedding
MLP
Graph Embedding

Figure 1: Overview of the proposed GNN architecture.

3 Method

Figure 1 shows an overview of the proposed GNN architecture. An input matrix of the coordinate (COO) format is processed through two paths. In the first path, a graph with well-defined node features is generated. Then, by applying graph convolutional layers and pooling layers, we obtain a graph embedding. In the second path, matrix features are extracted and treated as graph features. A feature vector is processed by a neural network such as a multilayer perceptron (MLP). Thus, the second path also generates a graph embedding. The two separate embeddings are concatenated and integrated into a new graph embedding through another MLP. One may regard this output embedding as an input of a classifier, after applying a scaling function such as a sigmoid function. Note that these two paths are independent until the mixing step is taken, and in this framework one can remove either path or add new paths of a similar structure.

3.1 Problem formulation

The objective of the selection problem is defined as *multi-label classification*, where each class represents a combination of a solver and a preconditioner. Table 1 shows the Krylov iterative solvers and preconditioners used in our settings. In total, 33 classes are considered, excluding combinations where solvers and preconditioners are incompatible. Therefore, for any input matrix, the target label of our GNN model is a binary vector $y \in \{0,1\}^{33}$, with a value of 1 representing recommendation, and 0 otherwise.

Table 1: Available preconditioners and Krylov iterative solvers

Capability	Algorithm
Preconditioners	Block Jacobi + ILU(0), QMD reordering
	Block Jacobi + ILU(1), QMD reordering
	Block Jacobi + LU
	ASM(1)
	ASM(2)
	Hypre/BoomerAMG
	Hypre Euclid
	Parasails approximate inverse from Hypre
	Block Jacobi + GMRES
Krylov iterative solvers	CG
	GMRES(30)
	BiCGStab
	LSQR
	Flexible GMRES (inner GMRES)

The performance of solvers and preconditioners on a given matrix A arises from solving a linear system $A\mathbf{x} = \mathbf{b}$, where all elements in $\mathbf{b}$ are 1's. By applying various combinations, we record the running time t and the residual $r = \|\mathbf{b} - A\hat{\mathbf{x}}\|_2$ for each choice, where $\hat{\mathbf{x}}$ is the computed solution. Then, we use the following metrics to compute a score:

$$\text{score}(t, r) = \log(1 + w_1/t)\log(1 + w_2/r)\ , \tag{2}$$

where w_1, w_2 are user-defined weights (default 1) based on the budgets of running time and tolerance of residual error. For the cases that run out of time or fail to solve the system, the score will be 0. Equation 2 is designed to distinguish between the classes with exponential differences in residual or runtime cost.

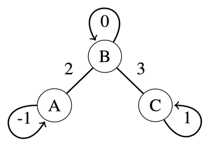
We define an empirical threshold equal to 90% of the highest score. Any class that has a score higher than the threshold will be labeled as 1, and others will be labeled as 0.

3.2 Graph representation and preprocessing

For any input matrix A, a graph is first generated based on the adjacency matrix of A. Then a self-loop is added to each node. The off-diagonal nonzero elements of A are attached to corresponding edges as edge features, and diagonal elements are attached to self-loops respectively. As a rule of thumb, diagonal elements are deemed more important than off-diagonal elements. Our graph representation emphasizes diagonal elements, since self-loops are visited more frequently throughout the message-passing process. Diagonal elements with larger magnitudes correspond to larger edge values on self-loops and thus have a larger impact on the computation graph.

Here we demonstrate this idea with an example of a 3×3 matrix.

$$A = \begin{bmatrix} -1 & 2 & 0 \\ 2 & 0 & 3 \\ 0 & 3 & 1 \end{bmatrix} \quad \rightarrow$$



Recalling that the essence of our GNN model is to predict the performance of solvers and preconditioners in solving linear systems, any modification on a linear system is reasonable as long as it does not affect the selection. Therefore, all columns and rows that contain only diagonal nonzero elements can be removed. In the context of graph interpretation, this refers to the operation of deleting isolated nodes. Diagonal matrices are removed from the dataset for the same reason. Furthermore, we add a small constant value to every self-loop in the graph because diagonal shifting is commonly applied to singular matrices in iterative linear solvers.

3.3 Node and graph features

As mentioned at the beginning of Section 3, the graph embedding is generated by processing two types of features. *Node features* attached to graph nodes are processed through graph convolutional layers, while *graph features* are processed via other neural network layers, for example, MLP. Then a mixing layer, for example another MLP, is performed to render a final graph embedding.

We introduce a node feature to describe diagonal dominance of row i. Assume that α_i denotes the ratio between magnitudes of diagonal and off-diagonal elements for each row,

$$\alpha_i = \begin{cases} \frac{|A_{ii}|}{\sum_{j\neq i}|A_{ij}|}, & \sum_{j\neq i}|A_{ij}| > 0\ ; \\ \infty, & \sum_{j\neq i}|A_{ij}| = 0\ . \end{cases} \tag{3}$$

Then, the node feature $\mathbf{x}_i$ for node i is defined as

$$\mathbf{x}_i = \frac{\alpha_i}{\alpha_i + 1}\ , \tag{4}$$

such that $\mathbf{x}_i \in [0, 1]$ and $\mathbf{x}_i = 0.5$ is the critical point of whether row i is diagonally dominant. Similarly, the diagonal decay of row i can be defined by replacing $\sum_{j\neq i}|A_{ij}|$ in Eq. 3 with $\max_{j\neq i}|A_{ij}|$. Both diagonal dominance and decay can also be computed for each column. Moreover, the local degree profile [25] is also considered.

Table 2 gives a complete list of graph features used in this paper. To reduce the cost of extracting features that have no significant impact on the classification accuracy, we use the reduced feature set provided by [21, 22]. Additionally, normalization or standardization must be performed when concatenating various features.

Table 2: List of node and graph features.

Capacity	Features
Node	Diagonal dominance (col & row) Diagonal decaying (col & row) Local degree profile
Graph	Average distance of nonzero to diagonal Number of nonzero elements 1-norm ∞-norm Column variability Minimum number of column non-zero elements Row variability Minimum number of row nonzero elements Number of diagonals that have non-zero elements Estimated condition number

3.4 Graph convolutional layers

We analyze four conventional graph convolution models:

- *Graph attention networks* (GATs). GATs [26, 27] support taking edge features as input, where edge features are multiplied with transformation matrices and then concatenated with node features.
- *The edge version of graph isomorphism networks* (GINEs) [28, 29]. GINEs aggregate edge features along with node features.
- *Graph convolutional networks* (GCNs) [30]. GCNs are different from the above models. GCNs can take only non-negative edge weights, so a sigmoid or softmax function is required for edge features.
- *A modified version of GraphSAGE* [31]. The aggregator is represented by

  $$\mathbf{x}_i \leftarrow W_1\mathbf{x}_i + W_2 \cdot \mathrm{mean}_{j\in\mathcal{N}(i)}\mathbf{x}_j e_{ji} \ , \tag{5}$$

  where $\mathbf{x}_i$ is the node embedding of node i; $\mathcal{N}(i)$ is the neighbor of node i; W_1, W_2 are training weights; and e_{ji} is the edge feature from node j to node i.

We make slight modifications to GCN and GraphSAGE in order to utilize both node and edge features. Edge features, in other words, nonzero elements of the matrix, can be arbitrary real values. This requires that the desired convolutional network be able to distinguish between A and its absolute counterpart $|A|$ if there are both positive and negative elements in A. It also should produce the same results for A and kA for any scalar $k \neq 0$.

3.5 Two-level pooling

By adapting the idea of *Multiset Pooling* [32], we designed a two-level pooling based on the fact that computations between distinct connected components are independent. In our approach, the graph embedding is generated in the following steps. First, node embeddings are aggregated within the connected components to which they belong. Then, the connected component embeddings are summarized to yield the graph embedding.

The message-passing mechanism can be viewed as a voting procedure in which the central node collects messages from its neighbor nodes and votes for the classification of the graph from its perspective via node embedding. Since each node can only reach the connected component to which it belongs, however, we aggregate the node embeddings within each connected component. Common embedding techniques include mean pooling, histogram of voting, and virtual node embedding.

Without treating each connected component independently, the selection of solvers and preconditioners for the whole matrix heavily depends on the most difficult connected components to solve, for example, the most ill-conditioned one. Therefore, we apply a min-pooling technique to connected component embeddings to generate the graph embedding, which indicates the solver and preconditioner that are suitable and efficient for all connected components.

To illustrate the second level of pooling, we use a simple example where we choose from two candidate preconditioned solvers for a block diagonal matrix that consists of two square matrices,

$$C = \left[\begin{array}{c:c} A & \\ \hdashline & B \end{array}\right] . \tag{6}$$

Here A and B are two distinct connected components of C. If both solvers can be applied to A and only the first solver is applicable to B, it gives rise to multi-labels $y_A, y_B \in \{0,1\}^2$, where $y_A = [1,1]$ and $y_B = [1,0]$. Then, the multi-label for C is $y_C = [1,0] = \text{min-pooling}(y_A, y_B)$.

4 Numerical results

The experiments for the GNN models were performed on a workstation with an NVIDIA RTX 3090 GPU and an Intel i7-11700KF CPU. Scikit-learn [33] was used to implement traditional ML methods. GNN frameworks were based mainly on PyTorch [34] and PyTorch-Geometric [35], and evaluation metrics were provided by TorchMetrics [36].

4.1 Benchmark dataset

To assess the performance of the GNN models, we built a benchmark using matrices from the SuiteSparse Matrix Collection [37]. Only square matrices $A \in \mathbb{R}^{n\times n}$ with a number of rows $1000 \leq m \leq 10000$ and a number of nonzero elements $\text{nnz}(A) \leq 200000$ were selected. The dataset contained 614 matrices and was split into training and testing datasets using 5-fold cross-validation. Each matrix resulted in 33 entries because of the 33 combinations of solvers and preconditioners. Each entry in the dataset consisted of the matrix ID, the solver choice, the preconditioner choice, the solution time (with negative values indicating failure reasons such as using an incompatible preconditioner, time out, or divergence) and the ℓ_2 residual norm. Each linear solve was run in parallel using 64 MPI processes on the KNL partition on Theta [38], a Cray XC40 supercomputer at Argonne National Laboratory.

4.2 Evaluation metrics

The evaluation metrics for multilabel classification in this paper includes *label ranking average precision* (LRAP) [39] and *normalized discounted cumulative gain* (NDCG) [40, 41, 42]. Both methods enable taking prediction probability as input and return scores in $[0,1]$, where higher scores indicate better classification accuracy. Note that the metric *true positive rate* (TPR) was occasionally used in the related literature, but we do not use it because it is meaningless for multi-label classification tasks. TPR can be arbitrarily high if one sets the decision threshold sufficiently low.

We denote the true labels by y and the prediction by $\hat{y}$. The definition of LRAP is as follows:

$$LRAP(y,\hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{\|y_i\|_0}\sum_{j:y_{ij}=1}\frac{|\mathcal{L}_{ij}|}{\text{rank}_{ij}} , \tag{7}$$

where n is the number of samples, $\mathcal{L}_{ij} = \{k : y_{ik} = 1, \hat{y}_{ik} > \hat{y}_{ij}\}$, $\text{rank}_{ij} = |\{k : \hat{y}_{ik} > \hat{y}_{ij}\}|$, and $\|\cdot\|_0$ computes the l_0-norm, which is equal to the number of nonzeros.

The DCG score is defined as follows:

$$DCG(y,\hat{y}) = \sum_{r=1}^{m}\frac{y_{f(r)}}{\log(1+r)} , \tag{8}$$

where m is the number of classes and f is a ranking function induced from y and $\hat{y}$. The NDCG score is the DCG score divided by the DCG score of y, which is considered as the ideal DCG score.

4.3 Comparison with traditional ML methods

We compared the proposed GNN models with traditional ML methods supporting multi-label classification, such as random forest (RF) [43], k-nearest neighbor (kNN) [44], multilayer perceptron (MLP) [45, 46], and Ridge Classifier. GNN models in our experiments follow the same architecture and differ solely from the convolutional layers. Each graph convolution model consists of five layers. The MLP for processing graph features has two layers, and the MLP for combining node and graph embeddings has two layers. More GNN layers can cause oversmoothing, and effective training strategies [47] need to be used for such situations.

Table 3 gives 5-fold average LRAP and NDCG scores and standard errors of the considered methods on the test dataset. It shows that GNN models are comparable to traditional ML-based models under the LRAP metric and significantly outperform traditional models under the NDCG metric. Specifically, the best-performing GNN model, GAT, outperforms the best-performing traditional method, RF, by more than 25%.

4.4 Overhead of the GNN inference

The overhead of the GNN inference consists of two parts: building the input graph from an input matrix and processing the data through our GNN model. Table 4 shows the detailed time costs for several representative matrices. As we can see, the processing time is negligible compared with the solving time. Although the cost of building the graphs is significantly higher than the processing time, it is still lower than the worst solving time of the linear system. Note that the same linear system is often solved many times in real applications that use PETSc. These observations imply that the cost of using the GNN models is justifiable and worthwhile relative to the solving time of the linear systems.

Table 3: Classification scores of considered methods on test dataset.

Method	LRAP	NDCG
RF	**0.7778 ± 0.0262**	**0.5618 ± 0.0299**
MLP (1 layer)	0.7231 ± 0.0378	0.5181 ± 0.0366
MLP (2 layers)	0.7570 ± 0.0530	0.5424 ± 0.0475
k-nearest neighbors	0.6465 ± 0.0405	0.4902 ± 0.0307
Ridge Classifier	0.3245 ± 0.0706	0.2473 ± 0.0239
GINE	0.7480 ± 0.0357	0.8126 ± 0.0285
GAT	**0.7770 ± 0.0267**	**0.8246 ± 0.0325**
GraphSAGE	0.7492 ± 0.0068	0.8043 ± 0.0388
GCN	0.7664 ± 0.0338	0.8222 ± 0.0216

Table 4: Building, processing and solving time of selected matrices. t_b, t_p, and t_s correspond to building time, processing time, and solving time of the linear system. Predicted t_s corresponds to the combination of solver and preconditioner recommended by our GNN models. Worst t_s corresponds to the worst combination, excluding the failed cases.

Matrix	size	nnz	t_b	t_p	predicted t_s	worst t_s
1138_bus	1138	4054	0.2084	0.0035	0.5195	5.7607
msc01440	1440	44998	2.6324	0.0035	0.5183	8.3124
cage9	3534	41594	2.4863	0.0035	0.2116	4.2138
cavity13	2356	72034	4.4684	0.0039	0.6144	>600
circuit_1	4875	105339	2.0890	0.0035	0.4243	>600

4.5 Case studies

We notice that both the LRAP and NDCG metrics focus on the ranked positions of "good" classes, which are the classes associated with ground-truth label 1. The difference is that, for LRAP, given a "good" class j, the rank of j in all "good" classes $|\mathcal{L}_{ij}|$ goes to the numerator and the rank of j in all classes $rank_{ij}$ becomes the denominator. And in the end, we will take the average score of all "good" classes. While for NDCG, these "good" classes are not considered separately. In particular, the numerator is the DCG of predicted positions of all "good" classes, and the denominator is the DCG of true positions of all "good" classes. Generally, when a "good" class is ranked low, both LRAP and NDCG can still be high if there are many "good" classes that are well-predicted. But NDCG focuses more on the leading classes. In other words, it highly depends on whether the first several predicted positions represent "good" classes. In contrast, LRAP penalizes harder than NDCG does when a "good" class is ranked low. Therefore, LRAP requires more "good" classes to be ranked high.

In our case, doing well on NDCG conveys better effectiveness because our users usually only need one or a few choices that can solve the linear system efficiently and accurately. And typically, they do not need our model to provide as many good suggestions as possible. Nevertheless, scoring high in both LRAP and NDCG can provide reliable robustness.

To interpret our experimental results and explain the large discrepancy between LRAP and NDCG scores, we performed a case study with the *Trefethen_2000* selected from the test dataset. Figure 2 shows the classwise prediction results for this matrix. There are three "good" classes (18, 19, and 20) as shown in green. GAT provides a better ranking (3, 1, and 6) for these "good" classes, while Random Forest (RF) yields 8, 1, and 7. As we can see in Fig. 2, the difference between LRAP is significantly higher than that on NDCG. The reason is that both methods correctly predict the class with the highest rank, so NDCG has a higher tolerance to other predictions than LRAP does.

Moreover, high NDCG scores in Table 3 suggest that our GNN models tend to provide safe recommendations and make fewer false positive predictions. Therefore, our models are less likely to rank "bad" classes high.

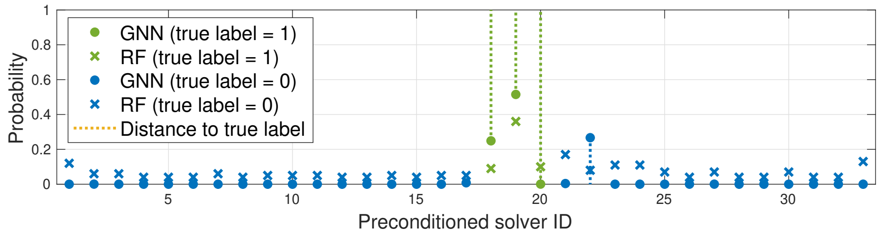


Figure 2: Prediction for matrix *Trefethen_2000*. Crosses denote predictions of RF. Here, GAT scores $LRAP = 0.7222, NDCG = 0.8711$, and RF scores $LRAP = 0.5536, NDCG = 0.7737$.

5 Conclusions

We have shown that graph neural networks have great potential in classification tasks where inputs are sparse matrices of varying sizes. By using the message-passing mechanism, defining node features properly, and using the output embedding, we can exploit the structural characteristics of the matrix such as the diagonal elements and sparsity patterns. Our methods also exploit graph features together with a two-level pooling, serving as a meaningful extension of standard GNNs.

Table 3 shows that the performances of 4 GNN models are indistinguishable. Future refinement includes exploring models of a different nature (e.g., transformers) that have the potential to outperform existing GNNs by a noticeable margin. In addition, graph coarsening techniques [48] will be deployed for arbitrary-sized (especially exceedingly large) matrices to improve the robustness and efficiency of the GNN models.

References

[1] TR Canfield, MSH Tang, and JE Foster. Nonlinear geometric and thermal effects in three dimensional Galerkin finite-element formulations for piezoelectric crystals. *Argonne National Laboratory, Argonne, IL*, 1991.

[2] Mark T Jones and Paul E Plassmann. Solution of large, sparse systems of linear equations in massively parallel applications. Technical report, Argonne National Lab., IL (United States), 1992.

[3] Esben Auken, Tue Boesen, and Anders V. Christiansen. Chapter two – a review of airborne electromagnetic methods with focus on geotechnical and hydrological applications from 2007 to 2017. volume 58 of *Advances in Geophysics*, pages 47–93. Elsevier, 2017.

[4] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. `http://petsc.org/`, 2022.

[6] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc/TAO Users Manual. Technical Report ANL-21/39 - Revision 3.17, Argonne National Laboratory, 2022.

[7] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[8] Luca Grementieri and Paolo Galeone. Towards neural sparse linear solvers. *arXiv preprint arXiv:2203.06944*, 2022.

[9] Richard Barrett, Michael Berry, Jack Dongarra, Victor Eijkhout, and Charles Romine. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. *Journal of Computational and Applied Mathematics*, 74(1-2):91–109, November 1996.

[10] Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. Application of machine learning to the selection of sparse linear solvers. *International Journal of High Performance Computing Applications*, 2006. Publisher: Citeseer.

[11] Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. Application of alternating decision trees in selecting sparse linear solvers. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, *Software Automatic Tuning*, pages 153–173. Springer New York, New York, NY, 2011.

[12] America Holloway and Tzu-Yi Chen. Neural networks for predicting the behavior of preconditioned iterative solvers. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2007*, volume 4487, pages 302–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.

[13] Erik Kuefler and Tzu-Yi Chen. On using reinforcement learning to solve sparse Linear systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2008*, volume 5101, pages 955–964. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[14] Sanjukta Bhowmick, Brice Toth, and Padma Raghavan. Towards low-cost, high-accuracy classifiers for linear solver selection. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2009*, volume 5544, pages 463–472. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[15] Paul R. Eller, Jing-Ru C. Cheng, and Robert S. Maier. Dynamic Linear Solver Selection for Transient Simulations Using Machine Learning on Distributed Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1915–1924, Shanghai, China, May 2012. IEEE.

[16] Lars Kotthoff, Ian P. Gent, and Ian Miguel. An evaluation of machine learning in algorithm selection for search problems. *AI Communications*, 25(3):257–270, 2012.

[17] Sanjiva Weerawarana, Elias N. Houstis, John R. Rice, Anupam Joshi, and Catherine E. Houstis. PYTHIA: a knowledge-based system to select scientific algorithms. *ACM Transactions on Mathematical Software*, 22(4):447–468, December 1996.

[18] Victor Eijkhout and Erika Fuentes. Machine learning for multi-stage selection of numerical methods. *New Advances in Machine Learning. INTECH*, pages 117–136, 2010.

[19] Victor Eijkhout and Erika Fuentes. A standard and software for numerical metadata. *ACM Transactions on Mathematical Software*, 35(4):1–20, February 2009.

[20] Lighthouse by LighthouseHPC. `http://lighthousehpc.github.io/lighthouse/`.

[21] Pate Motter, Kanika Sood, Elizabeth Jessup, and Boyana Norris. Lighthouse: an automated solver selection tool. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 16–24, Austin Texas, November 2015. ACM.

[22] Elizabeth Jessup, Pate Motter, Boyana Norris, and Kanika Sood. Performance-based numerical solver selection in the Lighthouse framework. *SIAM Journal on Scientific Computing*, 38(5):S750–S771, January 2016.

[23] The Trilinos Project Website. `https://trilinos.github.io`, May 2020.

[24] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[25] Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. *arXiv preprint arXiv:1811.03508*, 2018.

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

[27] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

[28] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[29] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019.

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

[31] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[32] Jinheon Baek, Minki Kang, and Sung Ju Hwang. Accurate learning of graph representations with graph multiset pooling. *arXiv preprint arXiv:2102.11533*, 2021.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] PyTorch. `https://www.pytorch.org`.

[35] PyTorch-Geometric. `https://www.pyg.org/`.

[36] PyTorch-Metrics. `https://torchmetrics.readthedocs.io/en/stable/`.

[37] the university of florida sparse matrix collection.

[38] Theta at argonne. `https://www.alcf.anl.gov/theta`.

[39] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. Mining multi-label data. *Data Mining and Knowledge Discovery Handbook*, pages 667–685, 2009.

[40] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[41] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. A theoretical analysis of NDCG ranking measures. In *Proceedings of the 26th annual conference on learning theory (COLT 2013)*, volume 8, page 6, 2013.

[42] Frank McSherry and Marc Najork. Computing information retrieval performance measures efficiently in the presence of tied scores. In *European conference on information retrieval*, pages 414–421. Springer, 2008.

[43] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[44] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[45] Geoffrey E Hinton. Connectionist learning procedures. In *Machine learning*, pages 555–610. Elsevier, 1990.

[46] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[47] Kuangqi Zhou, Yanfei Dong, Kaixin Wang, Wee Sun Lee, Bryan Hooi, Huan Xu, and Jiashi Feng. Understanding and Resolving Performance Degradation in Graph Convolutional Networks, 2021. arXiv:2006.07107 [cs, stat].

[48] Jie Chen, Yousef Saad, and Zechen Zhang. Graph coarsening: from scientific computing to machine learning. *SeMA Journal*, 79(1):187–223, 2022.