CallGraphNet: A GNN-based Approach for Call Graph Embedding Generation from Trace Data

Chuyi Zhao
Tufts University

1 Introduction

Over the past few decades, there has been a significant shift towards the utilization of microservices, gradually replacing traditional monolithic backend architectures. As distributed API services become more prevalent and software systems continue to scale, the interdependencies among these microservices become increasingly complex, generating large amounts of trace data. The massive scale of trace data presents challenges in efficiently designing the cluster architecture, and managing microservices during anomaly resolution or version updates. Developing tools to identify and cluster similar system behaviors could greatly assist in downstream analysis tasks, such as trace categorization, microservice behavior analysis, and automated diagnosis.

Graph-based learning techniques are powerful for calculating node-wise and graph-level embeddings as graph representations. Furthermore, since the dependent invocation relationships among microservices can be represented as standard directed call graphs, most Graph Neural Network (GNN) techniques can be applied in such scenarios. However, most trace data analyses focus either on specific downstream tasks, such as anomaly detection techniques[2, 9], which do not provide a general analysis, or rely on traditional mathematical methods or from hardware aspects, rather than GNN-based approaches. In this project, we propose CallGraphNet[1], a GNN-based framework for generally extracting general call graph embeddings from trace data, as a structural feature, and apply it over large-scale trace data from one of the largest cloud service providers. The effectiveness of the framework is demonstrated through comprehensive experiment analysis and comparison to baseline methods.

As designed, our framework takes a pair of call graphs as input and the *Graph Edit distance (GED)* [5] between them as a dissimilarity label, along with an integer k as a hyperparameter to determine the embedding vector size, and returns a k-dimension call graph embedding for both call graphs to fit the embedding distance to the label. The pipeline consists of the following steps:

1. Trace data is utilized to construct call graphs in the form of Direct Acyclic Graphs (DAGs), with each graph derived from an individual trace. Nodes represent microservices, and directed edges correspond to the invocation relationships between services.
2. We design node features and edge attributes for the GNN input based on service call information, such as the microservice id, call type, and communication paradigms among services.

[1] All source code is publicly available through this GitHub link.

3. Supervised Learning through bi-graph label prediction using the Earth Mover's Distance between embeddings to predict the call graph dissimilarity, here defined as the Graph Edit distance (GED). The core is to use GNN to learn graph-level embeddings for each call graph, as structural representations, for comparison of similar services. The embeddings capture the relationships and dependencies between services in the graph, allowing for the measurement of similarity or relevance between services. The GNN is selected from four standard ones including `GCNConv`, `SAGEConv`, `GATConv`, and `GATv2Conv`. More detailed methodologies are explained in Section 4, with Fig 1 for an overview.

By further analysis such as dimensional reduction over the output embedding, further analysis, visualization, and clustering of the clustered traces are facilitated.

In summary, we have achieved the following contributions to this work:

- We designed graph inputs based on large-scale trace data.
- We developed GallGraphNet, a GNN-based framework for supervised graph-level embedding extraction.
- We demonstrated the effectiveness of our framework through comparing it with the baseline method over a series of evaluation metrics.

2 Related Work

Current analytical works on trace data predominantly focus on integral and high-level reports, or they are specific to certain applications, such as architectural design at the hardware level or anomaly diagnosis within trace call graphs. However, my goal is to develop a practical method that allows users to find and explore some general graph-level features, that could help with various downstream tasks. To develop my recommendation framework, I mainly refer to the following works.

Analysis of trace data and microservice dependencies. Numerous analytical works focus on complex interactions between distributed services[6] within large-scale systems to improve system performance and maintainability by identifying potential faults and communication patterns.[11] These studies introduce various mathematical approaches to analyzing trace data, such as clustering, correlation analysis, and time-series analysis. Through these techniques, they accomplish the decomposition and visualization[10] of dependency structures within service clusters. And it is agreed that figuring out the characteristics of system behavior (contained in trace data here) would be beneficial for resolving to diagnose problems [8]. However, GNN-based methods offer new opportunities for more effective modeling and analysis of relationships, generating deeper insights into the structure and behavior of microservices.[7, 3]

Graph-based techniques to capture the graph structures. GNN-based techniques, use for recommendation, prediction, or regression, have experienced significant growth in recent years, largely due to their success in capturing complex relationships within large-scale data.[4] These techniques employ graph structures to represent entities and their interactions, effectively modeling complex systems. GNNs have been successfully applied to various domains, such as social networks, e-commerce platforms, and content-based recommendation systems, among others [12], learning from the graph attributes and structures and producing a lower dimensional representation accordingly. One example that captures the graph structure information for similarity comparison is *SimGNN*[1], as a GNN method to compute the Graph Edit Distance. However,

GNN application in microservice domains remains scarce. As mentioned above, trace data can be modeled as directed graphs, suggesting that GNN-based techniques could be effective in this area. Moreover, the "graph representation" representation as a GNN output, would be ideal for encoding and extracting the call graph structure information.

3 Background

Alibaba Cluster Trace Program. As mentioned previously, a growing number of cloud applications are being implemented as collections of microservices. However, academic research on industry-scale cloud applications is limited, as companies are reluctant to make such design information public. However, Alibaba did release some trace datasets along with a series of analysis papers[11, 6] in The Alibaba Cluster Trace Program.

The Alibaba Cluster Trace Program is a comprehensive dataset provided by Alibaba Cloud, one of the largest cloud service providers worldwide. This dataset offers a unique opportunity for researchers and developers to gain insights into the behavior and performance of large-scale, real-world cloud systems. The trace data contains detailed information about various aspects of Alibaba's cloud infrastructure, including jobs, tasks, containers, and machines, collected over a specific period.

In the context of our work, the Alibaba Cluster Trace Program dataset serves as the foundation for constructing call graphs that represent complex interdependencies among microservices. This invaluable resource allows us to apply our GNN-based framework to recommend similar microservices, ultimately contributing to more efficient management and utilization of cloud services.

4 Method

In this section, more details about the method theories will be talked over, including the input data forms, the problem definition, the workflow design, as well as some related concepts.

4.1 Input Data and Processing

The input data should be directed graphs, ideally in the form of `pytorch-geometric.data` types, containing the edge list and node attributes. In order to construct call graphs from the original messy trace dataset, we read the columns `rpcid`, `um`, and `dm` from the traces, addressing plenty of oddities in the dataset, enumerating all microservices and calls, and finally aggregate the calls as edge lists and the previous microservices identifiers as node attributes. This part is mainly finished by Chami Lamelas, as mentioned in Section 7. One important notation is that all call graphs generated through this pipeline are directed acyclic graphs (DAGs), and the 1-D feature of each node denotes the unique identifier of the microservice.

*Conceptionally, the **input data** for the framework could be denoted as a DAG $G = (X, E)$, where the 1-D node feature matrix $X \in \mathscr{R}^d (d = 1)$, and the edges $E \in \mathscr{R}^{|E| \times 2}$ are in the form of an edge list.*

4.2 Problem Definition

Since the primitive purpose is to explore and find some features of the trace data, and given call graphs constructed from the trace data, the problem could be simplified to "finding the repre-

sentation of a microservice call graph", or a graph-level embedding from the GNN perspective. Therefore, the problem could be initially stated as follows:

Graph Embedding Learning Problem. *Given a call graph as a DAG $G = (X,\ E)$, the GNN calculates a fixed-size vector $Z = GNN(X, E) \in \mathscr{R}^d$ as a output, where dimension d could be customized, so that Z could be considered as a structure representation for G, to compare call graph similarities and categorizing traces.*

Moreover, we adopt the *Graph Edit Distance (GED)*, *Earth Mover's Distance (EMD, or Wasserstein Distance)*, and *Mean Squared Error (MSE)* as a loss function, for the sake of the downstream task of calculating the bi-graph similarity (dissimilarity). The *GED*, defined as the minimum cost of a sequence of graph edit operations required to transform one graph into another, is a good measure of structural dissimilarity between two graphs, thus appropriate for ground truth labels, while *EMD*, likewise, measures the dissimilarity between two probability distributions, and here for the embedding vectors. Both distances are floating values ranging from 0 without an upper bound. Therefore, the graph embedding learning problem is simplified as a supervised learning problem, as the rough pipeline for this task could be defined as follows:

$$\hat{y} = EMD(Z_1,\ Z_2)$$
$$Loss = MSE(GED(G_1, G_2),\ \hat{y})$$

Supervised Graph Embedding-Based Bi-graph Dissimilarity Prediction. *Given a call graph in the form of a DAG as input, denoted as $G = (X,\ E)$, output the dissimilarity $\hat{y}$ between them based on the graph embeddings $Z \in \mathscr{R}^d$.*

The overview of the supervised learning workflow is shown in Fig 1. The method takes a pair of graphs as an input, takes the graph edit distance between them as a ground truth label, then uses GNN to calculate graph-level embeddings for both graphs. Finally, the Earth Mover's Distance between the two graph embeddings is calculated as a predicted dissimilarity score, in comparison to the ground truth label for the gradient descent using the *MSE* loss function.

In terms of the design of the embedding GNN model, I use a standard GNN, consisting of 5 message-passing layers then a mean pooling, with some nonlinear `ReLU` and `Dropout` operations in between for better performance, and a linear function, in the end, to tailor for the wanted output size. For the selection of message-passing layers, several standard layers are tried and compared, including `GCNConv`, `SAGEConv`, `GATConv`, and `GATv2Conv`, are trained and tested to find the optimal one. More experiment details and results are showcased in Section 5.

5 Experiments

5.1 Datasets

For the practical experiment part, a dataset of around 1,000 graph pairs in the exact form mentioned in 4.1, is processed for the *GED* label generation. However, due to the time limitation and slow *GED* calculations, nearly half takes over 10 minutes and has to be abandoned. Some attributes of the finished dataset are shown in Fig 1. Each graph pair is coordinated with a GED label, is generated for the input, and split in a proportion of `TRAINING SET: TESTING SET = 8:2`. Then the training set is further divided for validation, in a portion of `TRAINING SET: VALIDATION SET = 8:2`.

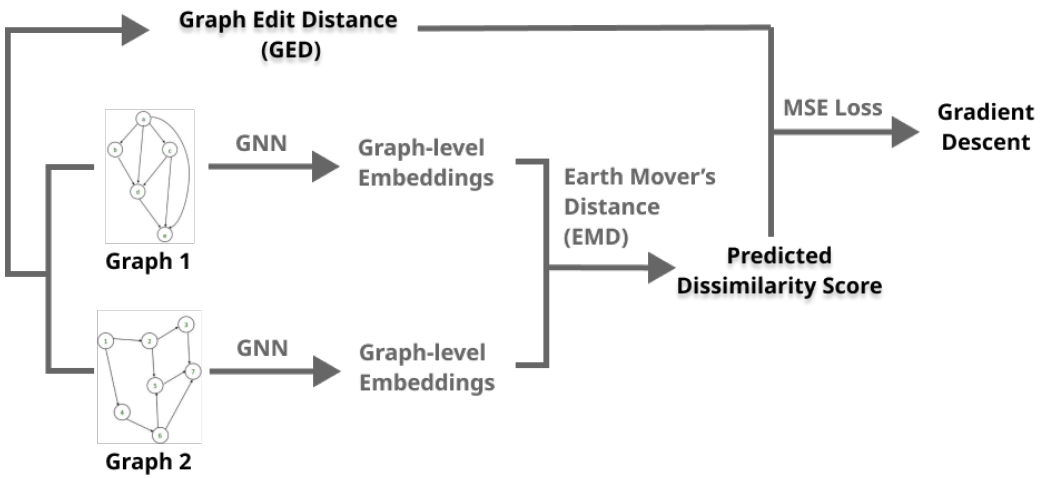

Figure 1: The overview of the supervised learning pipeline. The method takes a pair of graphs as an input, takes the graph edit distance as a ground truth label, then compares it to the output, the Earth Mover's Distance between the two graph embeddings, to calculate and descent the gradients.

	#nodes in G1	#nodes in G2	time for GED(s)	GED
mean	10.483083	10.746241	59.015902	18.648496
std	6.295433	8.362631	119.105771	16.375839
min	4.000000	4.000000	0.003789	1.000000
25%	7.000000	7.000000	0.330834	11.750000
50%	9.000000	9.000000	4.586815	15.000000
75%	12.000000	12.000000	52.917483	19.000000
max	77.000000	107.000000	598.286629	153.000000

Table 1: The overview of the experiment dataset with GED labels calculated.

5.2 Evaluation Metrics

As for the evaluation part, four metrics would be taken into consideration – the predicting *accuracy*, the amount of *information* captured, the model *transduction*, and *the time and space complexity*.

First off, we define the *accuracy* as the percentage of the prediction "error" within a `THRESHOLD_ERROR`, where the error $e = \frac{GED - prediction}{GED} \times 100\%$. In this way, we captured the portion of "good predictions" close to the *GED labels* and could compare among different GNN models to find and adopt the optimal one. As a presumption, the model with an *accuracy* over 0.8 and `THRESHOLD_ERROR`$\leq$ 0.3 should be considered as a good model.

The following three parameters are for the comparison of CallGraphNet versus the baseline *GED* method, arguing in which cases CallGraphNet might prevail over *GED*. All of the above would be talked over in depth in Section 5.3.

5.3 Results

After training and testing four models, the experiment results are shown in Table 2, where we use *MSE* as a loss function and the *accuracy* defined in Section 5.2 for evaluating the predicting model. Bellow I would analyze all four aspects mentioned previously.

From Table 2, although GATConv has the best performance, all four trained models do not appear to have a "good accuracy" as presumed. Given that over half of *GED* labels are small and no larger than 20, and that only a dataset of 1,000 is available, this is understandable. Hopefully, the model performance could be greatly improved as the larger dataset is available, which would be one future work.

	Validation Accuracy	Testing Accuracy
GCNConv	0.6153	0.4694
SAGEConv	0.5714	0.4898
GATConv	0.5476	0.5510
GATv2Conv	0.4762	0.4490

Table 2: The validation accuracy and testing accuracy of different GNN models.

In terms of the *information amount*, since our downstream task is to explore some call graph patterns and categorize them accordingly, some higher dimensional features or representations would be favored. While *GED* provides a good bi-graph-level value indicating the dissimilarity between the two call graphs, no graph-level information could be extracted from that single value, nor allow for visualization of behavioral patterns by a graph. However, CallGraphNet outputs embedding vectors as a representation for each call graph, which contains more information and could be clustered or analyzed further for visualization and categorization.

Transductive and *Complexity* are other advantages of CallGraphNet. As a counterpart, the "bi-graph dissimilarity problem" could be resolved by constructing a huge hashmap-like data structure and storing all precalculated *GED* labels between each possible pair. In that case, it is easy to fetch any wanted label in O(1). Nevertheless, such a method is not transductive, meaning it won't apply to any input containing a new-coming call graph not in the training set. Moreover, the time and space complexity are enormously larger than CallGraphNet, as shown in Table 3.

	GED	CallGraphNet
Time	NP Problem Takes 10min+ for $\leq 50\%$ pairs each Mean = 306.20s, std = 283.88	Simple 5-layer GNN Takes 10min- for 100% models for data loading, training, and testing all dataset in total
Space	Run for each possible pairs O(N^2) where N=#of call graphs	Store the pretrained `model.pt` O(1)

Table 3: The comparison of time and space complexity of GED and CallGraphNet methods.

In summary, CallGraphNev has several advantages over *GED* and works theoretically better for the desired downstream task, though the predicting performances are less than expected. However, due to the time limitation, the experiment could be greatly improved to better support this with better prediction accuracy. The next steps could be to process more input data and experiment with better-designed GNN models.

6 Conclusion

In conclusion, this essay introduces CallGraphNet, a GNN-based framework designed to extract call graph embeddings from trace data as a structural representation for comparing call graph dissimilarities and aiding downstream tasks, such as trace data analysis and clustering. As discussed, CallGraphNet offers several advantages over the baseline *GED* method, including its ability to produce informative results, transductive learning, and efficiency in terms of time and space. Although the prediction performance of CallGraphNet did not meet expectations, the framework demonstrates theoretical promise for the desired downstream tasks. Due to time constraints, further experimentation and improvements can be made to enhance the prediction accuracy. Future work may include processing additional input data and experimenting with more advanced GNN models to refine the framework's performance.

7 Acknowledgments

Gratitude to Chami Lamelas, Ph.D. candidate at Tufts University, for his assistance with trace data cleaning and call graph reconstruction.

References

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation, 2018.

[2] Ailin Deng and Bryan Hooi. Graph neural Network-Based anomaly detection in multivariate time series. June 2021.

[3] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1387–1397, New York, NY, USA, November 2020. Association for Computing Machinery.

[4] Heesoo Jung, Sangpil Kim, and Hogun Park. Dual policy learning for aggregation optimization in graph neural network-based recommender systems. February 2023.

[5] Julien Lerouge, Zeina Abu-Aisheh, Romain Raveaux, Pierre Héroux, and Sébastien Adam. Graph edit distance : a new binary linear programming formulation, 2015.

[6] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 412–426, New York, NY, USA, November 2021. Association for Computing Machinery.

[7] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, I-Hsiu Liu, and Ci-Wei Lan. Graph-based and scenario-driven microservice analysis, retrieval, and testing. *Future Gener. Comput. Syst.*, 100:724–735, November 2019.

[8] Raja Sambasivan, Alice Zheng, Eno Thereska, and Gregory Ganger. Categorizing and differencing system behaviours. 01 2007.

[9] Kuanzhi Shi, Jing Li, Yuecan Liu, Yuzhu Chang, and Xuyang Li. BSDG: Anomaly detection of microservice trace based on dual graph convolutional neural network. In *Service-Oriented Computing*, pages 171–185. Springer Nature Switzerland, 2022.

[10] Adam Tornhill. Microservice dependencies - visualization. `https://codescene.com/blog/visualize-microservice-dependencies-in-team-context/`. Accessed: 2023-3-29.

[11] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. Characterizing job microarchitectural profiles at scale: Dataset and analysis. In *Proceedings of the 51st International Conference on Parallel Processing*, number Article 47 in ICPP '22, pages 1–11, New York, NY, USA, January 2023. Association for Computing Machinery.

[12] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z Sheng, Mehmet A Orgun, Longbing Cao, Francesco Ricci, and Philip S Yu. Graph learning based recommender systems: A review. May 2021.