UCLA
UCLA Electronic Theses and Dissertations

Title
Neural Network-based Graph-Level Operator Learning

Permalink
https://escholarship.org/uc/item/02t1z1z3

Author
Bai, Yunsheng

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Neural Network-based Graph-Level Operator Learning

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Yunsheng Bai

2023

© Copyright by
Yunsheng Bai
2023

ABSTRACT OF THE DISSERTATION

Neural Network-based Graph-Level Operator Learning

by

Yunsheng Bai

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2023

Professor Yizhou Sun, Co-Chair

Professor Wei Wang, Co-Chair

Graph deep learning models have been popular in graph-based applications such as node classification, link prediction, community detection, etc. The expressive power of deep learning models combined with the increasing amount of graph data enables researchers to solve graph tasks that are traditionally done via algorithmic approaches. For example, neural network-based models have shown state-of-the-art performance on the graph classification task, outperforming kernel-based methods. However, how to perform graph matching related tasks such as Graph Edit Distance (GED) computation, Maximum Common Subgraph (MCS) detection, still requires careful design and utilization of graph deep learning models, due to the unique challenges posed by these NP-hard problems. In this dissertation, six recent researches on neural network-based graph-level operator learning are introduced. Extensive experiments show that the proposed models gain extraordinary improvements compared to the baseline approaches.

The dissertation of Yunsheng Bai is approved.

Quanquan Gu

Kai-Wei Chang

Yizhou Sun, Committee Co-Chair

Wei Wang, Committee Co-Chair

University of California, Los Angeles

2023

To my family, and myself.

A journey to end and to start.

TABLE OF CONTENTS

List of Figures xi

List of Tables xvii

Acknowledgments xx

Vita xxii

1 Introduction **1**

1.1 Graphs 1

1.2 Neural Network Operators for Graph-Level Applications 1

1.3 Research Roadmap and Thesis Contribution 3

1.4 Thesis Overview 4

2 SimGNN: A Neural Network Approach to Fast Graph Similarity Computation **7**

2.1 Introduction 7

2.2 Preliminaries 10

2.2.1 Graph Edit Distance (GED) 10

2.2.2 Graph Convolutional Networks (GCN) 11

2.3 The Proposed Approach: SimGNN 12

2.3.1 Strategy One: Graph-Level Embedding Interaction 13

2.3.2 Strategy Two: Pairwise Node Comparison 16

2.3.3 Time Complexity Analysis 17

2.4 Experiments 18

2.4.1 Datasets 18
2.4.2 Data Preprocessing 19
2.4.3 Baseline Methods 20
2.4.4 Parameter Settings 21
2.4.5 Evaluation Metrics 21
2.4.6 Results 22
2.4.7 Parameter Sensitivity 25
2.4.8 Case Studies 25
2.5 Related Work 27
2.5.1 Network/Graph Embedding 27
2.5.2 Graph Similarity Computation 28
2.6 Conclusion 28

3 Learning-based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching 29
3.1 Introduction 29
3.2 Related Work 31
3.3 Problem Definition 32
3.4 The Proposed Approach: GraphSim 33
3.4.1 Multi-Scale Neighbor Aggregation 35
3.4.2 Similarity Matrix Generation 36
3.4.3 CNN Based Similarity Score Computation 39
3.5 Experiments 40
3.5.1 Effectiveness 40
3.5.2 Efficiency 41

3.5.3 Analysis of Various Proposed Techniques in GraphSim 42
3.5.4 Analysis of Similarity "Images" in GraphSim 42
3.5.5 Case Studies 44
3.6 Conclusion 44

4 Unsupervised Inductive Graph-Level Representation Learning via Graph-Graph Proximity 46
4.1 Introduction 46
4.2 The Proposed Approach: UGraphEmb 48
4.2.1 Inductive Graph-Level Embedding 48
4.2.2 Unsupervised Loss via Inter-Graph Proximity Preservation 52
4.3 Experiments 54
4.3.1 Task 1: Graph Classification 54
4.3.2 Task 2: Similarity Ranking 55
4.3.3 Task 3: Embedding Visualization 57
4.3.4 Parameter Sensitivity of UGraphEmb 57
4.4 Related Work 58
4.5 Conclusion 59

5 Learning to Search for Fast Maximum Common Subgraph Detection . . 60
5.1 Introduction 60
5.2 Preliminaries 62
5.2.1 Problem Definition 62
5.2.2 Search Algorithm for MCS 63
5.3 Proposed Method 64
5.3.1 Leveraging DQN for Search 64

5.3.2 Representation Learning for DQN 65
5.3.3 Leveraging Search for DQN Training 68
5.4 Experiments 68
5.4.1 Baseline Methods 69
5.4.2 Parameter Settings 69
5.4.3 Results 70
5.4.4 Ablation and Parameter Study 72
5.4.5 Overhead of GLSearch 73
5.5 Conclusion 74

6 Unifying Geometric Regularization and Policy Gradients for Subgraph Matching 75
6.1 Introduction 75
6.2 Related Work 77
6.3 Preliminaries 79
6.3.1 Problem Definition 79
6.3.2 Search-based methods for subgraph matching 80
6.4 Proposed Method 82
6.4.1 RL Formulation of subgraph matching 82
6.4.2 Encoder-Decoder Design for Policy Estimation 84
6.4.3 Loss Function (CGR-Loss) 86
6.5 Evaluation Protocol 89
6.5.1 Dataset 89
6.5.2 Baselines 89
6.5.3 Experimental Setup 90

6.6 Experimental Results . . . 91
6.6.1 Outperforming State-of-the-Art Solvers . . . 91
6.6.2 Importance of Search-based Framework . . . 93
6.6.3 Outperforming GLSearch and NMatch . . . 93
6.6.4 Effect of Global Policy Network . . . 95
6.6.5 Effect of CGR-Loss . . . 95
6.6.6 Ablation Studies . . . 96
6.6.7 Performance Across Iterations . . . 96
6.7 Comparison with Related Works . . . 97
6.7.1 Comparison to GLSearch . . . 97
6.7.2 Comparison with Dmpnn . . . 99
6.7.3 Comparison with RL-QVO . . . 99
6.8 Conclusion . . . 99

7 ProgSG: Cross-Modality Representation Learning for Programs in Electronic Design Automation . . . 101
7.1 Introduction . . . 101
7.2 Preliminaries . . . 104
7.2.1 Background Introduction . . . 104
7.2.2 Problem Definitions . . . 105
7.2.3 Related Work . . . 106
7.3 Proposed Method: ProgSG . . . 107
7.3.1 Overall Architecture for Design Quality Prediction . . . 108
7.3.2 ProgSG-si: Graph-Summary-Augmented Sequence Representation . 108
7.3.3 Full Model ProgSG: Leveraging Fine-grained Alignment . . . 110

7.3.4 Pretraining GNNs for CDFGs . . . 111

7.4 Experiments . . . 112

7.4.1 Dataset and Evaluation Protocol . . . 113

7.4.2 Model Setup and Hyperparameters . . . 114

7.4.3 Experimental Results . . . 114

7.4.4 Attention Visualization . . . 115

7.5 Conclusion . . . 116

8 Conclusion . . . 117

LIST OF FIGURES

2.1 Illustration of similarity-preserving graph embedding. Each graph is mapped into an embedding vector (denoted as a dot in the plot), which preserves their similarity between each other in terms of a specific graph similarity metric. The green "+" sign denotes the embedding of an example query graph. Colors of dots indicate how similar a graph is to the query based on the ground truth (from red to blue, meaning from the most similar to the least similar). 8

2.2 The GED between the graph to the left and the graph to the right is 3, as the transformation needs 3 edit operations: (1) an edge deletion, (2) an edge insertion, and (3) a node relabeling. 11

2.3 An overview illustration of SimGNN. The blue arrows denote the data flow for Strategy 1, which is based on graph-level embeddings. The red arrows denote the data flow for Strategy 2, which is based on pairwise node comparison. 12

2.4 Some statistics of the datasets. 19

2.5 Visualizations of node attentions. The darker the color, the larger the attention weight. 24

2.6 Runtime comparison. 25

2.7 A query case study on AIDS. Meanings of the colors can be found in Fig. 2.4a. 25

2.8 A query case study on LINUX. 26

2.9 A query case study on IMDB. 26

2.10 Mean squared error w.r.t. the number of dimensions of graph-level embeddings, and the number of histogram bins. 26

3.1 The GED is 3, as the transformation needs 3 edit operations: Two edge deletions, and an edge insertion. The MCS size is 4. 34

3.2 Left: An overview illustration of our proposed method GraphSim. No graph-level representation is generated, and it directly uses the node-node similarity scores in the three similarity matrices corresponding to node embeddings at different scales. Right: Illustration of three similarity matrices from the Linux [1] dataset. For two isomorphic graphs (A and A), a strong symmetric diagonal block pattern is observed; for two less similar graphs (A and B), the dark diagonal pattern is less evident; for two graphs that are not similar at all (A and C), the symmetric block pattern is almost gone. For graphs of different sizes, we devise a consistent max padding scheme. Intuitively, the block patterns can be thought of as graph-graph similarity patterns at different scales, which are suitable for CNNs to capture. . 34

3.3 Running time comparison. The y-axis uses the log scale. The running time is averaged across queries. For neural network models, the running time for GED and MCS computation is very close to each other, so we take the average of the two. 42

3.4 Similarity "images" on two pairs of graphs from Aids trained with the GED metric. Different heatmap colors differentiate "images" from different scales of comparison. Top pair: GED=2—an edge deletion (edge between node 0 and 4 of graph (b)) and an edge addition (edge between node 3 and 4 of graph (b)). Bottom pair: GED=1—a node relabeling (node 7). 43

3.5 Visualization of ranking results under the GED metric. From top to bottom: Aids, Linux, Imdb, Ptc. 45

4.1 Overview of UGraphEmb. (a) Given a set of graphs, (b) UGraphEmb first computes the graph-graph proximity scores (normalized distance scores in this example), (c) yielding a "hyper-level graph" where each node is a graph in the dataset, and each edge has a proximity score associated with it, representing its weight/strength. UGraphEmb then trains a function that maps each graph into an embedding which preserves the proximity score. The bottom flow illustrates the details of graph-level embedding generation. (d) After embeddings are generated, similarity ranking can be performed. The green "+" sign denotes the embedding of an example query graph. Colors of dots indicate how similar a graph is to the query based on the ground truth (from red to blue, meaning from the most similar to the least similar). (e) Finally, UGraphEmb can perform fine-tuning on the proximity-preserving graph-level embeddings, adjusting them for the task of graph classification. Different colors represent different graph labels in the classification task. 49

4.2 Visualization of the IMDB dataset. From (a) to (g), for each method, 12 graphs are plotted. For (h) to (l), we focus on UGraphEmb: 5 clusters are highlighted in red circles. 12 graphs are sampled from each cluster and plotted to the right. 58

4.3 Classification accuracy on the IMDB dataset w.r.t. the dimension of graph-level embeddings and the percentage of graph pairs used for training. 59

5.1 Left: For graph pair $(\mathcal{G}_1, \mathcal{G}_2)$ with node labels, the induced connected MCS is the five-member ring structure highlighted in circle. Right: At this step, there are two nodes currently selected. According to whether each node is connected to the two selected nodes or not, the nodes not in the current solution are split into three bidomains (Section 5.2.2), denoted as "00", "01", and "10", where "0" indicates not connected to a node in the selected two nodes, and "1" indicates connected. For example, each node in the "10" bidomain is connected to the top "C" node in the subgraph and disconnected to the bottom "C" node in the subgraph. 62

5.2 An illustration of the search process for MCS detection. For $(\mathcal{G}_1, \mathcal{G}_2)$, the branch and bound search algorithm (Section 5.2.2 and Algorithm 5.1) yields a tree structure where each node represents one state (s_t) with id reflecting the order in which states are visited, and each edge represents an action (a_t) of selecting one more node pair. The search is essentially depth-first with pruning by the upper bound check. Our model learns the node pair selection strategy, i.e. which state to visit first. If state 6 can be visited before state 1, a large solution can be found in less iterations. When the search completes or a pre-defined search iteration budget is used up, the best solution (output subgraphs) will be returned, corresponding to state 13 (and 14). 66

5.3 Visualization of MCS results on Road. Nodes with large degrees have large circles. For each method, we show the two graphs being matched. Selected subgraphs are colored in green. 72

5.4 Comparison of the best solution sizes of different methods on Road. 74

6.1 The overall process of subgraph matching is a search algorithm that matches one node pair at a time for the input query q and target graph G guided by a learned policy. Due to the large action space incurred by the large G in practice, we propose to train a policy network to guide the selection of local candidate nodes in G at each search state. A global candidate set is computed via a filtering algorithm (line 3 of Algorithm 6.1) before the search starts. In the figure, we illustrate the global candidate set for a particular node in q, which has 6 candidate nodes in G. At each search state, a local candidate set is computed to further reduce the candidate actions (line 8 of Algorithm 6.2). In the figure, the node in q has 4 local candidate nodes in G. From this figure, we can see that global and local candidate states directly reduce the action space by pruning node-node pairs that cannot be matched. The predicted policy guides the search by visiting one node in the local candidate set first, and when backtracked, the search will select another from the candidates, resulting in two branches below s_1. The goal of the policy is to guide the search so that a more promising node is visited first, leading to the early discovery of solutions to this NP-hard problem under a limited time budget. 83

6.2 The growth in the number of input graph pairs where we find at least one solution. Notice, NSubS is able to outperform state-of-the-art solver baselines. Section 6.6.1 discusses this further. NSubS is also able to beat NMatch and GLSearch, models adapted from the representation learning and "learning to search" communities. Section 6.6.3 discusses this further. 88

6.3 The growth in the number of input graph pairs where we find at least one solution with respect to iterations. More details can be found in Section 6.6.7. 94

7.1 The overall diagrams of ProgSG-ca and ProgSG. "GNN", "TF", and "Dec" refer to Graph Neural Network Layer, Transformer Layer, and Decoder, respectively. For brevity, only some of the aligned token-node pairs derived from the assembly code are illustrated in dashed green arrows for ProgSG. 110

7.2 Bar plots of average attention scores of pragma-related tokens before (ProgSG-rand) and after (ProgSG) being fine-tuned. 116

LIST OF TABLES

2.1 Statistics of datasets. 17

2.2 Results on AIDS. 23

2.3 Results on LINUX. 23

2.4 Results on IMDB. Beam, Hungarian, and VJ together are used to determine the ground-truth results. 23

3.1 Effectiveness results on the GED metric. On AIDS and LINUX, A* provides ground-truth results, labeled with superscript $*$. On IMDB and PTC, A* fails to compute most GEDs within a 5-minute limit (thus denoted as $-$). Instead, the minimum GED returned by BEAM, HUNGARIAN, and VJ for each pair is used as the ground-truth GED. The mse is in 10^{-3}. 39

3.2 Effectiveness results on the MCS metric. On all the datasets, MCSPLIT provides ground-truth results, labeled with superscript $*$. The mse is in 10^{-3}. 41

3.3 GS-PAD and GS-RESIZE perform node ordering and use the node embeddings only by the last GCN layer to generate the similarity matrix. Since CNNs require fixed-length input, if the model does not use resizing, a simple way is to zero pad each similarity matrix to $N_{\max}$ by $N_{\max}$ (maximum graph size in the entire dataset), denoted as GS-PAD. GS-RESIZE uses the proposed max padding and resizing techniques. GS-NOORD uses all the proposed techniques including multi-scale comparison except for node ordering. The results are on the GED metric. The mse is in 10^{-3}. 43

4.1 Graph classification accuracy in percent. "-" indicates that the computation did not finish after 72 hours. We highlight the top 2 accuracy in bold. 55

4.2 Similarity ranking performance. Beam, Hungarian, and VJ are three approximate GED computation algorithms returning upper bounds of exact GEDs. We take the minimum GED computed by the three as ground-truth GEDs for training and evaluating all the methods on both Task 1 and 2. Their results are labeled with "$*$". HED is another GED solver yielding lower bounds. "-" indicates that the computation did not finish after 72 hours. 56

5.1 Results on small and medium graphs. Each synthetic dataset consists of 50 randomly generated pairs labeled as "⟨generation algorithm⟩-⟨number of nodes in each graph⟩". "BA", "ER", and "WS" refer to the Barabási-Albert (BA) [2], the Erdős-Rényi (ER) [3], and the Watts–Strogatz (WS) [4] algorithms, respectively. Nci109 consists of 100 chemical compound graph pairs whose average graph size is 28.73. We show the ratio of the (average) size of the subgraphs found by each method with respect to the best result on that dataset. 70

5.2 Results on real-world large graph pairs. Each dataset consists of one large real graph pair ($\mathcal{G}_1$, $\mathcal{G}_2$ may not be isomorphic, but $\mathcal{G}_{1s}$, $\mathcal{G}_{2s}$ are isomorphic guaranteed by search). Below each dataset name, we show its size $\min(|\mathcal{V}_1|, |\mathcal{V}_2|)$ to indicate these pairs are significantly larger than the ones in Table 5.1. Consistent with Table 5.1, we show the ratio of the subgraph sizes. 71

5.3 Abaltion study on real datasets. 72

6.1 Target graph description. Details are shown in Section 6.5.1. 89

6.2 Average number of subgraph matchings found after 5 minutes on one graph pair in each dataset. Note, multiple subgraph matchings can exist for a single graph pair. For clarity and compactness, each result has been divided by 1000, i.e. each number is in the unit of 10^3. More details can be found in Section 6.6.1 and 6.6.3. 90

6.3 Number of valid sugbraph matchings returned by IsoNet and search-based approaches on the BA dataset. More details can be found in Section 6.6.2. 91

6.4 NSubS Ablation Study on encoder and loss function design. More details can be found in Section 6.6.4, 6.6.5, and 6.6.6. 92

7.1 Target pragmas with their options. 105

7.2 Dataset statistics. "**#D**", "**#P**", "**A#P**", "**A#T**", "**A#N**", "**A#E**", and "**A#MP**" denote "# designs", "# programs", "avg # pragmas per program", "avg # tokens per program", "avg # nodes per program's CDFG", "avg # edges per program's CDFG", and "avg # matched node-token pairs", respectively. . 114

7.3 Prediction RMSE on SDx 2018.3 (v1) and Vitis 2020.2 (v2). 115

7.4 Result breakdown on SDx 2018.3 (v1) on individual test kernels. 115

7.5 Result breakdown on Vitis 2020.2 (v2) on individual test programs. 115

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my PhD advisors, Prof Yizhou Sun and Prof Wei Wang. They not only teach me how to conduct quality research, turning me from someone knowing little about graphs into a researcher but also demonstrate what exceptional human beings should look like. The characteristics of being such a person include but not limited to, showing kindness and respect to others, providing help to others when they need it, being persistent and tenacious even faced with tremendous challenges, etc. In that sense, I believe my PhD journey is not only about academic achievement but also about personal growth and enlightenment.

In addition to my advisors, I would like to thank my committee members. Prof. Kai-Wei Chang and Prof. Quanquan Gu for their discussions, feedback and suggestions. Their expertise and support always inspire me in my research throughout my PhD study. I also want to thank all my great instructors during my course study at UCLA from CS, ECE and Stat departments, especially Prof. Junghoo (John) Cho and Prof. Lieven Vandenberghe.

I am honored to work with faculty members such as Prof. Jason Cong, who provide me with valuable experiences and project management skills in interdisciplinary research from broader domains. As a member of the ScAI lab at UCLA, I am grateful to work with many talented colleagues including Ziniu Hu, Zongyue Qin, Derek Xu, Roshni Iyer, Kewei Cheng, Song Jiang, Yewen Wang, Zhiping Xiao, Shichang Zhang, Zijie Huang, Junheng Hao, Chelsea Ju, Jyun-Yu Jiang and Xiusi Chen. I enjoyed working with them and learned a lot from weekly course study and reading groups.

I am fortunate to have had three internships during my PhD study. I would like to express my greatest appreciation to all of my mentors and managers: Dr. Yada Zhu (IBM Research), Dr. Srinivasan Sengamedu (Amazon), and Mr. Wei Shao (Meta). They expanded my research scope into the industrial environment and encouraged me to explore interesting projects. During my internships (two in-person and two virtual internships), I feel lucky to meet many distinguished researchers, outstanding engineers and other interns (also all as

friends) and enjoy every minute of discussing with them and learning from their talks and presentations.

Last and most importantly, I want to say a huge thank you to my loving family. There is not enough word to express how much I owe to my father, Chuan Bai, and my mother, Hong Zhao, for their unconditional and endless love which gives me the strongest support through the darkest moments, crossing thousands of miles over the Pacific.

A journey is about to end and another awaits. Always be grateful for what life gives, and always be ready for whatever it takes.

VITA

2014 – 2017 B. Eng. in Computer Engineering, University of Michigan, Ann Arbor
Ann Arbor, Michigan

2018 – 2020 Teaching Assistant/Associate, Computer Science Department, UCLA
Los Angeles, CA, United States

2020 Research Intern, IBM Research
Remote, United States

2021 Applied Scientist Intern, Amazon.com Services
Remote, United States

2022 Research Scientist Intern, Meta
Menlo Park, CA, United States

2017 – 2023 Graduate Student Researcher, Computer Science Department, UCLA
Los Angeles, CA, United States

PUBLICATIONS

Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. **WSDM** (2019).

Yunsheng Bai*, Hao Ding*, Yizhou Sun, and Wei Wang. 2020. Learning-based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching. **AAAI** (2020).

Yunsheng Bai, Hao Ding, Yang Qiao, Agustin Marinovic, Ken Gu, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Unsupervised Inductive Graph-Level Representation Learning

via Graph-Graph Proximity. **IJCAI** (2019).

Roshni Iyer, **Yunsheng Bai**, Yizhou Sun, and Wei Wang. 2022. Dual-Geometric Space Embedding Model for Two-View Knowledge Graphs. **KDD** (2022).

Yunsheng Bai*, Derek Xu*, Yizhou Sun, and Wei Wang. 2021. GLSearch: Maximum Common Subgraph Detection via Learning to Search. **ICML** (2021).

Yunsheng Bai*, Derek Xu*, Yizhou Sun, and Wei Wang. 2022. Subgraph Matching via Detecting Small Query Graphs in A Large Graph via Neural Subgraph Search. **KDD** (2023). Under Review.

Zongyue Qin, **Yunsheng Bai**, and Yizhou Sun. 2019. Graph Hashing via Graph Neural Network for Similarity Search. **KDD** (2020).

Atefeh Sohrabizadeh, **Yunsheng Bai**, Yizhou Sun, and Jason Cong. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. **DAC** (2022).

Yunsheng Bai, Atefeh Sohrabizadeh, Yizhou Sun, and Jason Cong. 2022. Improving GNN-Based Accelerator Design Automation with Meta Learning. **DAC** (2022).

Yiqiao Jin, **Yunsheng Bai**, Yanqiao Zhu, Yizhou Sun, Wei Wang. 2022. Code Recommendation for Open Source Software Developers. **WEB** (2023).

Yunsheng Bai*, Ken Gu*, Yizhou Sun, and Wei Wang. 2020. Bi-Level Graph Neural Networks for Drug-Drug Interaction Prediction. **ICML 2020 Graph Representation Learning and Beyond (GRL+) Workshop** (2020).

J. Harry Caufield, Yichao Zhou, **Yunsheng Bai**, David A. Liem, Anders O. Garlid, Kai-Wei Chang, Yizhou Sun, Peipei Ping, Wei Wang. 2019. A comprehensive entity and relationship typing system for information extraction from clinical narrative text. (2019).

Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, Jason Cong. 2023. Towards a Comprehensive Benchmark for FPGA Targeted High-Level Synthesis. **NeurIPS** (2023). Under Review.

CHAPTER 1

Introduction

1.1 Graphs

Graph in the most basic form can be defined as a set of nodes connected with edges. Due to the generality of the definition, the graph has been a very popular way to represent a broad range of data in the real world. To give a few examples, chemical compounds can be naturally modeled as graphs, where the nodes represent atoms and the edges, represent the chemical bonds. A computer program can also be viewed as a graph, e.g. an abstract syntax tree, a control data flow graph, etc. Larger examples include knowledge graphs and academic collaboration networks, which can be up to billion of nodes [5]. The accumulation of such graph data across different application domains call for the need to design various neural network-based operators applied to these graphs.

1.2 Neural Network Operators for Graph-Level Applications

Deep learning models for graph data have gained a significant amount of attention and success in recent years. On one hand, more and more novel models have been proposed to solve tasks such as node classification, link prediction, graph classification, network community detection etc. Among various models, Graph Concolutional Network (GCN) [6, 7] is a seminal work that achieves much better performance than earlier methods. Since its success, a huge amount of graph neural network models have been proposed, such as GraphSAGE [8], graph Attention Network (GAT) [9], Graph Isomorphism Network (GIN) [10], etc. On the other hand, a large amount of graph data has been accumulated [11, 12] that

greatly facilitates research on graph deep learning.

The huge expressive power of deep learning models together with the growing amount of graph datasets enables researchers to explore and re-explore a lot of traditional graph-related tasks. For example, graph classification has been traditionally solved by graph kernel approaches [13], but recently more and more graph pooling-based neural network models are proposed to solve it [14, 15, 16]. However, when it comes to graph matching, few researchers aim at solving it via learning-based approaches. This is partly due to the fact that graph matching tasks are typically NP-hard, such as Graph Edit Distance (GED) [17] computation, Maximum Common Subgraph (MCS) [18] detection. The NP-hard nature of these tasks prevents the direct adoption of deep learning models that are originally designed for simpler tasks such as node and graph classification. How to design graph-level operators for graph similarity and matching tasks remains a challenge.

Besides various applications for graph data, another more fundamental problem is the exploration of better ways to represent graphs in a vector space, i.e. graph representation learning. Existing approaches typically utilize the link structure of a graph to learn its embedding/representation [14], but few methods attempt to leverage the inter-graph information, e.g. graph-graph proximity defined by GED or MCS, to enable the deep learning models to learn more expressive representations. This is especially an issue when it comes to learn the whole-graph representation, i.e. one embedding per graph, for tasks that require the positions of different graphs to be captured, such as graph matching tasks. Besides this advantage, the incorporation of graph-graph proximity information also enables the neural network models to be unsupervised, which is especially helpful when labeled graph data is scarce.

Recent years have witnessed the growing popularity of domain-specific accelerators (DSAs) for accelerating various applications such as deep learning, search, autonomous driving, etc. To facilitate DSA designs, high-level synthesis (HLS) is used, which allows a developer to compile a high-level description in the form of software code in C and C++ into a design in low-level hardware description languages (such as VHDL or Verilog) and eventually

synthesized into a DSA on an ASIC (application-specific integrated circuit) or FPGA (field-programmable gate arrays). However, existing HLS tools still require a good amount of microarchitecture decisions, expressed in terms of pragmas (such as directives for parallelization and pipelining). To enable more people to design DSAs efficiently, it is desirable to automate such decisions with the help of deep learning for predicting the quality of HLS designs. This requires us a deeper understanding of the program, which is a combination of original code and pragmas. Naturally, these programs can be considered as sequence data, for which large language models (LLM) can help. In addition, these programs can be compiled and converted into a control data flow graph (CDFG), and the compiler also provides fine-grained alignment between the code tokens and the CDFG nodes. However, existing works either fail to leverage both modalities or combine the two in shallow or coarse ways. What is worse, they fail to leverage the vast amount of unlabeled source code data.

1.3 Research Roadmap and Thesis Contribution

In this dissertation, six research works are introduced. The first one is SimGNN [19], a novel neural network model designed for GED computation. It leverages the learning capacity of node and graph embedding methods to learn to predict the GED score for any graph pair. The second work, GraphSim [20], further improves the accuracy upon SimGNN. The third work, UGraphEmb [21], leverages graph-graph similarity for unsupervised representation learning upon a whole graph. The fourth work, GLSearch [22], is among the first work to explore neural network-based unsupervised graph-level representation by using graph-graph proximity. The fifth work, NSubS, solves the task of Subgraph Matching via a novel geometrically regularized loss function. The sixth work, ProgSA, aims to understand the semantics of programs represented as CDFG and source code text for predicting the quality of FPGA designs.

These six works are about neural network-based graph-level operator learning and focus on challenging tasks that operate on the entire graph level. Graph deep learning models are powerful tools, which can easily boost the performance of relatively simpler tasks such as

node and graph classification. However, care must be taken when harder tasks such as GED computation and MCS detection need to be handled. Thus, both the design choices and the reasons behind the choices will be emphasized. Significant improvements from extensive experiments are demonstrated. A detailed walkthrough of each project in the chapters of this dissertation is in Section 1.4.

1.4 Thesis Overview

The rest of this dissertation can be organized into the following parts.

Chapter 2: We introduce SimGNN, as in "*SimGNN: A Neural Network Approach to Fast Graph Similarity Computation*". SimGNN combines two strategies. First, we design a learnable embedding function that maps every graph into an embedding vector, which provides a global summary of a graph. A novel attention mechanism is proposed to emphasize the important nodes with respect to a specific similarity metric. Second, we design a pairwise node comparison method to supplement the graph-level embeddings with fine-grained node-level information. Our model achieves better generalization on unseen graphs, and in the worst case runs in quadratic time with respect to the number of nodes in two graphs. Taking GED computation as an example, experimental results on three real graph datasets demonstrate the effectiveness and efficiency of our approach. Specifically, our model achieves a smaller error rate and great time reduction compared to a series of baselines, including several approximation algorithms on GED computation, and many existing graph neural network-based models. Our study suggests SimGNN provides a new direction for future research on graph similarity computation and graph similarity search.

Chapter 3: In this chapter, we again address the problem of graph similarity computation but from another perspective, by directly matching two sets of node embeddings without the need to use fixed-dimensional vectors to represent whole graphs for their similarity computation. The model, GraphSim, achieves state-of-the-art performance on four real-world graph datasets under six out of eight settings (here we count a specific dataset and metric

combination as one setting), compared to existing popular methods for approximate Graph Edit Distance (GED) and Maximum Common Subgraph (MCS) computation.

Chapter 4: In this chapter, we introduce a novel approach to graph-level representation learning, which is to embed an entire graph into a vector space where the embeddings of two graphs preserve their graph-graph proximity. Our approach, UGraphEmb, is a general framework that provides a novel means to performing graph-level embedding in a completely unsupervised and inductive manner. The learned neural network can be considered as a function that receives any graph as input, either seen or unseen in the training set and transforms it into an embedding. A novel graph-level embedding generation mechanism called Multi-Scale Node Attention (MSNA), is proposed. Experiments on five real graph datasets show that ProgSG achieves competitive accuracy in the tasks of graph classification, similarity ranking, and graph visualization.

Chapter 5: As a much more challenging task that requires fine-grained node-node matching compared with graph similarity computation, Maximum Common Subgraph (MCS) detection is tackled in the next work, GLSearch, as in "*Glsearch: Maximum common subgraph detection via learning to search*". It is a Graph Neural Network (GNN) based on *learning to search* model. Our model is built upon the branch and bound algorithm, which selects one pair of nodes from the two input graphs to expand at a time. Instead of using heuristics, we propose a novel GNN-based Deep Q-Network (DQN) to select the node pair, making the search process faster and more adaptive. To further enhance the training of DQN, we leverage the search process to provide supervision in a pre-training stage and guide our agent during an imitation learning stage. Experiments on synthetic and real-world graph pairs demonstrate that our model learns a search strategy that is able to detect significantly larger common subgraphs than existing MCS solvers given the same computation budget. GLSearch can be potentially extended to solve many other combinatorial problems with constraints on graphs.

Chapter 6: In this chapter, we tackle the question of how one can efficiently and accurately detect the occurrences of a small query graph in a large target graph, which is a core oper-

ation in graph database search, biomedical analysis, social group finding, etc. This task is called subgraph matching which essentially performs subgraph isomorphism check between a query graph and a large target graph. One promising approach to this classical problem is the "learning-to-search" paradigm, where a reinforcement learning (RL) agent is designed with a learned policy to guide a search algorithm to quickly find the solution without any solved instances for supervision. However, naively applying "learning-to-search" ignores a large body of recent work from both the representation learning and combinatorial search communities, who have drastically improved ways of encoding node representations and accelerating the search framework for subgraph matching. In this chapter, we propose NSubS, which unifies "learning-to-search" with recent developments from the aforementioned communities through two major innovations: (1) A simple global policy network tuned for highly efficient search frameworks; (2) A contrastive geometrically-regularized policy gradient for improved RL training. Experiments on four large real-world target graphs show that NSubS can significantly improve the subgraph matching performance.

Chapter 7: In this chapter, we turn to the task of electronic design representation learning, and propose ProgSA allowing the source code sequence modality and the graph modalities to interact with each other in a deep and fine-grained way. To alleviate the scarcity of labeled designs, a pre-training method for HLS design representation learning is proposed based on a suite of compiler's data flow analysis tasks. Experimental results on two benchmark datasets show the superiority of ProgSA over baseline methods that either only consider one modality or combine the two without utilizing the alignment information. To the best of our knowledge, this is the first work that developed and validated the cross-modality alignment representation of the source-code program and its CDFGs. Although currently targeted for HLS designs, this approach can benefit more general software compilation tasks where the CDFG representation is also widely used.

Chapter 8: As the final part of this dissertation, we conclude our contributions with a summary of our research works.

CHAPTER 2

SimGNN: A Neural Network Approach to Fast Graph Similarity Computation

2.1 Introduction

Graphs are ubiquitous nowadays and have a wide range of applications in bioinformatics, chemistry, recommender systems, social network study, program static analysis, etc. Among these, one of the fundamental problems is to retrieve a set of similar graphs from a database given a user query. Different graph similarity/distance metrics are defined, such as Graph Edit Distance (GED) [17], Maximum Common Subgraph (MCS) [23], etc. However, the core operation, namely computing the GED or MCS between two graphs, is known to be NP-complete [23, 24]. For GED, even the state-of-the-art algorithms cannot reliably compute the exact GED within reasonable time between graphs with more than 16 nodes [25].

Given the huge importance yet great difficulty of computing the exact graph distances, there have been two broad categories of methods to address the problem of graph similarity search. The first category of remedies is the pruning-verification framework [24, 26, 27], under which the total amount of exact graph similarity computations for a query can be reduced to a tractable degree, via a series of database indexing techniques and pruning strategies. However, the fundamental problem of the exponential time complexity of exact graph similarity computation [28] remains. The second category tries to reduce the cost of graph similarity computation directly. Instead of calculating the exact similarity metric, these methods find approximate values in a fast and heuristic way [28, 29, 30, 31, 32]. However, these methods usually require rather complicated design and implementation based on

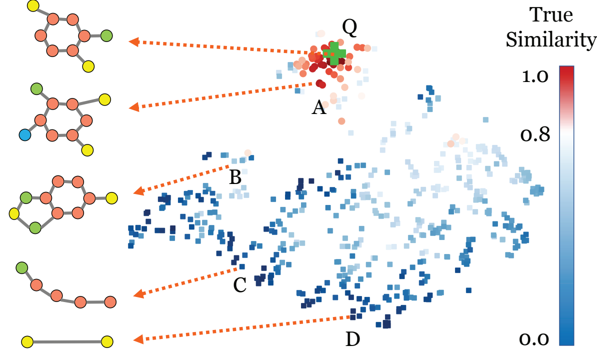


Figure 2.1: Illustration of similarity-preserving graph embedding. Each graph is mapped into an embedding vector (denoted as a dot in the plot), which preserves their similarity between each other in terms of a specific graph similarity metric. The green "+" sign denotes the embedding of an example query graph. Colors of dots indicate how similar a graph is to the query based on the ground truth (from red to blue, meaning from the most similar to the least similar).

discrete optimization or combinatorial search. The time complexity is usually still polynomial or even sub-exponential in the number of nodes in the graphs, such as A*-Beamsearch (Beam) [28], Hungarian [29], VJ [30], etc.

In this chapter, we propose a novel approach to speed-up the graph similarity computation, with the same purpose as the second category of methods mentioned previously. However, instead of directly computing the approximate similarities using combinatorial search, our solution turns it into a *learning* problem. More specifically, we design a neural network-based function that maps a pair of graphs into a similarity score. At the training stage, the parameters involved in this function will be learned by minimizing the difference between the predicted similarity scores and the ground truth, where each training data point is a pair of graphs together with their true similarity score. At the test stage, by feeding the learned function with any pair of graphs, we can obtain a predicted similarity score. We name such approach as ***SimGNN***, i.e., Similarity Computation via Graph Neural Networks.

SimGNN enjoys the key advantage of efficiency due to the nature of neural network computation. As for effectiveness, however, we need to carefully design the neural network architecture to satisfy the following three properties:

1. ***Representation-invariant***. The same graph can be represented by different adja-

cency matrices by permuting the order of nodes. The computed similarity score should be invariant to such changes.

2. ***Inductive***. The similarity computation should generalize to unseen graphs, i.e. compute the similarity score for graphs outside the training graph pairs.

3. ***Learnable***. The model should be adaptive to any similarity metric, by adjusting its parameters through training.

To achieve these goals, we propose the following two strategies. First, we design a learnable embedding function that maps every graph into an embedding vector, which provides a global summary of a graph through aggregating node-level embeddings. We propose a novel attention mechanism to select the important nodes out of an entire graph with respect to a specific similarity metric. This graph-level embedding can already largely preserve the similarity between graphs. For example, as illustrated in Fig. 2.1, Graph A is very similar to Graph Q according to the ground truth similarity, which is reflected by the embedding as its embedding is close to Q in the embedding space. Also, such embedding-based similarity computation is very fast. Second, we design a pairwise node comparison method to supplement the graph-level embeddings with fine-grained node-level information. As one fixed-length embedding per graph may be too coarse, we further compute the pairwise similarity scores between nodes from the two graphs, from which the histogram features are extracted and combined with the graph-level information to boost the performance of our model. This results in the quadratic amount of operations in terms of graph size, which, however, is still among the most efficient methods for graph similarity computation.

We conduct our experiments on GED computation, which is one of the most popular graph similarity/distance metrics. To demonstrate the effectiveness and efficiency of our approach, we conduct experiments on three real graph datasets. Compared with the baselines, which include several approximate GED computation algorithms, and many graph neural network based methods, our model achieves smaller error and great time reduction. It is worth mentioning that, our Strategy 1 already demonstrates superb performances compared

with existing solutions. When running time is a major concern, we can drop the more time-consuming Strategy 2 for trade-off.

Our contributions can be summarized as follows:

- We address the challenging while classic problem of graph similarity computation by considering it as a learning problem, and propose a neural network based approach, called SimGNN, as the solution.

- Two novel strategies are proposed. First, we propose an efficient and effective attention mechanism to select the most relevant parts of a graph to generate a graph-level embedding, which preserves the similarity between graphs. Second, we propose a pairwise node comparison method to supplement the graph-level embeddings for more effective modeling of the similarity between two graphs.

- We conduct extensive experiments on a very popular graph similarity/distance metric, GED, based on three real network datasets to demonstrate the effectiveness and efficiency of the proposed approach.

2.2 Preliminaries

2.2.1 Graph Edit Distance (GED)

In order to demonstrate the effectiveness and efficiency of SimGNN, we choose one of the most popular graph similarity/distance metric, Graph Edit Distance (GED), as a case study. GED has been widely used in many applications, such as graph similarity search [1, 24, 26, 27, 33], graph classification [29, 34], handwriting recognition [35], image indexing [36], etc.

Formally, the edit distance between $\mathcal{G}_1$ and $\mathcal{G}_2$, denoted by $\text{GED}(\mathcal{G}_1, \mathcal{G}_2)$, is the number of edit operations in the optimal alignments that transform $\mathcal{G}_1$ into $\mathcal{G}_2$, where an edit operation on a graph $\mathcal{G}$ is an insertion or deletion of a vertex/edge or relabelling of a vertex [1]. Intuitively, if two graphs are identical (isomorphic), their GED is 0. Fig. 3.1 shows an

[1] Although other variants of GED exist [37], we adopt this basic version.

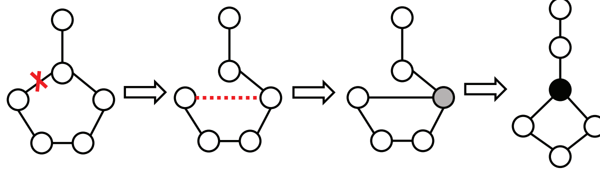


Figure 2.2: The GED between the graph to the left and the graph to the right is 3, as the transformation needs 3 edit operations: (1) an edge deletion, (2) an edge insertion, and (3) a node relabeling.

example of GED between two simple graphs.

Once the distance between two graphs is calculated, we transform it to a similarity score ranging between 0 and 1. More details about the transformation function can be found in Section 2.4.2.

2.2.2 Graph Convolutional Networks (GCN)

Both strategies in SimGNN require node embedding computation. In Strategy 1, to compute graph-level embedding, it aggregates node-level embeddings using attention; and in Strategy 2, pairwise node comparison for two graphs is computed based on node-level embeddings as well.

Among many existing node embedding algorithms, we choose to use Graph Convolutional Networks (GCN) [7], as it is graph *representation-invariant*, as long as the initialization is carefully designed. It is also *inductive*, since for any unseen graph, we can always compute the node embedding following the GCN operation. GCN now is among the most popular models for node embeddings, and belong to the family of neighbor aggregation based methods. Its core operation, graph convolution, operates on the representation of a node, which is denoted as $\boldsymbol{u_n} \in \mathbb{R}^D$, and is defined as follows:

$$\mathrm{conv}(\boldsymbol{u_n}) = f_1\Big(\sum_{m \in \mathcal{N}(n)} \frac{1}{\sqrt{d_n d_m}} \boldsymbol{u_m} \boldsymbol{W_1}^{(l)} + \boldsymbol{b_1}^{(l)}\Big) \tag{2.1}$$

where $\mathcal{N}(n)$ is the set of the first-order neighbors of node n plus n itself, d_n is the degree of node n plus 1, $\boldsymbol{W_1}^{(l)} \in \mathbb{R}^{D^l \times D^{l+1}}$ is the weight matrix associated with the l-th GCN layer, $\boldsymbol{b_1}^{(l)} \in \mathbb{R}^{D^{l+1}}$ is the bias, and $f_1(\cdot)$ is an activation function such as $\mathrm{ReLU}(x) = \max(0, x)$. In-

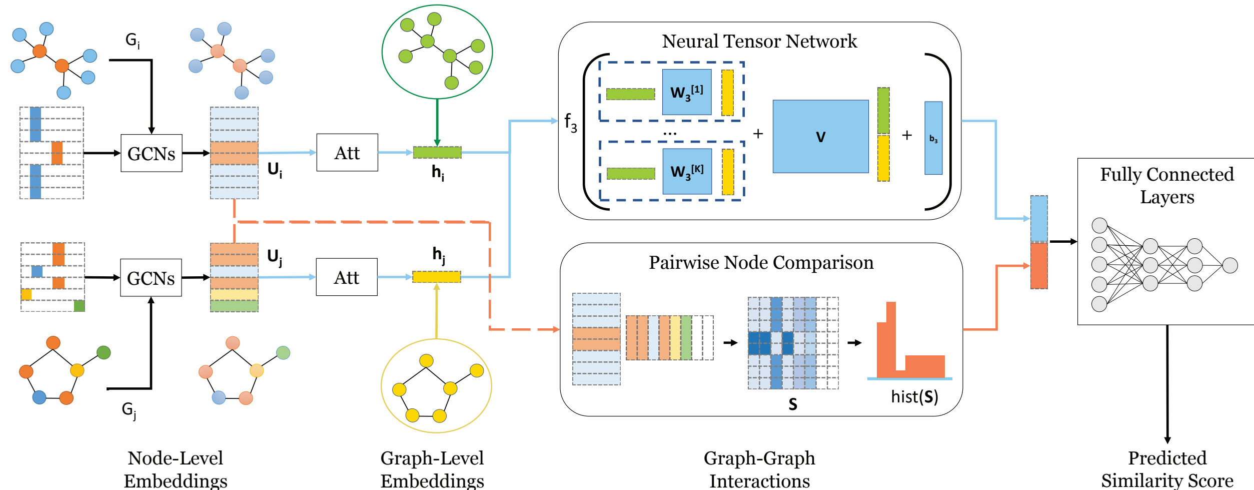


Figure 2.3: An overview illustration of SimGNN. The blue arrows denote the data flow for Strategy 1, which is based on graph-level embeddings. The red arrows denote the data flow for Strategy 2, which is based on pairwise node comparison.

tuitively, the graph convolution operation aggregates the features from the first-order neighbors of the node.

2.3 The Proposed Approach: SimGNN

Now we introduce our proposed approach SimGNN in detail, which is an end-to-end neural network based approach that attempts to learn a function to map a pair of graphs into a similarity score. An overview of SimGNN is illustrated in Fig. 7.1. First, our model transforms the node of each graph into a vector, encoding the features and structural properties around each node. Then, two strategies are proposed to model the similarity between the two graphs, one based on the interaction between two graph-level embeddings, the other based on comparing two sets of node-level embeddings. Finally, two strategies are combined together to feed into a fully connected neural network to get the final similarity score. The rest of the section details these two strategies.

2.3.1 Strategy One: Graph-Level Embedding Interaction

This strategy is based on the assumption that a good graph-level embedding can encode the structural and feature information of a graph, and by interacting the two graph-level embeddings, the similarity between two graphs can be predicted. It involves the following stages: (1) the node embedding stage, which transforms each node of a graph into a vector, encoding its features and structural properties; (2) the graph embedding stage, which produces one embedding for each graph by attention-based aggregation of node embeddings generated in the previous stage; (3) the graph-graph interaction stage, which receives two graph-level embeddings and returns the interaction scores representing the graph-graph similarity; and (4) the final graph similarity score computation stage, which further reduces the interaction scores into one final similarity score. It will be compared against the ground-truth similarity score to update parameters involved in the 4 stages.

2.3.1.1 Stage I: Node Embedding

Among the existing state-of-the-art approaches, we adopt GCN, a neighbor aggregation based method, because it learns an aggregation function (Eq. 3.1) that are representation-invariant and can be applied to unseen nodes. In Fig. 7.1, different colors represent different node types, and the original node representations are one-hot encoded. Notice that the one-hot encoding is based on node types (e.g., all the nodes with carbon type share the same one-hot encoding vector), so even if the node ids are permuted, the aggregation results would be the same. For graphs with unlabeled nodes, we treat every node to have the same label, resulting in the same constant number as the initialize representation. After multiple layers of GCNs (e.g., 3 layers in our experiment), the node embeddings are ready to be fed into the Attention module (Att), which is described as follows.

2.3.1.2 Stage II: Graph Embedding: Global Context-Aware Attention.

To generate one embedding per graph using a set of node embeddings, one could perform an unweighted average of node embeddings, or a weighted sum where a weight associated with a node is determined by its degree. However, which nodes are more important and should receive more weights is dependent on the specific similarity metric. Thus, we propose the following attention mechanism to let the model learn weights guided by the specific similarity metric.

Denote the input node embeddings as $\boldsymbol{U} \in \mathbb{R}^{N \times D}$, where the n-th row, $\boldsymbol{u_n} \in \mathbb{R}^D$ is the embedding of node n. First, a global graph context $\boldsymbol{c} \in \mathbb{R}^D$ is computed, which is a simple average of node embeddings followed by a nonlinear transformation: $\boldsymbol{c} = \tanh((\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{u_n})\boldsymbol{W_2})$, where $\boldsymbol{W_2} \in \mathbb{R}^{D \times D}$ is a learnable weight matrix. The context $\boldsymbol{c}$ provides the global structural and feature information of the graph that is adaptive to the given similarity metric, via learning the weight matrix. Based on $\boldsymbol{c}$, we can compute one attention weight for each node.

For node n, to make its attention a_n aware of the global context, we take the inner product between $\boldsymbol{c}$ and its node embedding. The intuition is that, nodes similar to the global context should receive higher attention weights. A sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$ is applied to the result to ensure the attention weights is in the range $(0, 1)$. We do not normalize the weights into length 1, since it is desirable to let the embedding norm reflect the graph size, which is essential for the task of graph similarity computation. Finally, the graph embedding $\boldsymbol{h} \in \mathbb{R}^D$ is the weighted sum of node embeddings, $\boldsymbol{h} = \sum_{n=1}^{N} a_n\boldsymbol{u_n}$. The following equation summarizes the proposed node attentive mechanism:

$$\boldsymbol{h} = \sum_{n=1}^{N} f_2(\boldsymbol{u_n^T c})\boldsymbol{u_n} = \sum_{n=1}^{N} f_2(\boldsymbol{u_n^T}\tanh((\frac{1}{N}\sum_{m=1}^{N}\boldsymbol{u_m})\boldsymbol{W_2}))\boldsymbol{u_n} \tag{2.2}$$

where $f_2(\cdot)$ is the sigmoid function $\sigma(\cdot)$.

2.3.1.3 Stage III: Graph-Graph Interaction: Neural Tensor Network

Given the graph-level embeddings of two graphs produced by the previous stage, a simple way to model their relation is to take the inner product of the two, $\boldsymbol{h_i} \in \mathbb{R}^D$, $\boldsymbol{h_j} \in \mathbb{R}^D$. However, as discussed in [38], such simple usage of data representations often lead to insufficient or weak interaction between the two. Following [38], we use Neural Tensor Networks (NTN) to model the relation between two graph-level embeddings:

$$g(\boldsymbol{h_i}, \boldsymbol{h_j}) = f_3(\boldsymbol{h_i^T} \boldsymbol{W_3^{[1:K]}} \boldsymbol{h_j} + \boldsymbol{V} \left[\begin{smallmatrix} \boldsymbol{h_i} \\ \boldsymbol{h_j} \end{smallmatrix} \right] + \boldsymbol{b_3}) \tag{2.3}$$

where $\boldsymbol{W_3^{[1:K]}} \in \mathbb{R}^{D \times D \times K}$ is a weight tensor, [] denotes the concatenation operation, $\boldsymbol{V} \in \mathbb{R}^{K \times 2D}$ is a weight vector, $\boldsymbol{b_3} \in \mathbb{R}^K$ is a bias vector, and $f_3(\cdot)$ is an activation function. K is a hyperparameter controlling the number of interaction (similarity) scores produced by the model for each graph embedding pair.

2.3.1.4 Stage IV: Graph Similarity Score Computation

After obtaining a list of similarity scores, we apply a standard multi-layer fully connected neural network to gradually reduce the dimension of the similarity score vector. In the end, one score, $\hat{s_{ij}} \in \mathbb{R}$, is predicted, and it is compared against the ground-truth similarity score using the following mean squared error loss function:

$$\mathcal{L} = \frac{1}{|\mathcal{D}|} \sum_{(i,j) \in \mathcal{D}} (\hat{s_{ij}} - s(\mathcal{G}_i, \mathcal{G}_j))^2 \tag{2.4}$$

where $\mathcal{D}$ is the set of training graph pairs, and $s(\mathcal{G}_i, \mathcal{G}_j)$ is the ground-truth similarity between $\mathcal{G}_i$ and $\mathcal{G}_j$.

2.3.1.5 Limitations of Strategy One

As mentioned in Section 2.1, the node-level information such as the node feature distribution and graph size may be lost by the graph-level embedding. In many cases, the differences

between two graphs lie in small substructures and are hard to be reflected by the graph level embedding. An analogy is that, in Natural Language Processing, the performance of sentence matching based on one embedding per sentence can be further enhanced through using fine-grained word-level information [39, 40]. This leads to our second strategy.

2.3.2 Strategy Two: Pairwise Node Comparison

To overcome the limitations mentioned previously, we consider bypassing the NTN module, and using the node-level embeddings directly. As illustrated in the bottom data flow of Fig. 7.1, if $\mathcal{G}_i$ has N_i nodes and $\mathcal{G}_j$ has N_j nodes, there would be $N_i N_j$ pairwise interaction scores, obtained by $\boldsymbol{S} = \sigma(\boldsymbol{U_i U_j^T})$, where $\boldsymbol{U_i} \in \mathbb{R}^{N_i \times D}$ and $\boldsymbol{U_j} \in \mathbb{R}^{N_j \times D}$ are the node embeddings of $\mathcal{G}_i$ and $\mathcal{G}_j$, respectively. Since the node-level embeddings are not normalized, the sigmoid function is applied to ensure the similarities scores are in the range of $(0, 1)$. For two graphs of different sizes, to emphasize their size difference, we pad fake nodes to the smaller graph. As shown in Fig. 7.1, two fake nodes with zero embedding are padded to the bottom graph, resulting in two extra columns with zeros in $\boldsymbol{S}$.

Denote $N = \max(N_1, N_2)$. The pairwise node similarity matrix $\boldsymbol{S} \in \mathbb{R}^{N \times N}$ is a useful source of information, since it encodes fine-grained pairwise node similarity scores. One simple way to utilize $\boldsymbol{S}$ is to vectorize it: $\text{vec}(\boldsymbol{S}) \in \mathbb{R}^{N^2}$, and feed it into the fully connected layers. However, there is usually no natural ordering between graph nodes. Different initial node ordering of the same graph would lead to different $\boldsymbol{S}$ and $\text{vec}(\boldsymbol{S})$.

To ensure the model is invariant to the graph representations as mentioned in Section 2.1, one could preprocess the graph by applying some node ordering scheme [41], but we consider a much more efficient and natural way to utilize $\boldsymbol{S}$. We extract its histogram features: $\text{hist}(\boldsymbol{S}) \in \mathbb{R}^B$, where B is a hyperparameter that controls the number of bins in the histogram. In the case of Fig. 7.1, seven bins are used for the histogram. The histogram feature vector is normalized and concatenated with the graph-level interaction scores $g(\boldsymbol{h_i}, \boldsymbol{h_j})$, and fed to the fully connected layers to obtain a final similarity score for the graph pair.

It is important to note that the histogram features alone are not enough to train the

Table 2.1: Statistics of datasets.

Dataset	Graph Meaning	#Graphs	#Pairs
AIDS	Chemical Compounds	700	490K
LINUX	Program Dependency Graphs	1000	1M
IMDB	Actor/Actress Ego-Networks	1500	2.25M

model, since the histogram is not a continuous differential function and does not support backpropagation. In fact, we rely on Strategy 1 as the primary strategy to update the model weights, and use Strategy 2 to supplement the graph-level features, which brings extra performance gain to our model.

To sum up, we combine the coarse global comparison information captured by Strategy 1, and the fine-grained node-level comparison information captured by Strategy 2, to provide a thorough view of the graph comparison to the model.

2.3.3 Time Complexity Analysis

Once SimGNN has been trained, it can be used to compute a similarity score for any pair of graphs. The time complexity involves two parts: (1) the node-level and global-level embedding computation stages, which needs to be computed once for each graph; and (2) the similarity score computation stage, which needs to be computed for every pair of graphs.

The node-level and global-level embedding computation stages. The time complexity associated with the generation of node-level and graph-level embeddings is $O(E)$ [7], where E is the number of edges of the graph. Notice that the graph-level embeddings can be pre-computed and stored, and in the setting of graph similarity search, the unseen query graph only needs to be processed once to obtain its graph-level embedding.

The similarity score computation stage. The time complexity for Strategy 1 is $O(D^2K)$, where D is the dimension of the graph level embedding, and K is the feature map dimension of the NTN. The time complexity for our Strategy 2 is $O(DN^2)$, where N is the number of nodes in the larger graph. This can potentially be reduced by node sampling to construct the similarity matrix S. Moreover, the matrix multiplication $\boldsymbol{S} = \sigma(\boldsymbol{U_1}\boldsymbol{U_2^T})$ can be greatly accelerated with GPUs. Our experimental results in Section 2.4.6.2 verify that there is no

significant runtime increase when the second strategy is used.

In conclusion, among the two strategies we have proposed: Strategy 1 is the primary strategy, which is efficient but solely based on coarse graph level embeddings; and Strategy 2 is auxiliary, which includes fine-grained node-level information but is more time-consuming. In the worst case, the model runs in quadratic time with respect to the number of nodes, which is among the state-of-the-art algorithms for approximate graph distance computation.

2.4 Experiments

2.4.1 Datasets

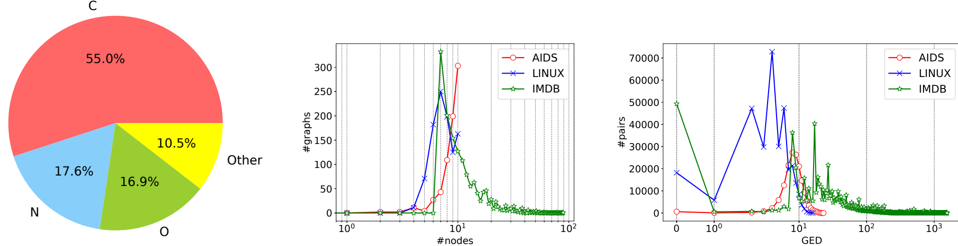
Three real-world graph datasets are used for the experiments. A concise summary and detailed visualizations can be found in Table 2.1 and Fig. 2.4, respectively.

AIDS. AIDS is a collection of antivirus screen chemical compounds from the Developmental Therapeutics Program at NCI/NIH 7 [2], and has been used in several existing works on graph similarity search [1, 24, 26, 27, 33]. It contains 42,687 chemical compound structures with Hydrogen atoms omitted. We select 700 graphs, each of which has 10 or less than 10 nodes. Each node is labeled with one of 29 types, as illustrated in Fig. 2.4a.

LINUX. The LINUX dataset was originally introduced in [1]. It consists of 48,747 Program Dependence Graphs (PDG) generated from the Linux kernel. Each graph represents a function, where a node represents one statement and an edge represents the dependency between the two statements. We randomly select 1000 graphs of equal or less than 10 nodes each. The nodes are unlabeled.

IMDB. The IMDB dataset [13] (named "IMDB-MULTI") consists of 1500 ego-networks of movie actors/actresses, where there is an edge if the two people appear in the same movie. To test the scalability and efficiency of our proposed approach, we use the full dataset without any selection. The nodes are unlabeled.

[2] https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data



(a) Node label distribution of AIDS. (b) Distribution of graph sizes. (c) Distribution of GEDs of the training pairs.

Figure 2.4: Some statistics of the datasets.

2.4.2 Data Preprocessing

For each dataset, we randomly split 60%, 20%, and 20% of all the graphs as training set, validation set, and testing set, respectively. The evaluation reflects the real-world scenario of graph query: For each graph in the testing set, we treat it as a query graph, and let the model compute the similarity between the query graph and every graph in the database. The database graphs are ranked according to the computed similarities to the query.

Since graphs from AIDS and LINUX are relatively small, an exponential-time exact GED computation algorithm named A* [37] is used to compute the GEDs between all the graph pairs. For the IMDB dataset, however, A* can no longer be used, as a recent survey of exact GED computation [25] concludes, "no currently available algorithm manages to reliably compute GED within reasonable time between graphs with more than 16 nodes."

To properly handle the IMDB dataset, we take the smallest distance computed by three well-known approximate algorithms, Beam [28], Hungarian [29, 42], and VJ [30, 43]. The minimum is taken instead of the average, because their returned GEDs are guaranteed to be greater than or equal to the true GEDs. Details on these algorithms can be found in Section 2.4.3. Incidentally, the ICPR 2016 Graph Distance Contest [3] also adopts this approach to obtaining ground-truth GEDs for large graphs.

To transform ground-truth GEDs into ground-truth similarity scores to train our model, we first normalize the GEDs according to [44]: $\text{nGED}(\mathcal{G}_1, \mathcal{G}_2) = \frac{\text{GED}(\mathcal{G}_1,\mathcal{G}_2)}{(|\mathcal{G}_1|+|\mathcal{G}_2|)/2}$, where $|\mathcal{G}_i|$ denotes the number of nodes of $\mathcal{G}_i$. We then adopt the exponential function $\lambda(x) = e^{-x}$

[3] https://gdc2016.greyc.fr/

to transform the normalized GED into a similarity score in the range of $(0, 1]$. Notice that there is a one-to-one mapping between the GED and the similarity score.

2.4.3 Baseline Methods

Our baselines include two types of approaches, fast approximate GED computation algorithms and neural network based models.

The first category of baselines includes three classic algorithms for GED computation. (1) *A*-Beamsearch (Beam)* [28]. It is a variant of the A* algorithm in sub-exponential time. (2) *Hungarian* [29, 42] and (3) *VJ* [30, 43] are two cubic-time algorithms based on the Hungarian Algorithm for bipartite graph matching, and the algorithm of Volgenant and Jonker, respectively.

The second category of baselines includes seven models of the following neural network architectures. (1) *SimpleMean* simply takes the unweighted average of all the node embeddings of a graph to generate its graph-level embedding. (2) *HierarchicalMean* and (3) *HierarchicalMax* [6] are the original GCN architectures based on graph coarsening, which use the global mean or max pooling to generate a graph hierarchy. We use the implementation from the Github repository of the first author of GCN [4]. The next four models apply the attention mechanism on nodes. (4) *AttDegree* uses the natural log of the degree of a node as its attention weight, as described in Section 2.3.1.2. (5) *AttGlobalContext* and (6) *AttLearnableGlobalContext (AttLearnableGC)* both utilize the global graph context to compute the attention weights, but the former does not apply the nonlinear transformation with learnable weights on the context, while the latter does. (7) *SimGNN* is our full model that combines the best of Strategy 1 (AttLearnableGC) and Strategy 2 as described in Section 2.3.2.

[4] https://github.com/mdeff/cnn_graph

2.4.4 Parameter Settings

For the model architecture, we set the number of GCN layers to 3, and use ReLU as the activation function. For the initial node representations, we adopt the one-hot encoding scheme for AIDS reflecting the node type, and the constant encoding scheme for LINUX and IMDB, since their nodes are unlabeled, as mentioned in Section 2.3.1.1. The output dimensions for the 1st, 2nd, and 3rd layer of GCN are 64, 32, and 16, respectively. For the NTN layer, we set K to 16. For the pairwise node comparison strategy, we set the number of histogram bins to 16. We use 4 fully connected layers to reduce the dimension of the concatenated results from the NTN module, from 32 to 16, 16 to 8, 8 to 4, and 4 to 1.

We conduct all the experiments on a single machine with an Intel i7-6800K CPU and one Nvidia Titan GPU. As for training, we set the batch size to 128, use the Adam algorithm for optimization [45], and fix the initial learning rate to 0.001. We set the number of iterations to 10000, and select the best model based on the lowest validation loss.

2.4.5 Evaluation Metrics

The following metrics are used to evaluate all the models: *Time.* The wall time needed for each model to compute the similarity score for a pair of graphs is collected. *Mean Squared Error (mse).* The mean squared error measures the average squared difference between the computed similarities and the ground-truth similarities.

We also adopt the following metrics to evaluate the ranking results. *Spearman's Rank Correlation Coefficient (ρ)* [46] and *Kendall's Rank Correlation Coefficient (τ)* [47] measure how well the predicted ranking results match the true ranking results. *Precision at k (p@k).* p@k is computed by taking the intersection of the predicted top k results and the ground-truth top k results divided by k. Compared with p@k, ρ and τ evaluates the global ranking results instead of focusing on the top k results.

2.4.6 Results

2.4.6.1 Effectiveness

The effectiveness results on the three datasets can be found in Table 2.2, 2.3, and 2.4. Our model, SimGNN, consistently achieves the best or the second best performance on all metrics across the three datasets. Within the neural network based methods, SimGNN consistently achieves the best results on all metrics. This suggests that our model can learn a good embedding function that generalizes to unseen test graphs.

Beam achieves the best precisions at 10 on AIDS and LINUX. We conjecture that it can be attributed to the imbalanced ground-truth GED distributions. As seen in Fig. 2.4c, for AIDS, the training pairs have GEDs mostly around 10, causing our model to train the very similar pairs less frequently than the dissimilar ones. For LINUX, the situation for SimGNN is better, since most GEDs concentrate in the range of [0, 10], the gap between the precisions at 10 of Beam and SimGNN become smaller.

It is noteworthy that among the neural network based models, AttDegree achieves relatively good results on IMDB, but not on AIDS or LINUX. It could be due to the unique ego-network structures commonly present in IMDB. As seen in Fig. 2.9, the high-degree central node denotes the particular actor/actress himself/herself, focusing on which could be a reasonable heuristic. In contrast, AttLearnableGC adapts to the GED metric via a learnable global context, and consistently performs better than AttDegree. Combined with Strategy 2, SimGNN achieves even better performances.

Visualizations of the node attentions can be seen in Fig. 2.5. We observe that the following kinds of nodes receive relatively higher attention weights: hub nodes with large degrees, e.g. the "S" in (a) and (b), nodes with labels that rarely occur in the dataset, e.g. the "Ni" in (f), the "Pd" in (g), the "Br" in (h), nodes forming special substructures, e.g. the two middle "C"s in (e), etc. These patterns make intuitive sense, further confirming the effectiveness of the proposed approach.

Table 2.2: Results on AIDS.

Method	mse(10^{-3})	ρ	τ	p@10	p@20
Beam	12.090	0.609	0.463	**0.481**	0.493
Hungarian	25.296	0.510	0.378	0.360	0.392
VJ	29.157	0.517	0.383	0.310	0.345
SimpleMean	3.115	0.633	0.480	0.269	0.279
HierarchicalMean	3.046	0.681	0.629	0.246	0.340
HierarchicalMax	3.396	0.655	0.505	0.222	0.295
AttDegree	3.338	0.628	0.478	0.209	0.279
AttGlobalContext	1.472	0.813	0.653	0.376	0.473
AttLearnableGC	1.340	0.825	0.667	0.400	0.488
SimGNN	**1.189**	**0.843**	**0.690**	**0.421**	**0.514**

Table 2.3: Results on LINUX.

Method	mse(10^{-3})	ρ	τ	p@10	p@20
Beam	9.268	0.827	0.714	**0.973**	0.924
Hungarian	29.805	0.638	0.517	0.913	0.836
VJ	63.863	0.581	0.450	0.287	0.251
SimpleMean	16.950	0.020	0.016	0.432	0.465
HierarchicalMean	6.431	0.430	0.525	0.750	0.618
HierarchicalMax	6.575	0.879	0.740	0.551	0.575
AttDegree	8.064	0.742	0.609	0.427	0.460
AttGlobalContext	3.125	0.904	0.781	0.874	0.864
AttLearnableGC	2.055	0.916	0.804	0.903	0.887
SimGNN	**1.509**	**0.939**	**0.830**	**0.942**	**0.933**

Table 2.4: Results on IMDB. Beam, Hungarian, and VJ together are used to determine the ground-truth results.

Method	mse(10^{-3})	ρ	τ	p@10	p@20
SimpleMean	3.749	0.774	0.644	0.547	0.588
HierarchicalMean	5.019	0.456	0.378	0.567	0.553
HierarchicalMax	6.993	0.455	0.354	0.572	0.570
AttDegree	2.144	0.828	0.695	0.700	0.695
AttGlobalContext	3.555	0.684	0.553	0.657	0.656
AttLearnableGC	1.455	0.835	0.700	0.732	0.742
SimGNN	**1.264**	**0.878**	**0.770**	**0.759**	**0.777**

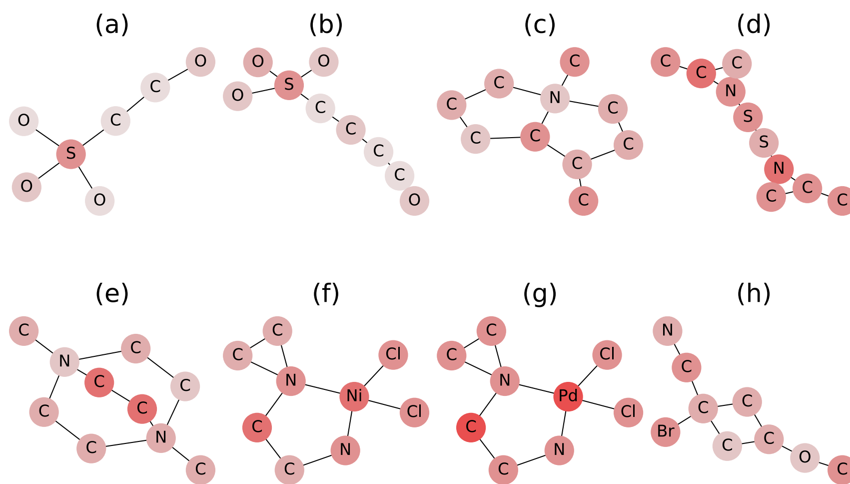


Figure 2.5: Visualizations of node attentions. The darker the color, the larger the attention weight.

2.4.6.2 Efficiency

The efficiency comparison on the three datasets is shown in Fig. 3.3. The neural network based models consistently achieve the best results across all the three datasets. Specifically, compared with the exact algorithm, A*, SimGNN is 2174 times faster on AIDS, and 212 times faster on LINUX. The A* algorithm cannot even be applied on large graphs, and in the case of IMDB, its variant, Beam, is still 46 times slower than SimGNN. Moreover, the time measured for SimGNN includes the time for graph embedding. As mentioned in Section 2.3.3, if graph embeddings are pre-computed and stored, SimGNN would spend even less time. All of these suggest that in practice, it is reasonable to use SimGNN as a fast approach to graph similarity computation, which is especially true for large graphs, as in IMDB, our computation time does not increase much compared with AIDS and LINUX.

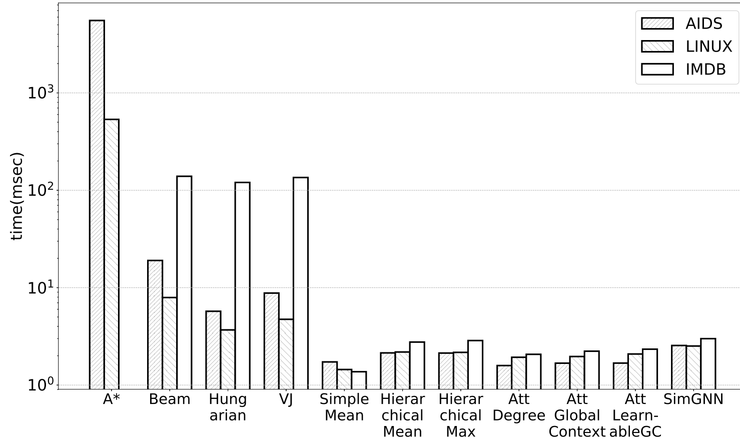


Figure 2.6: Runtime comparison.

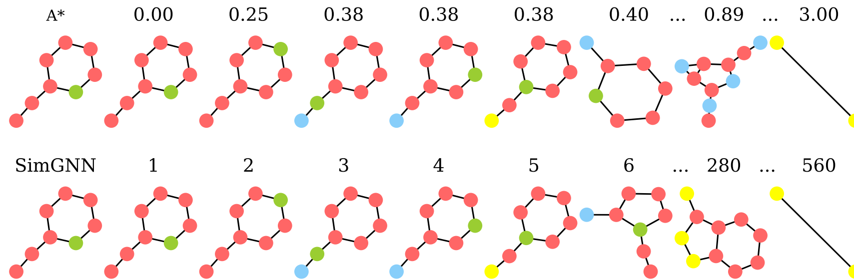


Figure 2.7: A query case study on AIDS. Meanings of the colors can be found in Fig. 2.4a.

2.4.7 Parameter Sensitivity

We evaluate how the dimension of the graph-level embeddings and the number of the histogram bins can affect the results. We report the mean squared error on AIDS. As can be seen in Fig. 4.3, the performance becomes better if larger dimensions are used. This makes intuitive sense, since larger embedding dimensions give the model more capacity to represent graphs. In our Strategy 2, as shown in Fig. 4.3, the performance is relatively insensitive to the number of histogram bins. This suggests that in practice, as long as the histogram bins are not too few, relatively good performance can be achieved.

2.4.8 Case Studies

We demonstrate three example queries, one from each dataset, in Fig. 2.7, 2.8, and 2.9. In each demo, the top row depicts the query along with the ground-truth ranking results, labeled

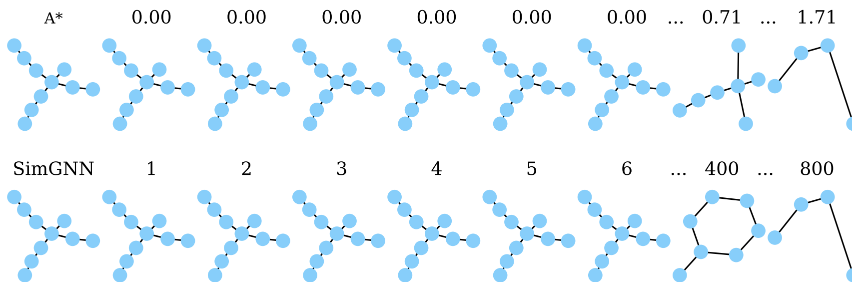


Figure 2.8: A query case study on LINUX.

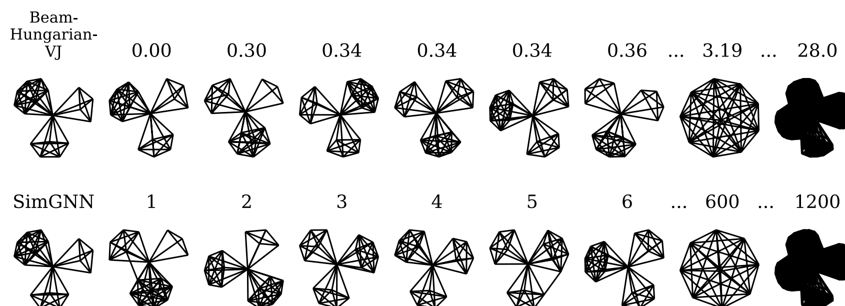


Figure 2.9: A query case study on IMDB.

with their normalized GEDs to the query; The bottom row shows the graphs returned by our model, each with its rank shown at the top. SimGNN is able to retrieve graphs similar to the query, e.g. in the case of LINUX (Fig. 2.8), the top 6 results are the isomorphic graphs to the query.

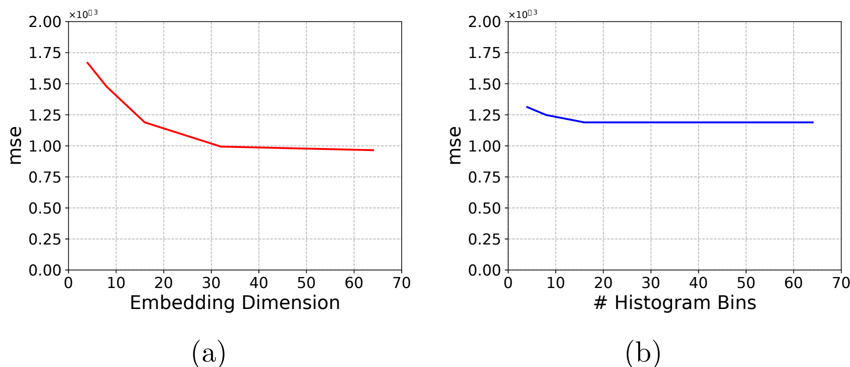


Figure 2.10: Mean squared error w.r.t. the number of dimensions of graph-level embeddings, and the number of histogram bins.

2.5 Related Work

2.5.1 Network/Graph Embedding

Node-level embedding. Over the years, there are several categories of methods that have been proposed for learning node representations, including matrix factorization based methods (NetMF [48]), skip-gram based methods (DeepWalk [49], Node2Vec [50], LINE [51]), autoencoder based methods (SDNE [52]), neighbor aggregation based methods (GCN [6, 7], GraphSAGE [8]), etc.

Graph-level embedding. The most intuitive way to generate one embedding per graph is to aggregate the node-level embeddings, either by a simple average or some weighted average [53, 54], named the "sum-based" approaches [55]. A more sophisticated way to represent graphs can be achieved by viewing a graph as a hierarchical data structure and applying graph coarsening [6, 14]. Besides, [56] aggregates sets of nodes via histograms, and [41] applies node ordering on a graph to make it CNN suitable.

Attention mechanism on graphs. [57] uses an attention-guided walk to find the most relevant parts for graph classification. As a result, it only selects a fixed amount of nodes out of an entire graph, which is too coarse for our task, graph similarity computation. [9, 58] assign an attention weight to each neighbor of a node for node-level tasks. We are among the first to apply the attention mechanism for the task of graph similarity computation.

Graph neural network applications. A great amount of graph-based applications have been tackled by neural network based methods, most of which are framed as node-level prediction tasks [59, 60]. However, once moving to the graph-level tasks, most existing works deal with the classification of a single graph [6, 14, 41]. In this work, we consider graph similarity computation for the first time.

2.5.2 Graph Similarity Computation

Graph distance/similarity metrics. The Graph Edit Distance (GED) [17] can be considered as an extension of the String Edit Distance metric [61], which is defined as the minimum cost taken to transform one graph to the other via a sequence graph edit operations. Another metric is the Maximum Common Subgraph (MCS), which is equivalent to GED under a certain cost function [18]. Graph kernels [13, 62, 63, 64] can be considered as a family of different graph similarity metrics, used primarily for graph classification.

Pairwise GED computation algorithms. A flurry of approximate algorithms has been proposed to reduce the time complexity with the sacrifice in accuracy [28, 29, 30, 31, 32]. We are aware of some very recent work claiming their time complexity is $O(n^2)$ [31], but their code is unstable at this stage for comparison.

Graph Similarity search.

Computing GED is a primitive operator in graph database analysis, and has been adopted in a series of works on graph similarity search [1, 24, 26, 27, 33]. These studies focus on database-level techniques to speed up the overall querying process involving exact GED computations.

2.6 Conclusion

We are at the intersection of graph deep learning and graph search problem, and taking the first step towards bridging the gap, by tackling the core operation of graph similarity computation , via a novel neural network based approach. The central idea is to learn a neural network based function that is representation-invariant, inductive, and adaptive to the specific similarity metric, which takes any two graphs as input and outputs their similarity score. Our model runs very fast compared to existing classic algorithms on approximate Graph Edit Distance computation, and achieves very competitive accuracy.

CHAPTER 3

Learning-based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching

3.1 Introduction

Recent years we have witnessed the growing importance of graph-based applications in the domains of chemistry, bioinformatics, recommender systems, social network study, static program analysis, etc. One of the fundamental problems related to graphs is the computation of distance/similarity between two graphs. It not only is a core operation in graph similarity search and graph database analysis [1, 24], but also plays a significant role in a wide range of applications. For example, in computer security, similarity between binary functions is useful for plagiarism and malware detection [65]; in anomaly detection, similarity between communication graphs could help identify network intrusions from the graph-based connection records [66]; in social network analysis, similarity between different user message graphs may reveal interesting behavioral patterns [67].

Among various definitions of graph similarity/distance, Graph Edit Distance (GED) [17] and Maximum Common Subgraph (MCS) [23] are two popular and domain-agnostic metrics. However, the computation of exact GED and MCS is known to be NP-hard [23, 24], incurring significant computational burden in practice [25]. For example, a recent study shows that even the state-of-the-art algorithms cannot reliably compute the exact GED between graphs of more than 16 nodes within a reasonable time [25].

Given the great significance yet huge challenge of computing the exact graph distance/similarity, various approximate algorithms have been proposed to compute the graph distance/similarity in a fast but heuristic way, including traditional algorithmic approaches [29, 30, 32] as well as more recent data-driven neural network approaches [19, 68, 69, 70].

Compared with traditional algorithmic approaches which typically involve knowledge and heuristics specific to a metric, the neural network approaches *learn* graph similarity from data: During training, the parameters are learned by minimizing the loss between the predicted similarity scores and the ground truth; during testing, unseen pairs of graphs can be fed into these models for fast approximation of their similarities.

However, a major limitation of most current neural network models is that they rely on graph-level embeddings to model the similarity of graphs: Each graph is first represented as a fixed-length vector, and then the similarity of two graphs can be modeled as a vector operation on the two embeddings, e.g. cosine similarity. However, real-world graphs typically come in very different sizes, which the fixed-length vector representation may fail to fully capture. Even when the two graphs of interest are similar in sizes, the actual difference between them can lie in very small local substructures, which is hard to be captured by the single vector. This is especially problematic for the task of graph similarity computation, where the goal is to compare the difference between all nodes and edges of the two graphs. For simple or regular graphs, this approach may work well, but for more complicated scenarios in which graphs are of very different structures and/or the task is to find the fine-grained node-node correspondence [71], this approach often produces less effective models.

In this chapter, we propose to avoid the generation of graph-level embeddings, and instead directly perform neural operations on the two sets of node embeddings. Inspired by two classic families of algorithms for graph similarity/distance [29, 30, 64], our model GraphSim turns the two sets of node embeddings into a similarity matrix consisting of the pairwise node-node similarity scores, and is trained in an end-to-end fashion (Fig. 7.1). By carefully ordering the nodes in each graph, the similarity matrix encodes the similarity patterns specific to the graph pair, which allows the standard image processing techniques to be adapted to

model the graph-graph similarity. The new challenges in the graph setting compared to the standard image processing using Convolutional Neural Networks (CNN) are that:

- ***Permutation invariance***. The same graph can be represented by different adjacency matrices by permuting the order of nodes, and the model should not be sensitive to such permutation.
- ***Spatial locality preservation***. CNN architectures assume the input data has spatial locality, i.e, close-by data points are more similar to each other. How to make our embedding-based similarity matrix preserve such spatial locality is important.
- ***Graph size invariance***. The CNN architecture requires fixed-length input. How to handle graphs with different sizes is another question to address.
- ***Multi-scale comparison.*** Finally, graphs naturally contain patterns of different scales that may be unknown in advance. The algorithm should be able to capture and leverage structural information and features of multiple granularities.

To tackle these challenges, we propose GraphSim, which addresses the graph similarity computation task in a novel way by direct usage of node-level embeddings via pairwise node-node similarity scores. We show that GraphSim can be combined with various node embedding approaches, improving performance on four graph similarity computation datasets under six out of eight settings. Finally, we show that GraphSim can learn interpretable similarity patterns that exist in the input graph pairs.

3.2 Related Work

Graph Representation Learning Over the years, there have been a great number of works dealing with the representation of nodes [8], and graphs [14]. Among the node embedding methods, neighbor aggregation based methods, e.g. GCN [7], GraphSAGE [8], GIN [10], etc., are permutation-invariant, and have gained a lot of attention.

Neural network based methods have been used in a broad range of graph applications, most of which are framed as node-level prediction tasks [72] or single graph classification [14]. In this work, we consider the task of graph similarity computation, which is under the general problem of graph matching [73].

Text and Graph Matching with Neural Networks Text matching has a long history with many successful applications [74]. Among various methods for text matching, promising results in matching sequences of word embeddings [40] inspire us to explore the potential of using node embeddings for the task of graph matching directly without graph-level representations. In contrast, neural network based graph matching remains largely unexplored, and most existing works still rely on first generating one embedding per graph using graph neural networks, and then modeling the graph-graph similarity using the two graph-level representations.

We examine several existing works on similarity computation for graphs: (1) SIAMESE MPNN (SMPNN) [68] is an early work that models the similarity as a simple summation of certain node-node similarity scores. (2) GCNMEAN and GCNMAX [69] apply the GCN architectures with graph coarsening [6] to generate graph-level embeddings for the similarity. (3) SIMGNN [19] attempts to use node-node similarity scores by taking their histogram features, but still largely relies on the graph-level embeddings due to the histogram function being non-differentiable. (4) GMN [70] is a recent work which manages to introduce node-node similarity information into graph-level embeddings via a cross-graph attention mechanism, but the cross-graph communication only updates the node embeddings, and still generates one embedding per graph from the updated node embeddings.

3.3 Problem Definition

Graphs are data structures with a node set $\mathcal{V}$ and a edge set $\mathcal{E}$, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The number of nodes of $\mathcal{V}$ is denoted as $N = |\mathcal{V}|$. Each node and edge can be associated with labels, such as atom and chemical bond type in a molecular graph. In this

study, we confine our graphs as undirected and unweighted graphs, but it is not hard to extend GraphSim to other types of graphs, since GraphSim is a general framework for graph similarity computation.

Given two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, different distance/similarity metrics can be defined.

Graph Edit Distance (GED) The edit distance between two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is the number of edit operations in the optimal alignments that transform $\mathcal{G}_1$ into $\mathcal{G}_2$, where an edit operation on a graph $\mathcal{G}$ is an insertion or deletion of a node/edge or relabelling of a node [1]. We transform GED into a similarity metric ranging between 0 and 1 using a one-to-one mapping function.

Maximum Common Subgraph (MCS) A Maximum Common Subgraph of two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a subgraph common to both $\mathcal{G}_1$ and $\mathcal{G}_2$ such that there is no other subgraph of $\mathcal{G}_1$ and $\mathcal{G}_2$ with more nodes. The MCS definitions have two variants [75]: In one definition, the MCS must be a connected graph; in the other, the MCS can be disconnected. In this chapter, we adopt the former definition. For $\mathcal{G}_1$ and $\mathcal{G}_2$, their similarity score is defined as the number of nodes in their MCS, i.e. $|\text{MCS}(\mathcal{G}_1, \mathcal{G}_2)|$.

Learning-based Graph Similarity Computation Given a graph similarity definition, our goal is to learn a neural network-based function that takes two graphs as input and outputs the desired similarity score through training, which can be applied to any unseen graphs for similarity computation at the test stage.

In later sections, we will introduce how to design a neural network architecture to serve this purpose, and why such design is reasonable by providing connections to set matching.

3.4 The Proposed Approach: GraphSim

GraphSim consists of the following sequential stages: 1) *Multi-scale neighbor aggregation layers* generate vector representations for each node in the two graphs at different scales;

[1] Other variants of GED definitions exist [37], and we adopt this basic version for each in this work.

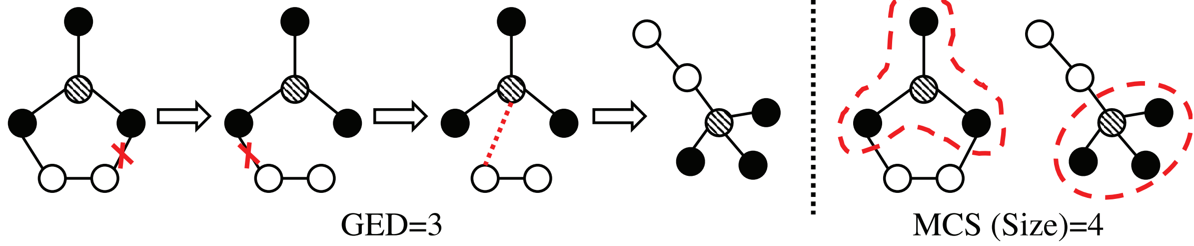


Figure 3.1: The GED is 3, as the transformation needs 3 edit operations: Two edge deletions, and an edge insertion. The MCS size is 4.

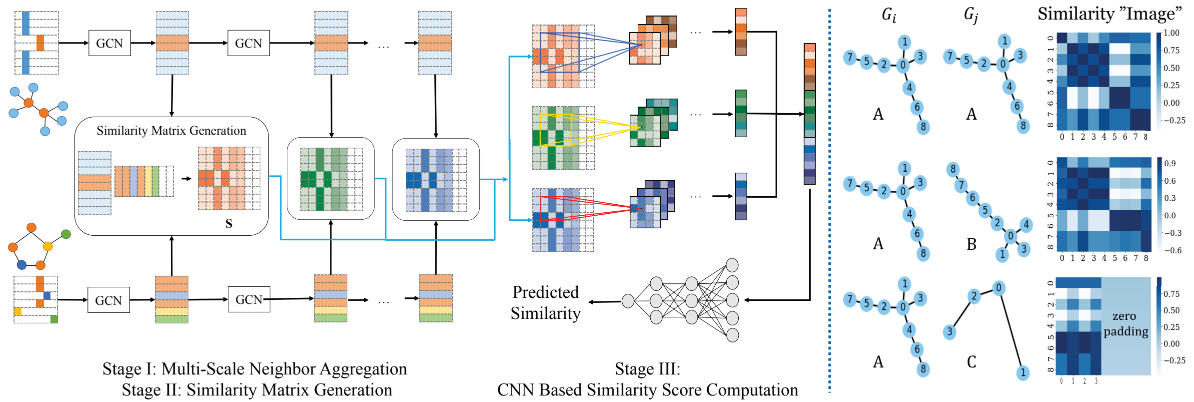


Figure 3.2: Left: An overview illustration of our proposed method GraphSim. No graph-level representation is generated, and it directly uses the node-node similarity scores in the three similarity matrices corresponding to node embeddings at different scales. Right: Illustration of three similarity matrices from the LINUX [1] dataset. For two isomorphic graphs (A and A), a strong symmetric diagonal block pattern is observed; for two less similar graphs (A and B), the dark diagonal pattern is less evident; for two graphs that are not similar at all (A and C), the symmetric block pattern is almost gone. For graphs of different sizes, we devise a consistent max padding scheme. Intuitively, the block patterns can be thought of as graph-graph similarity patterns at different scales, which are suitable for CNNs to capture.

2) *Similarity matrix generation layers* compute the inner products between the embeddings of every pair of nodes in the two graphs, resulting in multiple similarity matrices capturing the node-node interaction scores at different scales; 3) *CNN layers* convert the similarity computation problem into a pattern recognition problem, which provides multi-scale features to a *fully connected network* to obtain a final predicted graph-graph similarity score. An overview of our model is illustrated in Fig. 7.1.

3.4.1 Multi-Scale Neighbor Aggregation

We build upon an active line of research on graph neural networks for generating node representations. Graph Convolutional Networks (GCN) [6, 7], for example, is a neighbor aggregation approach which generates node embeddings from the local substructure information of each node, which is an inductive method and can be applied to unseen nodes. In Fig. 7.1, different node types are represented by different colors and one-hot encoded as the initial node representation. For graphs with unlabeled nodes, we use the same constant vector as the initial representation.

The core operation, graph convolution, operates on the representation of a node, which is denoted as $\boldsymbol{u_i} \in \mathbb{R}^D$, and is defined as follows:

$$\text{conv}(\boldsymbol{u_i}) = \text{ReLU}(\sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{d_i d_j}} \boldsymbol{u}_j \boldsymbol{W}^{(n)} + \boldsymbol{b}^{(n)}) \tag{3.1}$$

where $\mathcal{N}(i)$ is the set of the first-order neighbors of node i plus i itself, d_i is the degree of node i plus 1, $\boldsymbol{W}^{(n)} \in \mathbb{R}^{D^{(n)} \times D^{(n+1)}}$ is the weight matrix of the n-th GCN layer, $\boldsymbol{b}^{(n)} \in \mathbb{R}^{D^{(n+1)}}$ is the bias, $D^{(n)}$ denotes the dimensionality of embedding vector at layer n, and $\text{ReLU}(x) = \max(0, x)$ is the activation function.

Intuitively, the graph convolution operation aggregates the features from the first-order neighbors of the node. Since applying the GCN layer once on a node can be regarded as aggregating the representations of its first-order neighbors and itself, sequentially stacking L layers would cause the final representation of a node to include its L-th order neighbors. In other words, the more GCN layers, the larger the scale of the learned embeddings.

Multi-Scale GCN The potential issue of using a deep GCN architecture is that the embeddings may lose subtle patterns in local neighborhood after aggregating neighbors multiple times. The issue is especially severe when the two graphs are very similar, and the differences mainly lie in small local substructures.

One natural way for humans to compare the difference between two graphs is to recur-

sively break down the whole graph into its compositional subgraphs via a top-down approach. Each subgraph is further decomposed into additional subgraph levels, until the entire specification is reduced to node level, producing a hierarchy of subgraph (de)composition. We therefore propose a multi-scale framework to extract the output of each of the many GCN layers for the construction of similarity matrices, which is shown next.

GraphSim is a general framework for similarity computation, and can work with GCN and any of its successors such as GraphSAGE [8], GIN [10], etc.

3.4.2 Similarity Matrix Generation

The use of node-node similarity scores roots in some classic approaches to modeling graph similarity, such as the Optimal Assignment Kernels for graph classification and the Bipartite Graph Matching for GED approximation. Here we present some key properties of these methods that are needed to motivate the usage of node-node similarity scores.

Optimal Assignment Kernels Given two graphs with node embeddings $\boldsymbol{X} \in \mathbb{R}^{N_1 \times D}$ and $\boldsymbol{Y} \in \mathbb{R}^{N_2 \times D}$, the Earth Mover's Distance graph kernel [64] solves the following transportation problem [76]:

$$\min \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \boldsymbol{T}_{ij} ||\boldsymbol{x}_i - \boldsymbol{y}_j||_2 \tag{3.2}$$

subject to several constraints. Intuitively, each of the two graphs is represented as a set of node embeddings, and the kernel finds the optimal way to transform one set of node embeddings to the other, with the cost to transform one pair of nodes being their Euclidean distance.

Bipartite Graph Matching The Hungarian [29] and VJ [30] algorithms for approximate GED computation solve the following assignment problem form:

$$\min \sum_{i=1}^{N'} \sum_{j=1}^{N'} \boldsymbol{T}_{ij} \boldsymbol{C}_{ij} \tag{3.3}$$

subject to several constraints with cost matrix $\boldsymbol{C}$ denoting the insertion, deletion, and sub-

stitution costs associated with the GED metric for every node pair in the two graphs.

Despite different goals, both the kernel method and GED approximate algorithms work directly on node-level information without the whole-graph representations. Specifically, both need the pairwise node-node distance scores between the two graphs, with different definitions of node-node distances.

Since GraphSim is trained end-to-end for graph similarity computation, we can calculate the inner products between all pairs of node embeddings in the two graphs at multiple scales, resulting in multiple similarity matrices. Treat each matrix as an image, the task of graph similarity measurement is viewed as an image processing problem in which the goal is to discover the optimal node matching pattern encoded in the image by applying CNNs. Consider this process as a similarity operator that transforms two sets of node embeddings into a score. Then GraphSim can be regarded as:

$$\min(h_{\boldsymbol{\Theta}}(\boldsymbol{X}, \boldsymbol{Y}) - s_{ij})^2 \tag{3.4}$$

where $h_{\Theta}(\boldsymbol{X}, \boldsymbol{Y})$ denotes the similarity matrix generation and its subsequent layers. The training is guided by the true similarity score s_{ij} for the update of weights $\boldsymbol{\Theta}$ associated with the neighbor aggregation layers that generate node embeddings $\boldsymbol{X}$ and $\boldsymbol{Y}$ as well as the subsequent CNNs. In contrast, for the optimization problems (3.2) and (3.3), in order to find the optimal value as the computed graph distance, the problem must be solved explicitly [42, 43].

BFS Ordering Different from pixels of images or words of sentences, nodes of a graph typically lack ordering. A different node ordering would lead to a different similarity matrix. To completely solve the graph node permutation problem, we need to find one canonical ordering for each graph in a collection of graphs, which is NP-hard as shown in an early work on CNN for graphs [41]. Moreover, the CNNs require spatial locality preservation. To alleviate these two issues, we utilize the breadth-first-search (BFS) node-ordering scheme proposed in GraphRNN [77] to sort and reorder the node embeddings. Since BFS is per-

formed on the graph, nearby nodes are ordered close to each other. It is worth noting that the BFS ordering scheme achieves a reasonable trade-off between efficiency and uniqueness of ordering, as it requires quadratic operations in the worst case (i.e. complete graphs) [77].

Besides the issues related to node permutation and spatial locality, one must address the challenge posed by graph size variance and the fact that CNN architecture requires fix-length input. To preserve the information of graph size variance while fix the size of similarity matrix, we propose the solutions as follows:

Max Padding One can fix the number of nodes in each graph by adding fake nodes to a pre-defined number, and therefore lead to a fix-size similarity matrix. However, such matrix will completely ignore the graph size information, which is pivotal as seen in the definitions of both GED and MCS. For example, the similarity matrix between two small but isomorphic graphs may be padded with a lot of zeros, potentially misleading the CNNs to predict they are dissimilar.

To reflect the difference in graph sizes in the similarity matrix, we devise max padding. Suppose $\mathcal{G}_1$ and $\mathcal{G}_2$ contain N_1 and N_2 nodes respectively, we pad $|N_1 - N_2|$ rows of zeros to the node embedding matrix of the smaller of the two graphs, so that both graphs contain $\max(N_1, N_2)$ nodes.

Matrix Resizing To apply CNNs to the similarity matrices, We resize the simlarity matrices through image resampling [78]. For implementation, we choose the bilinear interpolation, and leave the exploration of more advanced techniques for future work. The resulting similarity matrix $\boldsymbol{S}$ has fixed shape $M \times M$, where M is a hyperparameter controlling the degree of information loss caused by resampling.

The following equation summarizes the similarity matrix generation process:

$$\boldsymbol{S} = \mathrm{RES}_M(\widetilde{\boldsymbol{H}}_1 \widetilde{\boldsymbol{H}}_2^T) \tag{3.5}$$

where $\widetilde{\boldsymbol{H}}_i \in \mathbb{R}^{\max(N_1,N_2)\times D}, i \in \{1,2\}$ is the padded node embedding matrix $\boldsymbol{H}_i \in \mathbb{R}^{N_i \times D}, i \in \{1,2\}$ with zero or $|N_1 - N_2|$ nodes padded, and $\mathrm{RES}(\cdot) : \mathbb{R}^{\max(N_1,N_2)\times\max(N_1,N_2)} \mapsto \mathbb{R}^{M\times M}$

Table 3.1: Effectiveness results on the GED metric. On Aids and Linux, A* provides ground-truth results, labeled with superscript ∗. On Imdb and Ptc, A* fails to compute most GEDs within a 5-minute limit (thus denoted as −). Instead, the minimum GED returned by Beam, Hungarian, and VJ for each pair is used as the ground-truth GED. The mse is in 10^{-3}.

	Aids			**Linux**			**Imdb**			**Ptc**		
Method	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**
A*	0.000*	1.000*	1.000*	0.000*	1.000*	1.000*	−	−	−	−	−	−
Beam	12.090	0.609	0.481	9.268	0.827	0.973	2.413*	0.905*	0.803*	0.552*	0.998*	0.982*
Hungarian	25.296	0.510	0.360	29.805	0.638	0.913	1.845*	0.932*	0.825*	112.326*	0.919*	0.159*
VJ	29.157	0.517	0.310	63.863	0.581	0.287	1.831*	0.934*	0.815*	154.791*	0.904*	0.102*
HED	28.925	0.621	0.386	19.553	0.897	0.982	19.400	0.751	0.801	978.318	0.919	0.169
Smpnn	5.184	0.294	0.032	11.737	0.036	0.009	32.596	0.107	0.023	116.473	0.148	0.082
EmbAvg	3.642	0.601	0.176	18.274	0.165	0.071	71.789	0.229	0.233	32.601	0.393	0.173
GCNMean	3.352	0.653	0.186	8.458	0.419	0.141	68.823	0.402	0.200	6.525	0.546	0.150
GCNMax	3.602	0.628	0.195	6.403	0.633	0.437	50.878	0.449	0.425	7.501	0.506	0.152
SimGNN	1.189	0.843	0.421	1.509	0.939	0.942	1.264	0.878	0.759	0.850	0.944	0.507
GMN	1.886	0.751	0.401	1.027	0.933	0.833	4.422	0.725	0.604	1.613	0.672	0.262
GraphSim	**0.787**	**0.874**	**0.534**	**0.058**	**0.981**	**0.992**	**0.743**	**0.926**	**0.828**	**0.749**	**0.956**	**0.529**

is the resizing function, where M is a hyperparameter controlling the degree of information loss caused by resampling.

3.4.3 CNN Based Similarity Score Computation

We feed these matrices through multiple CNNs in parallel. As shown in Fig. 7.1, three CNN "channels" are used, each with its own CNN filters. At the end, the results are concatenated and fed into multiple fully connected layers, so that a final similarity score $\hat{s}_{ij}$ is generated for the graph pair $\mathcal{G}_i$ and $\mathcal{G}_j$. The mean square error loss function is used to train our model: $\mathcal{L} = \frac{1}{|\mathcal{D}|}\sum_{(i,j)\in\mathcal{D}}(\hat{s_{ij}} - s_{ij})^2$ where $\mathcal{D}$ is the set of training graph pairs, and s_{ij} is the true similarity score coming from any graph similarity metric. For GED, we apply a one-to-one mapping function to transform the true distance score into the true similarity score; for MCS, the normalized MCS size is treated as the true similarity score.

3.5 Experiments

We evaluate our model, GraphSim, against a number of state-of-the-art approaches to GED and MCS computation, with the major goals of addressing the following questions:

Q1 How accurate (effective) and fast (efficient) is GraphSim compared to the state-of-the-art approaches for graph similarity computation, including both approximate similarity computation algorithms and neural network based models?

Q2 How do the proposed ordering, resizing, and multi-scale comparison techniques help with the CNN-based GraphSim model?

Q3 Does GraphSim yield meaningful and interpretable similarity matrices/images on the input graph pairs?

Datasets To probe the ability of GraphSim to compute graph-graph similarities from graphs in different domains, we evaluate on four real graph datasets, AIDS, LINUX, IMDB, and PTC.

For each dataset, we split it into training, validation, and testing sets by 6:2:2, and report the averaged *Mean Squared Error (mse)*, *Spearman's Rank Correlation Coefficient* (ρ) [46], *Kendall's Rank Correlation Coefficient* (τ) [47], and *Precision at k (p@k)* to test the accuracy and ranking performance of each GED and MCS computation method.

Baseline Methods We consider both the state-of-the-art GED/MCS computation methods and baselines using neural networks. To ensure consistency, all neural network models use GCN for node embeddings except for GMN, and to demonstrate the flexibility of our framework, we show the performance improvement of GraphSim by replacing GCN with the more powerful GMN's node embedding methods in the supplementary material.

3.5.1 Effectiveness

As shown in Table 3.1 and 3.2, our model, GraphSim, consistently achieves the best results on all metrics across all the datasets with both the GED and MCS metrics. Specifically,

Table 3.2: Effectiveness results on the MCS metric. On all the datasets, Mcsplit provides ground-truth results, labeled with superscript $*$. The mse is in 10^{-3}.

	Aids			**Linux**			**Imdb**			**Ptc**		
Method	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**
Mcsplit	0.000*	1.000*	1.000*	0.000*	1.000*	1.000*	0.000*	1.000*	1.000*	0.000*	1.000*	1.000*
Smpnn	4.592	0.755	0.348	3.558	0.126	0.236	11.018	0.330	0.003	11.001	0.502	0.146
EmbAvg	6.466	0.701	0.236	2.663	0.427	0.343	17.853	0.524	0.166	23.018	0.556	0.302
GCNMean	5.956	0.776	0.316	2.706	0.439	0.368	9.316	0.753	0.364	9.166	0.712	0.337
GCNMax	5.525	0.782	0.328	2.466	0.543	0.440	8.234	0.796	0.435	7.905	0.752	0.385
SimGNN	3.433	0.822	0.374	0.729	0.859	0.850	1.153	0.938	0.705	3.781	0.782	0.279
GMN	**1.750**	**0.909**	**0.591**	0.598	0.906	0.830	0.590	0.941	0.795	3.835	0.841	**0.530**
GraphSim	2.402	0.858	0.505	**0.164**	**0.962**	**0.951**	**0.307**	**0.976**	**0.817**	**2.268**	**0.854**	0.504

GraphSim achieves the smallest error and the best ranking performance on the task of graph similarity computation under six out of eight settings. We repeated the running of our model 10 times on Aids, and the standard deviation of mse is $4.56 * 10^{-5}$. The Aids dataset is relatively small in terms of the average number of nodes per graph, potentially causing the CNN model to overfit.

3.5.2 Efficiency

In Fig. 3.3, the results are averaged across queries and in wall time. EmbAvg is the fastest method among all, but its performance is poor, since it simply takes the dot product between two graph-level embeddings (average of node embeddings) as the predicted similarity score. Beam and Hungarian run fast on Linux, but due to their higher time complexity, they scale poorly on the largest dataset, Imdb. The exact MCS solver Mcsplit is the state-of-the-art for MCS computation, and runs faster than the exact GED solver, A* on all datasets. However, in general, neural network based models are still much faster than these solvers, since they enjoy lower time complexity in general and additional benefits from parallelizability and acceleration provided by GPU.

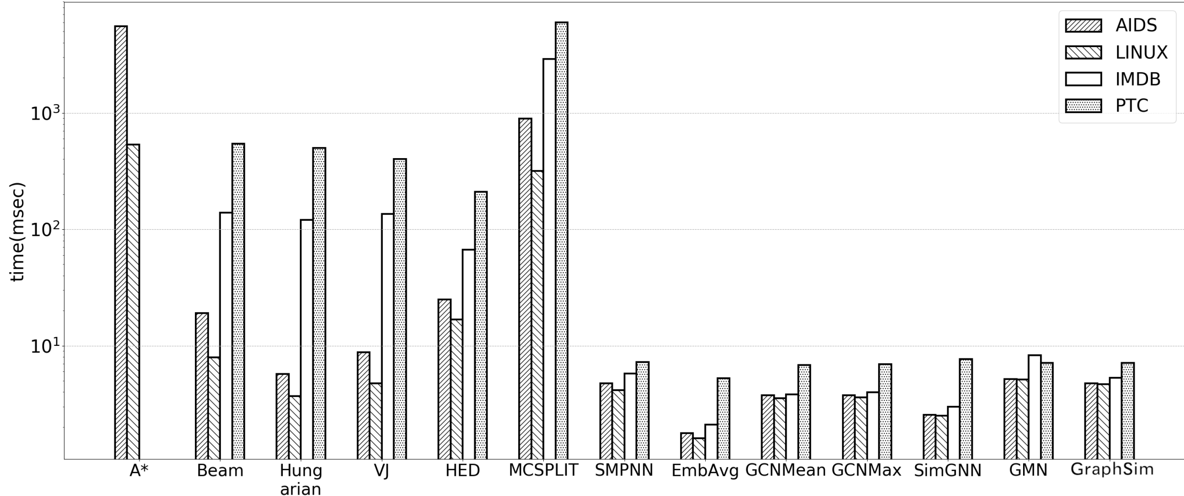


Figure 3.3: Running time comparison. The y-axis uses the log scale. The running time is averaged across queries. For neural network models, the running time for GED and MCS computation is very close to each other, so we take the average of the two.

3.5.3 Analysis of Various Proposed Techniques in GraphSim

To address **Q2**, we conduct several experiments by comparing GraphSim with three simpler variants, whose results are shown in Table 3.3. Among the three proposed techniques (i.e., max padding plus matrix resizing, ordering, and multi-scale comparison), max padding and matrix resizing affects the performance the most: Comparing GS-Pad with GS-Resize on Imdb and Ptc, our proposed padding and resizing technique greatly reduces the approximation error. Such significant improvements on Imdb and Ptc can be attributed to the large average graph size and graph size variance. Besides, by comparing the GS-Resize and GraphSim, we can see a performance boost brought by the multi-scale framework. The advantage of BFS ordering can be observed by comparison of GS-NoOrd and GraphSim.

3.5.4 Analysis of Similarity "Images" in GraphSim

We demonstrate six similarity matrices (plotted as heatmap images) generated by GraphSim on Aids in Fig. 3.4. Node ids come from BFS ordering. Since both pairs are quite similar, as mentioned in Section 3.4.3, we expect to see block patterns, which are actually observed in the six similarity matrices. From pair (1) (the top row), we can see that larger scale (the

Table 3.3: GS-PAD and GS-RESIZE perform node ordering and use the node embeddings only by the last GCN layer to generate the similarity matrix. Since CNNs require fixed-length input, if the model does not use resizing, a simple way is to zero pad each similarity matrix to N_{max} by N_{max} (maximum graph size in the entire dataset), denoted as GS-PAD. GS-RESIZE uses the proposed max padding and resizing techniques. GS-NOORD uses all the proposed techniques including multi-scale comparison except for node ordering. The results are on the GED metric. The mse is in 10^{-3}.

	Aids			**Linux**			**Imdb**			**Ptc**		
Method	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**	**mse**	ρ	**p@10**
GS-PAD	0.807	0.863	0.514	0.141	0.988	0.983	6.455	0.661	0.552	6.808	0.637	0.481
GS-RESIZE	0.811	0.866	0.499	0.103	0.991	0.986	0.896	0.914	0.810	0.791	0.948	0.513
GS-NOORD	0.803	0.867	0.478	0.071	0.983	0.976	1.105	0.902	0.744	0.838	0.945	0.515
GraphSim	**0.787**	**0.874**	**0.534**	**0.058**	**0.993**	**0.981**	**0.743**	**0.926**	**0.828**	**0.749**	**0.956**	**0.529**

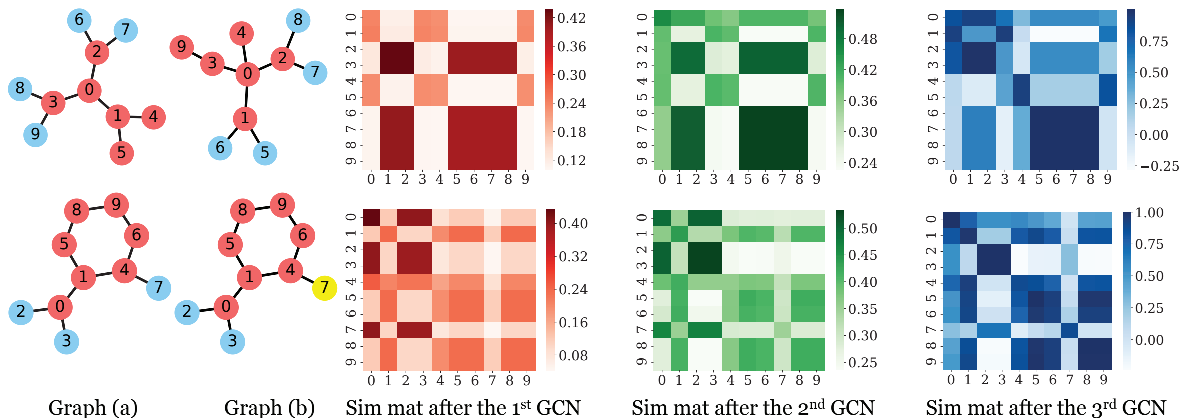


Figure 3.4: Similarity "images" on two pairs of graphs from AIDS trained with the GED metric. Different heatmap colors differentiate "images" from different scales of comparison. Top pair: GED=2—an edge deletion (edge between node 0 and 4 of graph (b)) and an edge addition (edge between node 3 and 4 of graph (b)). Bottom pair: GED=1—a node relabeling (node 7).

blue matrix) helps distinguish between nodes 3 and 4 of graph (b), which contributes to the GED edit sequence as shown in the caption of Fig 3.4. From pair (2) (the bottom row), we can see that the smaller scale (the red matrix) helps differentiate between node 7 in graph (a) and node 7 in graph (b). Notice that comparing at larger scales does not help tell their difference in node types, because their structural equivalence causes their similarities to be higher as seen in the green and blue matrices. Thus, Fig. 3.4 shows the importance of using multi-scale similarity matrices rather than a single one.

3.5.5 Case Studies

We demonstrate four example queries, one from each dataset in Fig. 3.5. In each demo, the top row depicts the query along with the ground-truth ranking results, labeled with their normalized GEDs to the query. The bottom row shows the graphs returned by our model, each with its rank shown at the top. GraphSim is able to retrieve graphs similar to the query. For example, in the case of LINUX, the top 6 results are exactly the isomorphic graphs to the query.

3.6 Conclusion

We introduced a CNN based method for better graph similarity computation. Using GraphSim in conjunction with existing methods for generating node embeddings, we improve the performance in several datasets on two graph proximity metrics: Graph Edit Distance (GED) and Maximum Common Subgraph (MCS). Interesting future directions include using hierarchical graph representation learning techniques to reduce computational time complexity involved in the node-node similarity computation, and applying the CNN based graph matching method to other graph matching tasks, e.g. network alignment, as well as the explicit generation of edit sequence for GED and node correspondence for MCS.

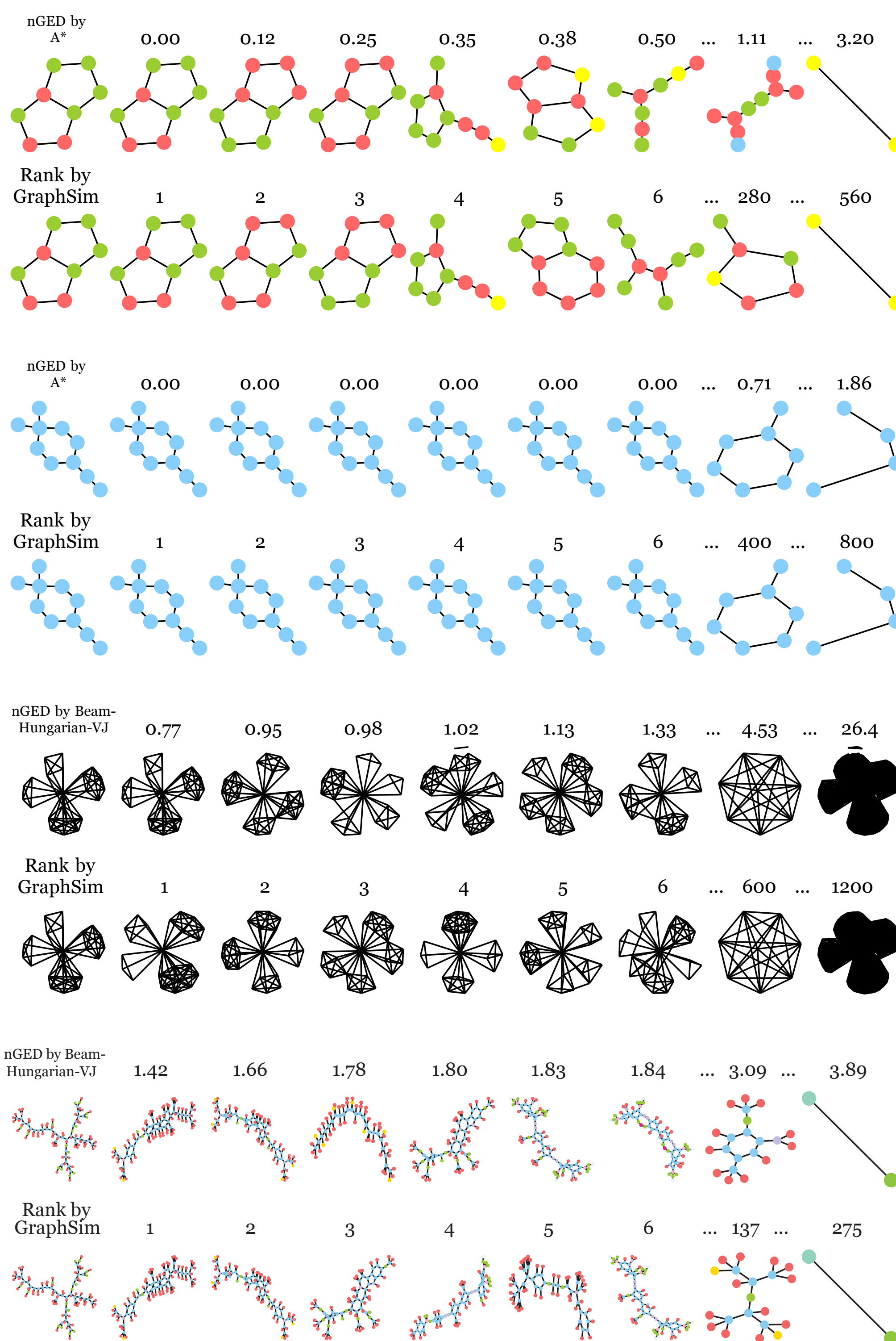


Figure 3.5: Visualization of ranking results under the GED metric. From top to bottom: AIDS, LINUX, IMDB, PTC.

CHAPTER 4

Unsupervised Inductive Graph-Level Representation Learning via Graph-Graph Proximity

4.1 Introduction

Recent years we have witnessed the great popularity of graph representation learning with success in not only node-level tasks such as node classification [7] and link prediction [79], but also graph-level tasks such as graph classification [14] and graph similarity/distance computation [19].

There has been a rich body of work [51, 80, 81] on node-level embeddings that turn each node in a graph into a vector preserving node-node proximity (similarity/distance). Most of these models are unsupervised and demonstrate superb performance in node classification and link prediction. It is natural to raise the question: Can we embed an entire graph into a vector in an unsupervised way, and how? However, most existing methods for graph-level embeddings assume a supervised model [14, 15], with only a few exceptions, such as Graph Kernels [13] and Graph2Vec [82]. Graph Kernels typically count subgraphs for a given graph and can be slow. Graph2Vec is transductive (non-inductive), i.e. it does not naturally generalize to unseen graphs outside the training set.

A key challenge facing designing an unsupervised graph-level embedding model is the lack of graph-level signals in the training stage. *Unlike node-level embedding which has a long history in utilizing the link structure of a graph to embed nodes, there lacks such natural proximity (similarity/distance) information between graphs.* Supervised methods, therefore, typically resort to graph labels as guidance and use aggregation based methods, e.g. average

of node embeddings, to generate graph-level embeddings, with the implicit assumption that good node-level embeddings would automatically lead to good graph-level embeddings using only "***intra*-graph** information" such as node attributes, link structure, etc.

However, this assumption is problematic, as simple aggregation of node embeddings can only preserve limited graph-level properties, which is, however, often insufficient in measuring graph-graph proximity ("***inter*-graph**" information). Inspired by the recent progress on graph proximity modeling [19, 69], we propose a novel framework, ***UGraphEmb*** (*U*nsupervised *Graph*-level *Emb*bedding) that employs multi-scale aggregations of node-level embeddings, guided by the graph-graph proximity defined by well-accepted and domain-agnostic graph proximity metrics such as Graph Edit Distance (GED) [17], Maximum Common Subgraph (MCS) [23], etc.

The goal of UGraphEmb is to learn high-quality ***graph-level*** representations in a completely ***unsupervised*** and ***inductive*** fashion: During training, it learns a function that maps a graph into a universal embedding space best preserving graph-graph proximity, so that after training, any new graph can be mapped to this embedding space by applying the learned function. Inspired by the recent success of pre-training methods in the text domain, such as ELMO [83], BERT [84], and GPT [85]. we further fine-tune the model via incorporating a supervised loss, to obtain better performance in downstream tasks, including but not limited to:

- **Graph classification**. The embeddings can be fed into any classification model such as logistic regression for graph classification.
- **Graph similarity ranking**. The embeddings learned by UGraphEmb preserve the graph-graph proximity by design, and for a graph query, a ranked list of graphs that are similar to the query can be retrieved.
- **Graph visualization**. The embeddings can be projected into a 2-D space for graph visualization, where each graph is a point. It renders human insights into the dataset and facilitates further database analysis.

In summary, our contributions are three-fold:

1. We formulate the problem of unsupervised inductive graph-level representation learning, and make an initial step towards pre-training methods for graph data. We believe that, given the growing amount of graph datasets of better quality, this work would benefit future works in pre-training methods for graphs.

2. We provide a novel framework, UGraphEmb, to generate ***graph-level*** embeddings in a completely ***unsupervised*** and ***inductive*** fashion, well preserving graph proximity. A novel **Multi-Scale Node Attention (MSNA)** mechanism is proposed to generate graph-level embeddings.

3. We conduct extensive experiments on five real network datasets to demonstrate the superb quality of the embeddings by UGraphEmb.

4.2 The Proposed Approach: UGraphEmb

We present the overall architecture of our unsupervised inductive graph-level embedding framework UGraphEmb in Figure 7.1. The key novelty of UGraphEmb is the use of graph-graph proximity. To preserve the proximity between two graphs, UGraphEmb generates one embedding per graph from node embeddings using a novel mechanism called **Multi-Scale Node Attention (MSNA)**, and computes the proximity using the two graph-level embeddings.

4.2.1 Inductive Graph-Level Embedding

4.2.1.1 Node Embedding Generation

For each graph, UGraphEmb first generates a set of node embeddings. There are two major properties that the node embedding model has to satisfy:

- **Inductivity**. The model should learn a function such that for any new graph unseen

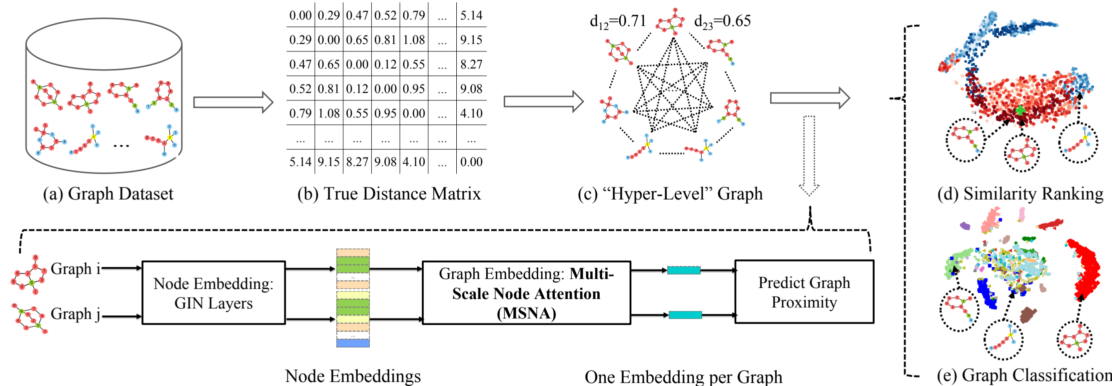


Figure 4.1: Overview of UGraphEmb. (a) Given a set of graphs, (b) UGraphEmb first computes the graph-graph proximity scores (normalized distance scores in this example), (c) yielding a "hyper-level graph" where each node is a graph in the dataset, and each edge has a proximity score associated with it, representing its weight/strength. UGraphEmb then trains a function that maps each graph into an embedding which preserves the proximity score. The bottom flow illustrates the details of graph-level embedding generation. (d) After embeddings are generated, similarity ranking can be performed. The green "+" sign denotes the embedding of an example query graph. Colors of dots indicate how similar a graph is to the query based on the ground truth (from red to blue, meaning from the most similar to the least similar). (e) Finally, UGraphEmb can perform fine-tuning on the proximity-preserving graph-level embeddings, adjusting them for the task of graph classification. Different colors represent different graph labels in the classification task.

in the training set, the learned function can be applied to the graph, yielding its node embeddings.

- **Permutation-invariance**. The same graph can be represented by different adjacency matrices by permuting the order of nodes, and the node embedding model should not be sensitive to such permutation.

Among various node embedding models, neighbor aggregation methods based on Graph Convolutional Networks (GCN) [7] are permutation-invariant and inductive. The reason is that the core operation, graph convolution, updates the representation of a node by aggregating the embedding of itself and the embeddings of its neighbors. Since the aggregation function treats the neighbors of a node as a set, the order does not affect the result.

A series of neighbor aggregation methods have been proposed with different ways to aggregate neighbor information, e.g. GraphSAGE [8], GAT [9], GIN [10], etc. Since UGraphEmb is a general framework for *graph*-level embeddings, and all these models satisfy

the two properties, any one of these methods can be integrated. We therefore adopt the most recent and state-of-the-art method, Graph Isomorphism Network (GIN) [10], in our framework:

$$\boldsymbol{u_i}^{(k)} = \text{MLP}^{(k)}_{\boldsymbol{W_k}} \left(\left(1 + \epsilon^{(k)}\right) \cdot \boldsymbol{u_i}^{(k-1)} + \sum_{j \in \mathcal{N}(i)} \boldsymbol{u_j}^{(k-1)} \right) \tag{4.1}$$

where $\boldsymbol{u_i}$ is the representation of node i, $\mathcal{N}(i)$ is the set of neighbors of node i, $\text{MLP}^{(k)}_{\boldsymbol{W_k}}$ denotes multi-layer perceptrons for the k-th GIN layer with learnable weights $\boldsymbol{W_k}$, and ϵ is a scalar that can either be learned by gradient descent or be a hyperparameter. GIN has been proven to be theoretically the most powerful GNN under the neighbor aggregation framework [10].

4.2.1.2 Graph Embedding Generation

After node embeddings are generated, UGraphEmb generates one embedding per graph using these node embeddings. Existing methods are typically based on aggregating node embeddings, by either a simple sum or average, or some more sophisticated way to aggregate.

However, since our goal is to embed each graph as a single point in the embedding space that preserves graph-graph proximity, the graph embedding generation model should:

- **Capture structural difference at multiple scales**. Applying a neighbor aggregation layer on nodes such as GIN cause the information to flow from a node to its direct neighbors, so sequentially stacking K layers would cause the final representation of a node to include information from its K-th order neighbors. However, after many neighbor aggregation layers, the learned embeddings could be too coarse to capture the structural difference in small local regions between two similar graphs. Capturing structural difference at multiple scales is therefore important for UGraphEmb to generate high-quality graph-level embeddings.

- **Be adaptive to different graph proximity metrics**.

UGraphEmb is a general framework that should be able to preserve the graph-graph proximity under any graph proximity metric, such as GED and MCS. A simple aggregation of node embeddings without any learnable parameters limits the expressive power of existing graph-level embedding models.

To tackle both challenges in the graph embedding generation layer, we propose the following **Multi-Scale Node Attention (MSNA)** mechanism. Denote the input node embeddings of graph $\mathcal{G}$ as $\boldsymbol{U_{\mathcal{G}}} \in \mathbb{R}^{N \times D}$, where the n-th row, $\boldsymbol{u_n} \in \mathbb{R}^D$ is the embedding of node n. The graph level embedding is obtained as follows:

$$\boldsymbol{h_{\mathcal{G}}} = \mathrm{MLP}_{\boldsymbol{W}} \left(\Big\Vert_{k=1}^{K} \mathrm{ATT}_{\boldsymbol{\Theta}^{(k)}}(\boldsymbol{U_{\mathcal{G}}}) \right) \tag{4.2}$$

where $\Vert$ denotes concatenation, K denotes the number of neighbor aggregation layers, ATT denotes the following multi-head attention mechanism that transforms node embeddings into a graph-level embedding, and $\mathrm{MLP}_{\boldsymbol{W}}$ denotes multi-layer perceptrons with learnable weights $\boldsymbol{W}$ applied on the concatenated attention results.

The intuition behind Equation 4.2 is that, instead of only using the node embeddings generated by the last neighbor aggregation layer, we use the node embeddings generated by each of the K neighbor aggregation layers.

ATT is defined as follows:

$$\mathrm{ATT}_{\boldsymbol{\Theta}}(\boldsymbol{U_{\mathcal{G}}}) = \sum_{n=1}^{N} \sigma(\boldsymbol{u_n}^T \mathrm{ReLU}(\boldsymbol{\Theta}(\frac{1}{N}\sum_{m=1}^{N} \boldsymbol{u_m})))\boldsymbol{u_n}. \tag{4.3}$$

where N is the number of nodes, σ is the sigmoid function $\sigma(x) = \frac{1}{1+\exp{(-x)}}$, and $\boldsymbol{\Theta^{(k)}} \in \mathbb{R}^{D \times D}$ is the weight parameters for the k-th node embedding layer.

The intuition behind Equation 4.3 is that, during the generation of graph-level embeddings, the attention weight assigned to each node should be adaptive to the graph proximity metric. To achieve that, the weight is determined by both the node embedding $\boldsymbol{u_n}$, and a learnable graph representation. The learnable graph representation is adaptive to a partic-

ular graph proximity via the learnable weight matrix $\boldsymbol{\Theta}^{(k)}$.

4.2.2 Unsupervised Loss via Inter-Graph Proximity Preservation

4.2.2.1 Definition of Graph Proximity

The key novelty of UGraphEmb is the use of graph-graph proximity. It is important to select an appropriate graph proximity (similarity/distance) metric. We identify three categories of candidates:

- **Proximity defined by graph labels**.

  For graphs that come with labels, we may treat graphs of the same label to be similar to each other. However, such proximity metric may be too coarse, unable to distinguish between graphs of the same label.

- **Proximity given by domain knowledge or human experts**.

  For example, in drug-drug interaction detection [60], a domain-specific metric to encode compound chemical structure can be used to compute the similarities between chemical graphs. However, such metrics do not generalize to graphs in other domains. Sometimes, this information may be very expensive to obtain. For example, to measure brain network similarities, a domain-specific preprocessing pipeline involving skull striping, band-pass filtering, etc. is needed. The final dataset only contains networks from 871 humans [69].

- **Proximity defined by domain-agnostic and well-accepted metrics**.

  Metrics such as Graph Edit Distance (GED) [17] and Maximum Common Subgraph (MCS) [23] have been widely adopted in graph database search [27, 86], are well-defined and general to any domain.

In this chapter, we use GED as an example metric to demonstrate UGraphEmb. GED measures the minimum number of edit operations to transform one graph to the other, where

an edit operation on a graph is an insertion or deletion of a node/edge or relabelling of a node. Thus, the GED metric takes both the graph structure and the node labels/attributes into account. The supplementary material contain more details on GED.

4.2.2.2 Prediction of Graph Proximity

Once the proximity metric is defined, and the graph-level embeddings for $\mathcal{G}_i$ and $\mathcal{G}_j$ are obtained, denoted as $\boldsymbol{h}_{\mathcal{G}_i}$ and $\boldsymbol{h}_{\mathcal{G}_j}$, we can compute the similarity/distance between the two graphs.

Multidimensional scaling (MDS) is a classic form of dimensionality reduction [87]. The idea is to embed data points in a low dimensional space so that their pairwise distances are preserved, e.g. via minimizing the loss function

$$\mathcal{L}(\boldsymbol{h_i}, \boldsymbol{h_j}, d_{ij}) = (||\boldsymbol{h_i} - \boldsymbol{h_j}||_2^2 - d_{ij})^2 \tag{4.4}$$

where $\boldsymbol{h_i}$ and $\boldsymbol{h_j}$ are the embeddings of points i and j, and d_{ij} is their distance.

Since GED is a well-defined graph distance metric, we can minimize the difference between the predicted distance and the ground-truth distance:

$$\mathcal{L} = \mathbb{E}_{(i,j)\sim\mathcal{D}}(\hat{d_{ij}} - d_{ij})^2 = \mathbb{E}_{(i,j)\sim\mathcal{D}}(||\boldsymbol{h_{\mathcal{G}_i}} - \boldsymbol{h_{\mathcal{G}_j}}||_2^2 - d_{ij})^2. \tag{4.5}$$

where (i, j) is a graph pair sampled from the training set and d_{ij} is the GED between them.

Alternatively, if the metric is similarity, such as in the case of MCS, we can use the following loss function:

$$\mathcal{L} = \mathbb{E}_{(i,j)\sim\mathcal{D}}(\hat{s_{ij}} - s_{ij})^2 = \mathbb{E}_{(i,j)\sim\mathcal{D}}(\boldsymbol{h_{\mathcal{G}_i}^T} \boldsymbol{h_{\mathcal{G}_j}} - s_{ij})^2. \tag{4.6}$$

After training, the learned neural network can be applied to any graph, and the graph-level embeddings can facilitate a series of downstream tasks, and can be fine-tuned for specific

tasks. For example, for graph classification, a supervised loss function can be used to further enhance the performance.

4.3 Experiments

We evaluate our model, UGraphEmb, against a number of state-of-the-art approaches designed for unsupervised node and graph embeddings, to answer the following questions:

Q1 How superb are the graph-level embeddings generated by UGraphEmb, when evaluated with downstream tasks including graph classification and similarity ranking?

Q2 Do the graph-level embeddings generated by UGraphEmb provide meaningful visualization for the graphs in a graph database?

Q3 Is the quality of the embeddings generated by UGraphEmb sensitive to choices of hyperparamters?

Datasets We evaluate the methods on five real graph datasets, PTC, IMDB, WEB, NCI109, and REDDIT. The largest dataset, REDDIT, has 11929 graphs.

4.3.1 Task 1: Graph Classification

Intuitively, the higher the quality of the embeddings, the better the classification accuracy. Thus, we feed the graph-level embeddings generated by UGraphEmb and the baselines into a logistic regression classifier to evaluate the quality: (1) GRAPH KERNELS (GRAPHLET (GK), DEEP GRAPHLET (DGK), SHORTEST PATH (SP), DEEP SHORTEST PATH (DSP), WEISFEILER-LEHMAN (WL), and DEEP WEISFEILER-LEHMAN (DWL)) ; (2) GRAPH2VEC [82]; (3) NETMF [81]; (4) GRAPHSAGE [8].

For GRAPH KERNELS, we also try using the kernel matrix and SVM classifier as it is the standard procedure outlined in [13], and report the better accuracy of the two. For (3) and (4), we try different averaging schemes on node embeddings to obtain the graph-level embeddings and report their best accuracy.

Method	Ptc	Imdb	Web	Nci109	Reddit
GK	57.26	43.89	21.37	62.06	31.82
DGK	57.32	44.55	23.65	62.69	32.22
SP	58.24	37.01	18.16	73.00	–
DSP	60.08	39.67	22.65	73.26	–
WL	66.97	49.33	26.44	**80.22**	39.03
DWL	70.17	49.95	34.56	**80.32**	39.21
Graph2Vec	60.17	47.33	**40.91**	74.26	35.24
NetMF	56.65	30.67	19.71	51.84	23.24
GraphSAGE	52.17	34.67	20.38	65.09	25.01
UGraphEmb	**72.54**	**50.06**	37.36	69.17	**39.97**
UGraphEmb-F	**73.56**	**50.97**	**45.03**	74.48	**41.84**

Table 4.1: Graph classification accuracy in percent. "-" indicates that the computation did not finish after 72 hours. We highlight the top 2 accuracy in bold.

As shown in Table 4.1, UGraphEmb without fine-tuning, i.e. using only the unsupervised "inter-graph" information, can already achieve top 2 on 3 out of 5 datasets and demonstrates competitive accuracy on the other datasets. With fine-tuning (UGraphEmb-F), our model can achieve the best result on 4 out of 5 datasets. Methods specifically designed for graph-level embeddings (Graph Kernels, Graph2Vec, and UGraphEmb) consistently outperform methods designed for node-level embeddings (NetMF and GraphSAGE), suggesting that *good node-level embeddings do* ***not*** *naturally imply good graph-level representations.*

4.3.2 Task 2: Similarity Ranking

For each dataset, we split it into training, validation, and testing sets by 6:2:2, and report the averaged *Mean Squared Error (mse)*, *Kendall's Rank Correlation Coefficient (τ)* [47], and *Precision at 10 (p@10)* to test the ranking performance.

Table 4.2 shows that UGraphEmb achieves state-of-the-art ranking performance under all settings except one. This should not be a surprise, because only UGraphEmb utilizes the ground-truth GED results collectively determined by Beam [28], Hungarian [29], and VJ [30]. UGraphEmb even outperforms HED [88], a state-of-the-art approximate GED computation algorithm, under most settings, further confirming its strong ability to generate proximity-preserving graph embeddings by learning from a specific graph proximity metric, which is GED in this case.

Method	Ptc		Imdb		Web		Nci109		Reddit	
	τ	p@10	τ	p@10	τ	p@10	τ	p@10	τ	p@10
GK	0.291	0.135	0.329	0.421	0.147	0.101	0.445	0.012	0.007	0.009
DGK	0.222	0.103	0.304	0.410	0.126	0.009	0.441	0.010	0.011	0.012
SP	0.335	0.129	0.009	0.011	0.008	0.065	0.238	0.012	–	–
DSP	0.344	0.130	0.007	0.010	0.011	0.072	0.256	0.019	–	–
WL	0.129	0.074	0.034	0.038	0.014	0.246	0.042	0.006	0.089	0.017
DWL	0.131	0.072	0.039	0.041	0.017	0.262	0.049	0.009	0.095	0.023
Graph2Vec	0.128	0.188	0.697	0.624	0.014	0.068	0.033	0.127	0.008	0.017
NetMF	0.004	0.012	0.003	0.143	0.002	0.010	0.001	0.008	0.009	0.042
GraphSAGE	0.011	0.033	0.042	0.059	0.009	0.010	0.018	0.040	0.089	0.017
Beam	0.992*	0.983*	0.892*	0.968*	0.963*	0.957*	0.615*	0.997*	0.657*	1.000*
Hungarian	0.755*	0.465*	0.872*	0.825*	0.706*	0.160*	0.667*	0.164*	0.512*	0.808*
VJ	0.749*	0.403*	0.874*	0.815*	0.704*	0.151*	0.673*	0.097*	0.502*	0.867*
HED	0.788	0.433	0.627	0.801	**0.667**	0.291	0.199	0.174	0.199	0.237
UGraphEmb	**0.840**	**0.457**	**0.853**	**0.816**	0.618	**0.303**	**0.476**	**0.189**	**0.572**	**0.365**

Table 4.2: Similarity ranking performance. Beam, Hungarian, and VJ are three approximate GED computation algorithms returning upper bounds of exact GEDs. We take the minimum GED computed by the three as ground-truth GEDs for training and evaluating all the methods on both Task 1 and 2. Their results are labeled with "$*$". HED is another GED solver yielding lower bounds. "-" indicates that the computation did not finish after 72 hours.

4.3.3 Task 3: Embedding Visualization

Visualizing the embeddings on a two-dimensional space is a popular way to evaluate node embedding methods [51]. However, we are among the first to investigate the question: Are the graph-level embeddings generated by a model like UGraphEmb provide meaningful visualization?

We feed the graph emebddings learned by all the methods into the visualization tool t-SNE [89]. The three deep graph kernels, i.e. DGK, DSP, and WDL, generate the same embeddings as the non-deep versions, but use additional techniques [13] to modify the similarity kernel matrices, resulting in different classification and ranking performance.

From Figure 4.2, we can see that UGraphEmb can separate the graphs in Imdb into multiple clusters, where graphs in each cluster share some common substructures.

Such clustering effect is likely due to our use of graph-graph proximity scores, and is thus not observed in NetMF or GraphSAGE. For Graph Kernels and Graph2Vec though, there are indeed clustering effects, but by examining the actual graphs, we can see that graph-graph proximity is not well-preserved by their clusters (e.g. for WL graph 1, 2 and 9 should be close to each other; for Graph2Vec, graph 1, 2, and 12 should be close to each other), explaining their worse similarity ranking performance in Table 4.2 compared to UGraphEmb.

4.3.4 Parameter Sensitivity of UGraphEmb

We evaluate how the dimension of the graph-level embeddings and the percentage of graph pairs with ground-truth GEDs used to train the model can affect the results. We report the graph classification accuracy on Imdb.

As can be seen in Figure 4.3, the performance becomes marginally better if larger dimensions are used. For the percentage of training pairs with ground-truth GEDs, the performance drops as less pairs are used. Note that the x-axis is in log-scale. When we only use 0.001% of all the training graph pairs (only 8 pairs with ground-truth GEDs), the performance is

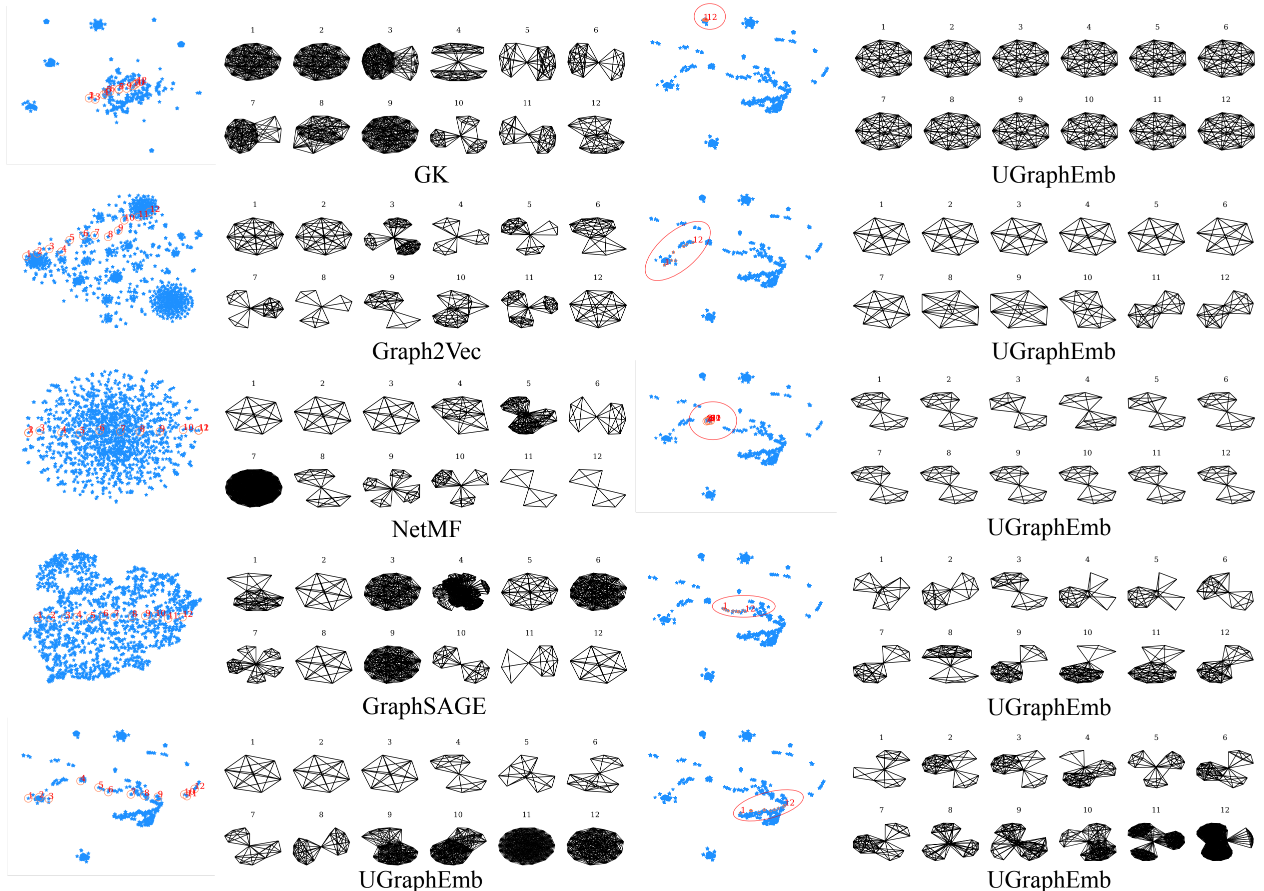


Figure 4.2: Visualization of the IMDB dataset. From (a) to (g), for each method, 12 graphs are plotted. For (h) to (l), we focus on UGraphEmb: 5 clusters are highlighted in red circles. 12 graphs are sampled from each cluster and plotted to the right.

still better than many baseline methods, exhibiting impressive insensitivity to data sparsity.

4.4 Related Work

Unsupervised graph representation learning has a long history. Classic works including NETMF [81], LINE [51], DeepWalk [49], etc., which typically generate an embedding for each node in ***one*** graph. Theoretical analysis shows that many of these works cannot handle embeddings for multiple graphs in the sense that the node embeddings in one graph are not comparable to those in another graph in any straightforward way [90]. A simple permutation of node indices could cause the node embedding to be very different.

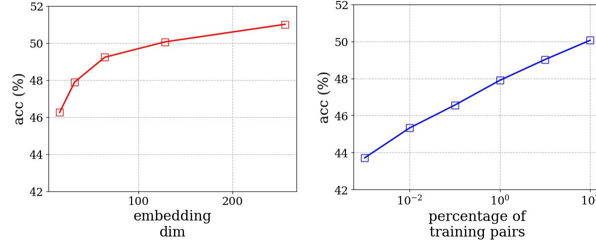


Figure 4.3: Classification accuracy on the IMDB dataset w.r.t. the dimension of graph-level embeddings and the percentage of graph pairs used for training.

More recently, some of the methods based on Graph Convolutional Networks (GCN) [6, 7], such as VGAE [91], satisfy the desired permutation-invariance property. Categorized as "graph autoencoders" [92], they also belong to the family of graph neural network methods. Although satisfying the permutation-invariance requirement, these autoencoders are still designed to generate unsuperised node embeddings.

Methods designed for unsupervised graph-level embeddings include GRAPH2VEC [82] and GRAPH KERNELS [13], which however are either not based on learning or not inductive. Unlike node-level information which is reflected in the neighborhood of a node, graph-level information is much more limited. A large amount of graph neural network models resort to graph labels as a source of such information, making the models supervised aiming to improve graph classification accuracy specifically, such as DIFFPOOL [14], CAPSGNN [15], etc., while UGraphEmb learns a function that maps each graph into an embedding that can be used to facilitate many downstream tasks.

4.5 Conclusion

We present UGraphEmb, an end-to-end neural network based framework aiming to embed an entire graph into an embedding preserving the proximity between graphs in the dataset under a graph proximity metric, such as Graph Edit Distance (GED). A novel mechanism for generating graph-level embeddings is proposed. Experiments show that the produced graph-level embeddings achieve competitive performance on three downstream tasks: graph classification, similarity ranking, and graph visualization.

CHAPTER 5

Learning to Search for Fast Maximum Common Subgraph Detection

5.1 Introduction

Due to the flexible and expressive nature of graphs, designing machine learning approaches to solve graph tasks is gaining increasing attention from researchers. Among various graph tasks detecting the largest subgraph that is commonly present in both input graphs, known as Maximum Common Subgraph (MCS) [23] (as shown in Figure 5.1), is an important yet particularly hard task. MCS naturally encodes the degree of similarity between two graphs, is domain-agnostic, and thus has occurred in many domains such as software analysis [93], graph database systems [86] and cloud computing platforms [94]. In drug design, the manual testing of the effects of a new drug is known to be a major bottleneck, and the identification of compounds that share common or similar subgraphs which tend to have similar properties can effectively reduce the manual labor [95].

MCS detection is NP-hard in its nature and is thus a very challenging task. On one hand, the state-of-the-art exact MCS detection algorithms based on branch and bound run in exponential time in worst cases [96]. What is worse, they rely on several heuristics on how to explore the search space. For example, McSp [97] uses node degree as its heuristic by choosing high-degree nodes to visit first, but in many cases the true MCS contains small-degree nodes. On the other hand, existing machine learning approaches to graph matching such as [98] and [99] either do not address the MCS detection task directly or rely on labeled data requiring the pre-computation of MCS results by running exact solvers.

In this chapter, we present GLSearch (Graph Learning to Search), a general framework for MCS detection combining the advantages of search and reinforcement learning. GLSearch learns to search by adopting a Deep Q-Network (DQN) [100] to replace the node selection heuristics required by state-of-the-art MCS solvers, leading to faster arrival of the optimal solution for an input graph pair, which is particularly useful when the simpler heuristics fail and graphs are large with a limited search budget. Thanks to the learning capacity of Graph Neural Networks (GNN), our DQN is specially designed for the MCS detection task with a novel reformulation of DQN to better capture the effect of different node selections. Given the large action space incurred by large graph pairs, to enhance the training of DQN, we leverage the search algorithm to not only provide supervised signals in a pre-training stage but also offer guidance during an imitation learning stage.

Algorithm 5.1: Branch and Bound for MCS

1: **Input:** Input graph pair $\mathcal{G}_1$, $\mathcal{G}_2$.
2: **Output:** $maxSol$.
3: Initialize stack ← `new` Stack().
4: Initialize $maxSol \leftarrow$ empty solution.
5: stack.$push(s_0)$;
6: **while** stack $\neq \emptyset$ **do**
7: $\quad s_t \leftarrow$ stack.$pop()$;
8: $\quad curSol \leftarrow s_t.getCurSol()$;
9: $\quad$ **if** $|curSol| > |maxSol|$ **then**
10: $\quad\quad maxSol \leftarrow curSol$;
11: $\quad$ **end if**
12: $\quad UB_t \leftarrow |curSol| +$ overestimate(s_t);
13: $\quad$ **if** $UB_t \leq |maxSol|$ **then**
14: $\quad\quad$ **continue**;
15: $\quad$ **end if**
16: $\quad \mathcal{A}_t \leftarrow s_t$.actions;
17: $\quad a_t \leftarrow policy(s_t, \mathcal{A}_t)$;
18: $\quad s_t$.actions $\leftarrow s_t$.actions $\setminus \{a_t\}$;
19: $\quad$ stack.$push(s_t)$;
20: $\quad s_{t+1} \leftarrow$ env.update$(s_t, \mathcal{A}_t)$;
21: $\quad$ stack.$push(s_{t+1})$;
22: **end while**

Experiments on real graph datasets that are significantly larger than exisitng datasets adopted by state-of-the-art MCS solvers demonstrate that GLSearch outperforms baseline solvers and machine learning models for graph matching in terms of effectiveness by a large margin. Our contributions can be summarized as follows:

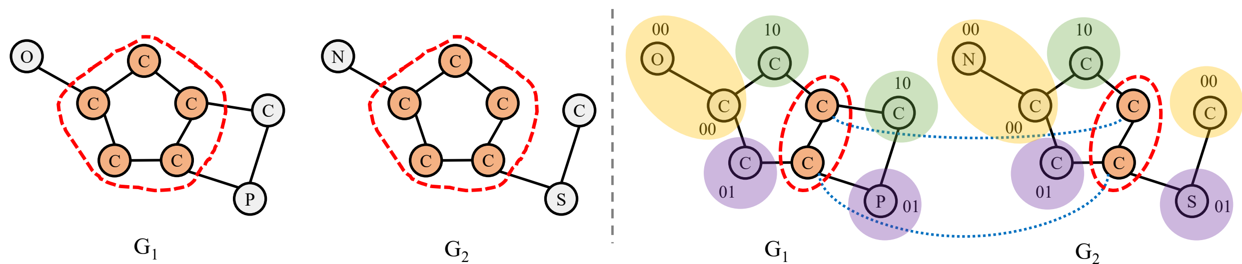


Figure 5.1: Left: For graph pair $(\mathcal{G}_1, \mathcal{G}_2)$ with node labels, the induced connected MCS is the five-member ring structure highlighted in circle. Right: At this step, there are two nodes currently selected. According to whether each node is connected to the two selected nodes or not, the nodes not in the current solution are split into three bidomains (Section 5.2.2), denoted as "00", "01", and "10", where "0" indicates not connected to a node in the selected two nodes, and "1" indicates connected. For example, each node in the "10" bidomain is connected to the top "C" node in the subgraph and disconnected to the bottom "C" node in the subgraph.

- We address the challenging yet important task of Maximum Common Subgraph detection for general-domain input graph pairs and propose GLSearch as the solution.
- The key novelty is the DQN which learns to search. Specifically, it is trained under the reinforcement learning framework to make the best decision at each search step in order to quickly find the best MCS solution during search. The search in turns helps training of DQN in a pre-training stage and an imitation learning stage.
- We conduct extensive experiments on medium-size synthetic graphs and very large real-world graphs to demonstrate the effectiveness of the proposed approach compared against a series of string baselines in MCS detection and graph matching.

5.2 Preliminaries

5.2.1 Problem Definition

We denote a graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ and $\mathcal{E}$ denote the vertex and edge set. An induced subgraph is defined as $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ where $\mathcal{E}_s$ preserves all the edges between nodes in $\mathcal{V}_s$, i.e. $\forall i, j \in \mathcal{V}_s$, $(i, j) \in \mathcal{E}_s$ if and only if $(i, j) \in \mathcal{E}$. In this chapter, we aim at detecting

the Maximum Common induced Subgraph (MCS) between an input graph pair, denoted as $\text{MCS}(\mathcal{G}_1, \mathcal{G}_2)$, which is the largest induced subgraph that is contained in both $\mathcal{G}_1$ and $\mathcal{G}_2$. In addition, we require $\text{MCS}(\mathcal{G}_1, \mathcal{G}_2)$ to be a connected subgraph. We allow the nodes of input graphs to be labeled, in which case the labels of nodes in the MCS must match as in Figure 5.1. Graph isomorphism and subgraph isomorphism can be regarded as two special tasks of MCS: $|\text{MCS}(\mathcal{G}_1, \mathcal{G}_2)| = |\mathcal{V}_1| = |\mathcal{V}_2|$ if $\mathcal{G}_1$ are isomorphic to $\mathcal{G}_2$, $|\text{MCS}(\mathcal{G}_1, \mathcal{G}_2)| = \min(|\mathcal{V}_1|, |\mathcal{V}_2|)$ when $\mathcal{G}_1$ (or $\mathcal{G}_2$) is subgraph isomorphic to $\mathcal{G}_2$ (or $\mathcal{G}_1$).

5.2.2 Search Algorithm for MCS

Among various algorithms for MCS, we adopt the state-of-the-art search-based algorithm in our framework. The basic version, McSp, is presented in [97] and the more advanced version, McSp+RL, is proposed in [96]. The whole search algorithm, outlined in Algorithm 5.1[1], is a branch-and-bound algorithm that maintains a best solution found so far throughout the search, which is initialized as empty subgraphs. In each search iteration, denote the current search state as s_t consisting of $\mathcal{G}_1$, $\mathcal{G}_2$, the current selected subgraphs $\mathcal{G}_{1s} = (\mathcal{V}_{1s}, \mathcal{E}_{1s})$ and $\mathcal{G}_{2s} = (\mathcal{V}_{2s}, \mathcal{E}_{2s})$ as well as their node-node mappings. The algorithm tries to select one node pair, (v_i, v_j), where v_i is from $\mathcal{G}_1$ and v_j is from $\mathcal{G}_2$, as its action, denoted as a_t, and either backtracks to the parent search state if the solution is not promising or continues the search otherwise. Various heuristics on node pair selection policy, denoted as "*policy*", are proposed in McSp and McSp+RL. For example, in McSp, nodes of large degrees are selected before small-degree nodes.

There are two major limitations of McSp and McSp+RL: (1) Such heuristics-based node pair selection policy cannot adapt to different graph structures[2]; (2) The search may

[1]The original algorithm is recursive. To highlight our novelty, we rewrite into an equivalent iterative version.

[2]McSp+RL claims it uses reinforcement learning (RL) to compute a score for each node pair. However, the scores for each node are the only learnable parameters, whose update relies on a heuristic. This requires the learning of scores for each new testing graph pair. In contrast, we use continuous embeddings to represent graphs which are fed into a DQN to compute a score trained on many graph pairs. Once trained, GLSearch can be applied to any new testing pair without retraining.

enter a bad state and get "stuck" without finding a better (larger) solution, $maxSol$, for many iterations.

At each search state, in order to compute the upper bound "UB_t" and reduce the action space "$\mathcal{A}_t$", i.e. the candidate node pairs to select from, the concept of "bidomain" is introduced. Bidomains partition the nodes in the remaining subgraphs, i.e. outside $\mathcal{G}_{1s}$ and $\mathcal{G}_{2s}$, into equivalent classes. Among all bidomains of a given state, $\mathcal{D}$, the k-th bidomain D_k consists of two sets of nodes, $\langle \mathcal{V}'_{k1}, \mathcal{V}'_{k2} \rangle$ where $\mathcal{V}'_{k1}$ and $\mathcal{V}'_{k2}$ have the same connectivity pattern with respect to the already matched nodes $\mathcal{V}_{1s}$ and $\mathcal{V}_{2s}$. Figure 5.1 shows an example with three bidomains. Due to the subgraph isomorphism constraint posed by MCS, only nodes in $\mathcal{V}'_{k1}$ can match to $\mathcal{V}'_{k2}$ and vice versa. This also guarantees the extracted subgraphs at each state are isomorphic to each other. Thus, each bidomain can contribute at most $\min(|\mathcal{V}'_{k1}|, |\mathcal{V}'_{k2}|)$ nodes to the future best solution. Therefore, the upper bound can be estimated as $\sum_{D_k \in \mathcal{D}} \min(|\mathcal{V}'_{k1}|, |\mathcal{V}'_{k2}|)$. This upper bound computation is consistently used for all the methods in the chapter. The major difference is in the policy for node pair selection.

5.3 Proposed Method

In this section we formulate the problem of MCS detection as learning an RL agent that iteratively grows the extracted subgraphs by adding new node pairs to the current subgraphs in a graph-structure-aware environment. We first describe the environment setup, then depict our proposed Deep Q-Network (DQN) which provides actions for our agent to grow the subgraphs in a search context. We also describe how to leverage supervised data via pre-training and imitation learning.

5.3.1 Leveraging DQN for Search

Since graph matching for MCS detection must satisfy a hard constraint that the resulting two subgraphs must be isomorphic to each other, instead of learning to match two graphs in one shot, we design an RL agent which explores the input graph pair and sequentially grows

the extracted two subgraphs one node pair at a time. The iterative subgraph extraction process can be described by a Markov Decision Process (MDP), where the definitions of state and action are the same as Section 5.2.2. The difference is that, for MDP, reward needs to be defined too. for MCS, the immediate reward for transitioning from one state to any next state is $r_t = +1$ since one new node pair is selected.

To address the issue that the algorithm may get stuck in a bad state for many iterations without finding a larger solution, we utilize additional information stored in Q-values computed by our learned model. We suppose backtracking to an earlier better state can alleviate such issue in practice, but there lacks a principled measure for McSp and McSp+RL to determine which earlier state is better. By design, our node pair selection policy is a learned DQN, so our agent knows not only the quality of immediate actions, but also the values associated with previous states. Therefore, if the best solution found so far does not increase, i.e. we do not enter line 10 of Algorithm 5.1 for a pre-defined number of iterations, in the next iteration, we find the best state as determined by the Bellman Equation, remove that state, then visit it on line 7. We refer to this improved search methodology as **promise-based search**.

5.3.2 Representation Learning for DQN

Since the action space can be large for MCS, we leverage the representation learning capacity of continuous representations for DQN design. At state s_t, for each action a_t, our DQN predicts a $Q(s_t, a_t)$ representing the future reward to go if the action $a_t = (i, j)$ where $i \in \mathcal{V}_1$ and $j \in \mathcal{V}_2$ is selected, intuitively corresponding to the largest number of nodes that will be eventually selected starting from the action edge (s_t, a_t) as in tree in Figure 7.1.

Based on the above insights, one can design a simple DQN leveraging the representation learning power of Graph Neural Networks (GNN) such as [7] and [9] by passing $\mathcal{G}_1$ and $\mathcal{G}_2$ to a GNN to obtain one embedding per node, $\{\boldsymbol{h}_i | \forall i \in \mathcal{V}_1\}$ and $\{\boldsymbol{h}_j | \forall j \in \mathcal{V}_2\}$. Denote CONCAT as concatenation, READOUT as a readout operation that aggregates node-level embeddings into subgraph embeddings $\boldsymbol{h}_{s1}$ and $\boldsymbol{h}_{s1}$, and whole-graph embeddings $\boldsymbol{h}_{\mathcal{G}_1}$ and $\boldsymbol{h}_{\mathcal{G}_2}$. A state

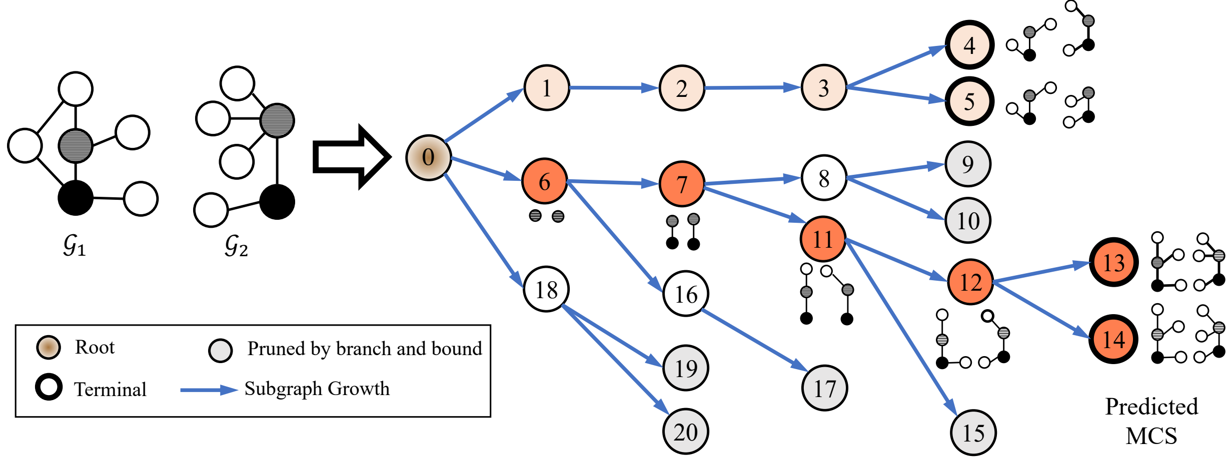


Figure 5.2: An illustration of the search process for MCS detection. For $(\mathcal{G}_1, \mathcal{G}_2)$, the branch and bound search algorithm (Section 5.2.2 and Algorithm 5.1) yields a tree structure where each node represents one state (s_t) with id reflecting the order in which states are visited, and each edge represents an action (a_t) of selecting one more node pair. The search is essentially depth-first with pruning by the upper bound check. Our model learns the node pair selection strategy, i.e. which state to visit first. If state 6 can be visited before state 1, a large solution can be found in less iterations. When the search completes or a pre-defined search iteration budget is used up, the best solution (output subgraphs) will be returned, corresponding to state 13 (and 14).

can then be represented as $\boldsymbol{h}_{s_t} = \text{CONCAT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2}, \boldsymbol{h}_{s1}, \boldsymbol{h}_{s2})$. An action can be represented as $\boldsymbol{h}_{a_t} = \text{CONCAT}(\boldsymbol{h}_i, \boldsymbol{h}_j)$. The Q function can the be designed as:

$$Q(s_t, a_t) = \text{MLP}\big(\text{CONCAT}(\boldsymbol{h}_{s_t}, \boldsymbol{h}_{a_t})\big) = \text{MLP}\big(\text{CONCAT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2}, \boldsymbol{h}_{s1}, \boldsymbol{h}_{s2}, \boldsymbol{h}_i, \boldsymbol{h}_j)\big). \tag{5.1}$$

However, there are several flaws to this simple design of Q function:

(A) $\boldsymbol{h}_i$ and $\boldsymbol{h}_j$ generated by typical GNNs encode only local neighborhood information, but $Q(s_t, a_t)$ represents the long-term effect of adding (i, j). What is worse, different node pairs have different embeddings, but their immediate rewards are always +1 in MCS.

(B) Swapping the order of $\mathcal{G}_1$ and $\mathcal{G}_2$ should not cause $Q(s_t, a_t)$ to change, but concatenating embeddings from the two graphs causes the DQN to be sensitive to their ordering.

(C) How to effectively leverage the node-node mappings between $\mathcal{G}_{1s}$ and $\mathcal{G}_{2s}$ for predicting $Q(s_t, a_t)$ remains a challenge.

To address these issues, we propose the following improvements over the simple DQN design.

Factoring out Action In order to maximally reflect the effect of adding node pair (i, j) to $\mathcal{G}_{1s}$ and $\mathcal{G}_{2s}$, we first notice that $Q^*(s_t, a_t) = r_t + \gamma V^*(s_{t+1}) = 1 + \gamma V^*(s_{t+1})$ in MCS, where Q and V are the Q and value functions, respectively, and γ is the discount factor. Then, in order to compute the effect of a_t, we can compute the value associated with s_{t+1} which does not depend on a_t and avoids the use of local $\boldsymbol{h}_i$ and $\boldsymbol{h}_j$.

Interaction between Input Graphs To resolve the graph symmetry issue, we first construct the interaction between the embeddings from two graphs, i.e. $\textsc{interact}(\boldsymbol{h}_{x1}, \boldsymbol{h}_{x2})$, where $\boldsymbol{h}_{x1}$ and $\boldsymbol{h}_{x2}$ represent any embedding from $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively and $\textsc{interact}(\cdot)$ is any commutative function to combine the two embeddings (e.g. summation). This interacted embedding is later concatenated with other useful representations and fed into a final MLP to compute the Q score.

Bidomain Representations Bidomains are derived from node-node mappings and partition the rest of $\mathcal{G}_1$ and $\mathcal{G}_2$, which is a more useful signal for predicting the future reward. In fact, as described in Section 5.2.2, bidomains have been adopted to in search-based MCS solvers to estimate the upper bound. Here, we require the harder prediction of $Q(s_t, a_t)$ for which we propose to use the representation of bidomains. Denote $\boldsymbol{h}_{D_k}$ as the representation for bidomain D_k. Similar to computing the graph-level and subgraph-level embeddings, we compute $\boldsymbol{D}_k$ as

$$\boldsymbol{h}_{D_k} = \textsc{interact}\big(\textsc{readout}(\{\boldsymbol{h}_i | i \in \mathcal{V}'_{k1}\}), \textsc{readout}(\{\boldsymbol{h}_j | j \in \mathcal{V}'_{k2}\})\big). \tag{5.2}$$

Since we require the MCS to be connected subgraphs, we differentiate bidomains $\mathcal{D}^{(c)}$ that are connected (adjacent) to $\mathcal{G}_{1s}$ and $\mathcal{G}_{2s}$ from the single bidomain D_0 disconnected (unconnected) from $\mathcal{G}_{1s}$ and $\mathcal{G}_{2s}$ (e.g. bidomain "00" in Figure 5.1). Given all the bidomain embeddings, we compute a single representation for $\mathcal{D}^{(c)}$, $\boldsymbol{h}_{\mathcal{D}c} = \textsc{readout}(\{\boldsymbol{h}_{D_k} | k \in \mathcal{D}^{(c)}\})$. Our final

DQN has the form:

$$Q(s_t, a_t) = 1+\gamma\text{MLP}\Big(\text{CONCAT}\big(\text{INTERACT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2}), \text{INTERACT}(\boldsymbol{h}_{s1}, \boldsymbol{h}_{s2}), \boldsymbol{h}_{\mathcal{D}c}, \boldsymbol{h}_{\mathcal{D}0}\big)\Big). \quad (5.3)$$

5.3.3 Leveraging Search for DQN Training

At each state s_t, the action space size in the worst case is quadratic to the number of nodes in the remaining subgraphs. Thus, to enhance the training of our DQN, before the standard training of DQN [101], we pre-train DQN and guide its exploration with expert trajectories supplied by the search algorithm.

For the pre-training stage, we first observe the overall mse loss is $(y_t - Q(s_t, a_t))^2$ where y_t the target for iteration t and $Q(s_t, a_t)$ is the predicted $Q(s_t, a_t)$. We then notice that for small training graph pairs, the complete exploration of search space can be performed to obtain the true target for every (s_t, a_t) by finding the longest sequence starting from s_t to a leaf node in the search tree.

For larger graph pairs though, finding the true target becomes too slow. In that case, after pre-training, we enter the imitation learning stage where we let the agent mimic the decision made by the state-of-the-art MCS search algorithm instead of relying on its own predicted $Q(s_t, a_t)$.

5.4 Experiments

We evaluate GLSearch against two state-of-the-art exact MCS detection algorithms and a series of approximate graph matching methods from various domains. We conduct experiments on a variety of medium-sized synthetic graph datasets and real-world graph datasets. Among the different baseline models, we find no consistent trend. This indicates the difficulty of our task, as existing methods can not find a consistent policy that guarantees good performance on datasets from different domains. Our model can substantially outperform the baselines, highlighting the significance of our contributions to learning for search.

5.4.1 Baseline Methods

There are two groups of methods: Exact MCS algorithms including McSp [97] and McSp+RL [96], learning based graph matching models including GW-QAP [102], I-PCA [98], and NeuralMcs [99].

All the methods either originally use or are adapted to use the branch and bound search framework in Section 5.2.2 with differences in node pair selection policy and training strategies. GW-QAP performs Gromov-Wasserstein discrepancy [103] based optimization *for each graph pair* and outputs a matching matrix $\boldsymbol{Y}$ for all node pairs indicating the likelihood of matching which is treated the same way as our q scores, i.e. at each search iteration we index into $\boldsymbol{Y}$ to select a node pair. I-PCA and NeuralMcs also output a matching matrix but require supervised training, and thus are trained using the same training data graph pairs as our GLSearch but with different loss functions and training signals. During testing, we apply the trained model on all testing graph pairs. For medium-size synthetic testing graph pairs, each method is given a budget of 500 search iterations. For large real-world graph pairs, each method is given a budget of 7500 search iterations. These budgets were chosen based on when the models' performances stabilized. Details about performance using other iteration budgets may be found in the Supplementary Material.

To validate the usefulness of the learned DQN, we compare GLSearch, our full model, with a randomly initialized model, GLSearch-Rand, which replaces the output of our DQN with a completely random scalar. We show the performance gain of our model through training by substantially outperforming this baseline on all real-world datasets.

5.4.2 Parameter Settings

For I-PCA, NeuralMcs and GLSearch, we utilize 3 layers of Graph Attention Networks (GAT) [9] each with 64 dimensions for the embeddings. The initial node embedding is encoded using the local degree profile [104]. We use $\text{ELU}(x) = \alpha(\exp(x)–1)$ for $x \leq 0$ and x for $x > 0$ as our activation function where $\alpha = 1$. We run all experiments with

Table 5.1: Results on small and medium graphs. Each synthetic dataset consists of 50 randomly generated pairs labeled as "⟨generation algorithm⟩-⟨number of nodes in each graph⟩". "BA", "ER", and "WS" refer to the Barabási-Albert (BA) [2], the Erdős-Rényi (ER) [3], and the Watts–Strogatz (WS) [4] algorithms, respectively. Nci109 consists of 100 chemical compound graph pairs whose average graph size is 28.73. We show the ratio of the (average) size of the subgraphs found by each method with respect to the best result on that dataset.

Method	**BA-50**	**BA-100**	**ER-50**	**ER-100**	**WS-50**	**WS-100**	**Nci109**
McSp	0.913	0.892	0.842	0.896	0.905	0.856	0.948
McSp+RL	0.923	0.857	0.844	0.877	0.913	0.875	0.948
gw-qap	0.945	0.887	0.855	0.925	0.916	0.898	0.966
I-pca	0.899	0.863	0.848	0.923	0.879	0.852	0.951
NeuralMcs	0.908	0.889	0.846	0.906	0.889	0.865	0.954
GLSearch-Rand	0.995	0.987	0.920	0.978	0.967	0.931	0.989
GLSearch	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Best Solution Size	19.12	34.38	26.56	37.64	29.48	55.56	10.48

Intel i7-6800K CPU and one Nvidia Titan GPU. For DQN, we use MLP layers to project concatenated embeddings to a scalar. We use sum followed by an MLP for readout and 1dconv+maxpool followed by an MLP for interact. For training, we set the learning rate to 0.001, the number of training iterations to 10000, and use the Adam optimizer [45]. The models were implemented with the PyTorch and PyTorch Geometric libraries [105].

5.4.3 Results

The key property of GLSearch is its ability to find the best solution in the fewest number of search iterations. As shown in Table 5.1, our model outperforms baselines in terms of size of extracted subgraphs on all medium-sized synthetic graph datasets and the small chemical compound dataset Nci109. However, baseline solvers are already quite powerful on these datasets. As it is easy to extract the maximum common subgraph on smaller graph datasets because the total search space grows exponentially with graph size, to truly show the performance advantage of GLSearch, we also run experiments on large real-world graph datasets with thousands of nodes.

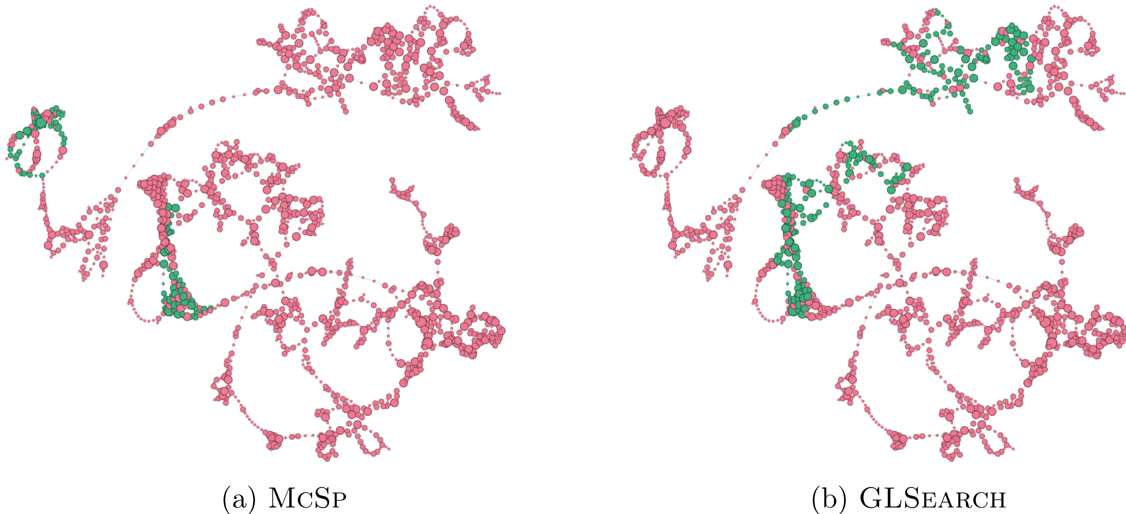
As shown in Table 5.2, our model outperforms baselines in terms of the size of the extracted subgraphs on all large real-world datasets. The exact solvers rely on heuristics for node selection, and consistently find smaller subgraphs compared to our results. Figure 5.3

Table 5.2: Results on real-world large graph pairs. Each dataset consists of one large real graph pair ($\mathcal{G}_1$, $\mathcal{G}_2$ may not be isomorphic, but $\mathcal{G}_{1s}$, $\mathcal{G}_{2s}$ are isomorphic guaranteed by search). Below each dataset name, we show its size $\min(|\mathcal{V}_1|, |\mathcal{V}_2|)$ to indicate these pairs are significantly larger than the ones in Table 5.1. Consistent with Table 5.1, we show the ratio of the subgraph sizes.

Method	**Road** 652	**DbEn** 1945	**DbZh** 1907	**Dbpd** 1907	**Enro** 3369	**CoPr** 3518	**Circ** 4275	**HPpi** 2152
McSp	0.374	0.815	0.797	0.722	0.694	0.684	0.498	0.864
McSp+RL	0.771	0.699	0.589	0.434	0.742	0.674	0.583	0.787
gw-qap	0.305	0.929	0.855	0.808	0.711	0.860	0.354	0.834
I-pca	0.267	0.551	0.589	0.607	0.650	0.707	0.203	0.762
NeuralMcs	0.977	0.785	0.616	0.620	0.737	0.742	0.561	0.785
GLSearch-Rand	0.641	0.762	0.658	0.639	0.814	0.755	0.603	0.814
GLSearch	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Best Solution Size	131	508	482	521	543	791	3515	404

compares results by McSp and GLSearch. Since McSp selects nodes with large degrees as its heuristic, the selected nodes tend to be confined in one dense cluster of large degree nodes in $\mathcal{G}_1$. This implies the subgraph in $\mathcal{G}_2$ matched to this dense cluster must also be dense (isomorphism constraint of MCS). In contrast, GLSearch is able to find long chains in $\mathcal{G}_1$ which allows easier matching in $\mathcal{G}_2$. In general, there are many cases of large real-world graph pairs where heuristics are not enough to extract large high quality subgraphs. Due to its leveraging both learning and search, GLSearch consistently finds subgraphs more than double the size of those found by search based baselines for large real-world graph pairs.

Compared with learning based graph matching models, GLSearch is the only model which learns a reward that is dependent on both state and action, i.e. $Q(s_t, a_t)$. gw-qap, I-pca, and NeuralMcs essentially pre-compute the matching scores for all the node pairs in the input graphs, and therefore at each search step, the scores cannot adapt to the particular state, i.e. the matching scores only depend on $\mathcal{G}_1, \mathcal{G}_2$. Notice our state representation includes $\mathcal{G}_1, \mathcal{G}_2$ as well, hence GLSearch has more representational power than baselines. Trained under a reinforcement learning framework guided by search, GLSearch also performs the best among learning based baselines.



(a) McSp (b) GLSearch

Figure 5.3: Visualization of MCS results on Road. Nodes with large degrees have large circles. For each method, we show the two graphs being matched. Selected subgraphs are colored in green.

Table 5.3: Abaltion study on real datasets.

Method	**Road**	**DbEn**	**DbZh**	**Dbpd**	**Enro**	**CoPr**	**Circ**	**HPpi**
GLSearch (no $\boldsymbol{h}_{\mathcal{G}}$)	0.977	0.878	0.925	0.845	0.860	0.987	0.980	0.960
GLSearch (no $\boldsymbol{h}_{s}$)	**1.000**	0.874	0.894	0.869	0.928	**1.000**	0.801	0.913
GLSearch (no $\boldsymbol{h}_{\mathcal{D}c}$)	0.803	0.780	0.687	0.818	0.740	0.804	0.505	0.849
GLSearch (no $\boldsymbol{h}_{\mathcal{D}0}$)	0.576	0.856	0.782	0.768	0.823	0.932	0.323	0.938
GLSearch (sum interact)	0.902	0.913	0.963	0.885	0.899	0.957	**1.000**	0.948
GLSearch (unfactored)	0.447	0.807	0.712	0.582	0.816	0.816	0.512	0.861
GLSearch (unfactored-i)	0.500	0.789	0.741	0.772	0.748	0.825	0.902	0.864
GLSearch	0.992	**1.000**	**1.000**	**1.000**	**1.000**	0.990	0.881	**1.000**
Best Solution Size	132	508	482	521	543	799	3989	404

5.4.4 Ablation and Parameter Study

To evaluate the effectiveness of different components proposed in our DQN model, we run ablation studies on all real world datasets.

We first measure the importance of each embedding vector fed to our DQN module, as described by Equation 5.3. We remove each embedding vector (specifically: $\boldsymbol{h}_{\mathcal{G}} = \text{INTERACT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2})$, $\boldsymbol{h}_s = \text{INTERACT}(\boldsymbol{h}_{s1}, \boldsymbol{h}_{s2})$, $\boldsymbol{h}_{\mathcal{D}c}$, and $\boldsymbol{h}_{\mathcal{D}0}$) individually from the DQN model and retrain the model under the same training settings. Table 5.3 is consistent with our conclusion that every embedding vector used by GLSearch is critical in capturing the search state's representation. Furthermore, we find leveraging bidomain representations is

very beneficial to our model.

We next measure the importance of interaction to address the symmetry issue of the MCS calculation, where input graph pairs must be order insensitive. We first test the necessity of using more complex interaction functions, by replacing our 1DCONV+MAXPOOL interaction with simple SUM for interaction (still followed by an MLP). As shown in Table 5.3, we see that simpler interaction functions may not be powerful enough to encode the interaction between 2 graphs. Particularly, this suggests that interaction is quite important to model performance.

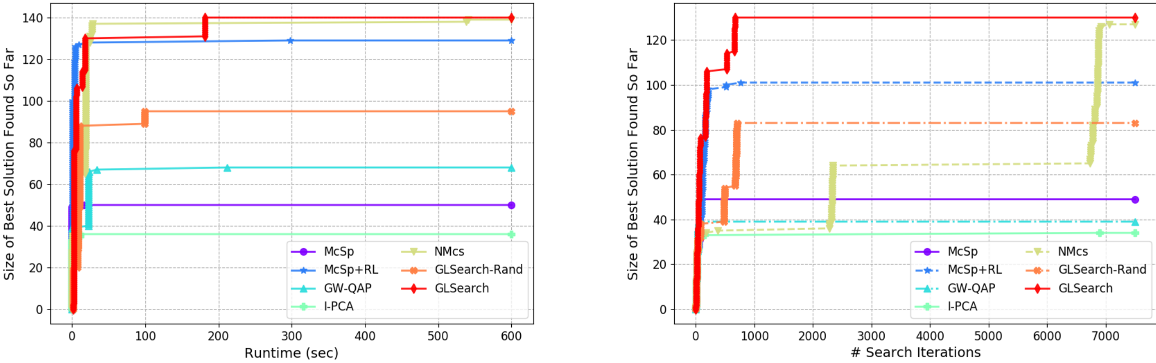
Finally, we measure the importance of factoring out actions from our DQN model. We test this with 2 models. The first utilizes Equation 5.1 to encode the Q-value, which we refer to as GLSearch (unfactored). Since Equation 5.1 also suffers from the issue of graph symmetry, we adapt this model to use the same interaction function as GLSearch to construct 3 order-invariant embeddings $\boldsymbol{h}_{\mathcal{G}} = \text{INTERACT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2})$, $h_s = \text{INTERACT}(\boldsymbol{h}_{s1}, \boldsymbol{h}_{s2})$, $\boldsymbol{h}_a = \text{INTERACT}(\boldsymbol{h}_i, \boldsymbol{h}_j)$ to concatenate and pass to the final MLP layer in Equation 5.1. We refer to this model as GLSearch (unfactored-i). Our results show that without factoring out the action, our performance is comparable to or worse than McSp, indicating the significant performance boost introduced by maximally reflecting the effect of adding node pairs.

5.4.5 Overhead of GLSearch

Although in each iteration, GLSearch needs to compute a Q score, and is thus more computationally expensive than McSp and McSp+RL, the decision GLSearch makes is "smarter" so that across many iterations, GLSearch can find a larger solution. In order to verify this, for each method, in each search iteration, we collect the best solution (largest subgraph) found so far, and plot the best solution size across search iterations. Since we also measure the running time at each iteration, we also plot the best solution size across running time.

As shown in Figure 5.4, across the search, initially GLSearch finds a smaller solution compared to some baseline methods, but, with more iterations and running time, GLSearch

quickly outperforms more and more baselines, eventually becoming the best method. Since all the methods follow the same search framework, and the only difference is the node pair selection policy, this experiment verifies that our learned policy can indeed find a larger common subgraph. Given infinite amount of time budget, all the methods would find the true MCS which is the same or larger than the current best solution, but the aim of this section is to show the advantage of the learned policy for search allows the finding of a better solution in fewer iterations and running time.



(a) Best solution size across time. (b) Best solution size across search iterations.

Figure 5.4: Comparison of the best solution sizes of different methods on Road.

5.5 Conclusion

We believe the interplay of search and learning is a promising research direction, and take a step towards bridging the gap by tackling the NP-hard challenging task, Maximum Common Subgraph detection. We have proposed a reinforcement learning method which unifies search and deep Q-learning into a single framework. By using the search to train our carefully designed DQN, the DQN provides better node selection policy for search to find large common subgraph solutions faster, which is experimentally verified on synthetic and real-world large graph pairs. In the future, we will explore the adaptation of our framework which combines learning with search to other constrained combinatorial problems, e.g. Maximum Clique Detection.

CHAPTER 6

Unifying Geometric Regularization and Policy Gradients for Subgraph Matching

6.1 Introduction

With the growing amount of graph data that naturally arises in many domains, solving graph-related tasks via machine learning has gained increasing attention. A particularly interesting graph-related task is subgraph matching, which requires the detection of all occurrences of a small query graph in an orders-of-magnitude larger target graph. subgraph matching has wide applications in graph database search [106], knowledge graph query [107], biomedical analysis [108], social group finding [109], quantum circuit design [110], etc. As a concrete example, subgraph matching is used for protein complex search in a protein-protein interaction network to test whether the interactions within a protein complex in a species are also present in other species [111].

There are two large bodies of work tackling subgraph matching from different angles. The first direction utilizes representation learning to encode subgraph properties directly into node or edge embeddings to compute soft node-node correspondances [105, 112, 113, 114]. The second direction uses heuristics or reinforcement learning to search for occurrences of small query graphs in the large target graph [22, 96, 99, 115, 116, 117].

On one hand, recent advances in representation learning have found increasingly better ways of compute soft node-node [105, 112, 113] or edge-edge [114] correspondence scores between query graphs and target graph. In particular, NMatch [113] focuses on learning representations that encode transitivity, anti-symmetry, intersection set, and non-trivial

intersection properties unique to subgraph matching. While these methods show promise, they do not focus on extracting an isomorphic subgraph, and are evaluated by ranking individual node-node or edge-edge correspondence in relation to a single known ground-truth mapping [105, 112, 113, 114]. Naive methods of extracting the subgraph are proposed in these works, such as local-neighborhood voting [113] or the Hungarian algorithm [114], but we find they only work for small graphs and fail to scale to larger query or target graph pairs.

On the other hand, search-based methods, such as traditional solvers [115, 118, 119, 120, 121, 122, 123, 124] and reinforcement learning (RL) [22, 96, 116], continue to advance advance the search procedure, which incrementally match node-node pairs between the small query graph and large target graph until the entire subgraph has been matched. These algorithms enumerate all possible node-node matches and attempt to exhaust the search space as quickly as possible to obtain a valid subgraph matching [115]. Said works adopt different heuristics for ordering query graph nodes to improve the speed of exhaustion [115, 118, 119, 120, 121, 122, 123, 124]. Recently, RL has been proposed to learn this ordering [96, 116], again with the goal of reducing search space. More recent works on graph matching greedily arrives at a good solution before fully exhausting the search space [22, 117]. These approaches, which we call "learning to search", scale to larger inputs, where the search space cannot fully be pruned [22].

Our goal is to combine the best design choices from these two parallel lines of work to perform non-induced isomorphic subgraph matching at scale. At scale, representation learning approaches [113, 114] alone fail to find subgraphs that satisfy isomorphism constraints due to a lack of principled methods using said representations. At scale, most traditional solvers [115, 118, 119, 120, 121, 122, 123, 124] and RL [116] approaches also fail, as exhausting the search space becomes computationally infeasible with larger input graph sizes, because subgraph matching is an NP-Hard task.

The most suitable paradigm for our problem is "learning to search" [22]. However, naively using "learning to search" ignores recent search algorithm and representation learning devel-

opments in subgraph matching. Notably, subgraph matching requires the entire subgraph to be matched. Because of this constraint, (1) search algorithms can prematurely prune large portions of the search space before the search starts [115]. For example, an algorithm can safely prune any node in the target graph that has a degree smaller than a query graph node, since the task requires all of the nodes and edges in the query must be matched in the target graph. Such early pruning enables the possibility of designing a simpler policy that is globally computed once before search which is faster and potentially easier to train compared against GLSearch [22]. Besides, (2) newer representation learning paradigms encode the transitivity, anti-symmetry, intersection set, and non-trivial intersection subgraph properties [113] directly into node embeddings. We propose NSubS, Neural Subgraph Search, which combines recent developments in subgraph matching with "learning to search" to perform scalable and effective subgraph matching on large graphs. Our contributions are:

- We propose an effective RL framework unifying recent developments in representation learning, search-based solvers, and learning to search paradigm for subgraph matching.
- We inject the transitivity, anti-symmetry, intersection set, and non-trivial intersection properties into a novel RL loss.
- We find a simple global graph matching neural network policy network achieves surprisingly good performance and propose a way of scaling GMN by disentangling intra- and inter-graph message passing.
- We conduct extensive experiments on real-world graphs to demonstrate the effectiveness of the proposed approach compared against a series of strong baselines in subgraph matching.

6.2 Related Work

Non-learning methods on subgraph matching Existing methods on subgraph matching can be broadly categorized into backtracking search algorithms [116, 118, 119, 120, 123,

125] and multi-way join approaches [126, 127, 128, 129]. The former category of approaches employ a branch and bound approach to grow the solution from an empty subgraph by gradually seeking one matching node pair at a time following a strategic order until the entire search space is explored. The multi-way join approaches rely on decomposing the query graph into nodes and edges and performing join operations repeatedly to combine the partially matched subgraphs to q. However, they tend to work well on small query graphs generally with less than 10 nodes [115], and thus we follow and compare against methods in the former category, whose details will be shown in Section 6.3.2.

Learning-based methods for subgraph-related problems The idea of designing graph neural networks for graph-graph similarity has been explored, but not subgraph matching. GMN [130] captures general notions of graph similarity through inter-graph message passing, but outputs a similarity score instead of the discrete matching between 2 graphs. DMPNN [131] uses node to edge conversions to obtain node and edge representations that better preserve isomorphism properties, but outputs approximate subgraph isomorphism counts instead of a discrete matching.

Representation learning methods for subgraph matching use a geometric loss to provide soft node-node correspondance scores, but can not return a discrete mapping of where the query occurs in the target graph. For example, NMATCH [113] learns node embeddings to predict a score for an input subgraph-graph pair indicating whether the subgraph is contained in another graph. ISONET [114] extends this idea using edge-edge correspondence scores to rank which query graphs are most likely to appear in the target.

Another direction of research aims to perform subgraph counting [116, 132, 133, 134] supervised on the number of specific substructures, and again lacks an explicitly search strategy and thus falls short of yielding solutions for subgraph matching. Researchers tackling consistent subgraph matching [135] handle complex node or edge constraints using a subgraph matching subroutine. Hence, advancements to subgraph matching can directly benefit such works.

Efforts on using RL for graph NP-hard problems The idea of using RL to replace heuristics in search algorithms for NP-hard graph-related tasks is not new [136, 137], and we identify two works similar to the present work. (1) GLSearch [22] detects the maximum common subgraph (MCS) in an input graph pair, which is different from subgraph matching which requires the entire q to be matched with G, allowing further improvement in the neural network and search design. (2) RL-QVO [116] tackles subgraph matching via ordering the nodes in the *query* graph as a global pre-processing step before search, which is an orthogonal direction to our approach to select nodes in the target graph computed at each search step.

6.3 Preliminaries

6.3.1 Problem Definition

We denote a query graph as $q = (V_q, E_q)$ and a target graph as $G = (V_G, E_G)$ where V and E denote the node and edge sets. q and G are associated with a node labeling function L_g which maps every node into a label l in a label set Σ. **Subgraph**: For a subset of nodes S of V_q, $q[S]$ denotes the subgraph of q with an node set S and a edge set consisting of all the edges in E_q that have both endpoints in S. In this chapter, we adopt the definition of non-induced subgraph. **Subgraph isomorphism**: q is subgraph isomorphic to G if there exists an injective node-to-node mapping $M : V_q \to V_G$ such that (1) $\forall u \in V_q, L_g(u) = L_g(M(u))$; and (2) $\forall e_{(u,u')} \in E_q, e_{(M(u),M(u'))} \in E_G$. **subgraph matching**: The task of subgraph matching aims to find the subgraphs in G that are isomorphic to q. We call M a solution, or a match of q to G. We call a pair (q, G) is solved if the algorithm can find any match under a given time limit, which we find a challenge for existing solvers on input graphs in experiments especially on large graphs. For solved pairs, the number of found subgraphs by an algorithm is reported. It is noteworthy that we require the entire query graph to be matched, and consider a pair unsolved if the query is only partially matched to the target graph.

Algorithm 6.1: Search-based subgraph matching.

1: **Input:** Query graph q, data graph G.
2: **Output:** Matches from q to G.
3: Filter: $C \leftarrow$ generate candidate node sets.
4: Order: $\phi \leftarrow$ generate an ordering for V_q.
5: Search: $\texttt{Backtracking}(q, G, C, \phi, \{\})$.

6.3.2 Search-based methods for subgraph matching

Due to the NP-Hard nature of subgraph matching, backtracking search [115] is a naturally suitable algorithm since it exhaustively explores the solution space by starting with an empty match and adding one new node pair to the current match at each step. When the current match cannot be further extended, the search backtracks to its previous search state, and explores other node pairs to extend the match. However, naively enumerating all the possible states in the entire search space is intractable in practice, and therefore existing efforts mainly aim to reduce the total number of search steps for the backtracking search via mainly three ways [115]: (1) Filter nodes in G to obtain a small set of candidate nodes for each node in q as a pre-processing step before the backtracking search; (2) Order the nodes in q before the search; (3) Generate a local candidate set of nodes in each step of the search based on the current search state. Algorithm 6.1 summarizes the overall backtracking search based framework for subgraph matching. It is worth noting that the first three means each correspond different steps in the algorithm, and therefore any improvement in any of the three steps can be regarded as orthogonal to each other.

The basic idea of backtracking search is outlined in Algorithm 6.2. The recursive algorithm starts with an empty node-node mapping, and tries to add one new node pair to the mapping M at each recursive call. The action is the new node pair (u_t, v_t), where $u_t \in V_q$ is selected according to the heuristic-based ordering ϕ, and $v_t \in V_G$ is selected according to a policy (which is to be learned by NSubS) to be one of the local candidate nodes (line 8), that can be mapped to u_t. It is noteworthy that this local candidate node set "$\mathcal{A}_{u_t} \subseteq V_G$" is refined over the global candidate sets C based on the current search state s_t. s_t is defined as

Algorithm 6.2: `Backtracking`(q, G, C, ϕ, M)

```
1: Input: q, G, C, ϕ, and current mapping M.
2: Output: Subgraph match mappings.
3: if |M| = |V_q| then
4:    output M;
5:    return;
6: end if
7: u_t ∈ V_q ← ϕ(M);
8: A_{u_t} ← s_t.getLocalCand(u_t);
9: A_{u_t,ordered} ← policy(s_t, A_{u_t});
10: for v_t in A_{u_t,ordered} do
11:    M ← M.add(u_t, v_t);
12:    Backtracking(q, G, C, ϕ, M);
13:    M ← M.remove(u_t, v_t);
14: end for
```

(q, G) along with the current mapping M and u_t. $\mathcal{A}_{u_t} \subseteq V_G$ ensures any node in $\mathcal{A}_{u_t}$ would lead to the extended subgraphs at s_t still being isomorphic to each other. Thus, the local candidate set $\mathcal{A}_{u_t}$ for u_t is the action space, for which we learn a policy (line 9) to order the nodes, resulting in an ordered list $\mathcal{A}_{u_t,\text{ordered}}$.

Note, effective candidate node sets are unique to subgraph matching, as subgraph matching requires the whole query graph to be matched. Hence, state-of-the-art search algorithms [115] immediately prune nodes in the target graph that cannot match any query node (e.g. the degree of the target node is lower than the degree of all query nodes). Recent advancements from the search community [22, 96, 99, 115, 116, 117] have greatly reduced the size of C and $\mathcal{A}_{u_t}$, such that the search algorithm only needs to enumerate a tiny fraction of nodes in V_G. The size of the candidate set, $|C[u]|$, can be more than 100x smaller than the size of all possible target node matchings for a given query node, $|V_G|$. For this reason, we believe subgraph matching employs a **highly efficient search framework**, in stark contrast to other NP-Hard problems, such as Maximum Common Subgraph, which lacks the constraints used in candidate set pruning.

While existing efforts can drastically reduce search space with C, a good ϕ, and a small $\mathcal{A}_{u_t}$, we observe that the size of $\mathcal{A}_{u_t}$ can still be up to thousands of nodes for many real-world

large target graphs, calling for a smarter policy to order the nodes not only in q but also in G ($\mathcal{A}_{u_t}$ to be specific). To the best of our knowledge, all the existing methods [22, 96, 99, 115, 116, 117, 134] adopt a random ordering in $\mathcal{A}_{u_t,\text{ordered}}$, as they focus on reducing the search space rather than greedily reaching a solution before fully completing search. We will experimentally show that this attributes to the failure of current state-of-the-art subgraph matching algorithms for many large graph pairs. In fact, a theoretically perfect policy, *policy**, can find the entire match of the query graph in V_q steps or recursive calls, assuming q has at least one match with G. This again inspires our proposed method to improve the policy for node selection from $\mathcal{A}_{u_t}$.

6.4 Proposed Method

In this section we formulate the problem of subgraph matching under the "learning to search" paradigm [22], where an RL agent iteratively chooses actions under a search framework to greedily arrive at a solution. We choose the state-of-the-art search algorithm for subgraph matching outlined in Algorithm 6.2 and detailed in Section 6.3.2, where the RL agent iteratively chooses which new node pairs to add to a partial subgraph matching. We first describe the environment setup, then depict our proposed encoder-decoder policy network, then our novel loss function.

6.4.1 RL Formulation of subgraph matching

To improve existing subgraph matching methods, we adopt "the learning to search" paradigm [22], where the environment is a state-of-the-art backtracking search algorithm, described in Section 6.3.2, which iteratively matches one node pair at a time until no more nodes can be matched. Our policy assigns a score to each action in the action space, $\mathcal{A}_{u_t}$, and therefore can be modeled as a policy network $\pi_\theta(a_t|s_t)$ that computes a probability distribution over $\mathcal{A}_{u_t}$ for the current state s_t, where the node pair to select consists of (u_t, v_t). However, since u_t is predetermined by ϕ, we regard an action as a node v_t selected from $\mathcal{A}_{u_t}$. Since

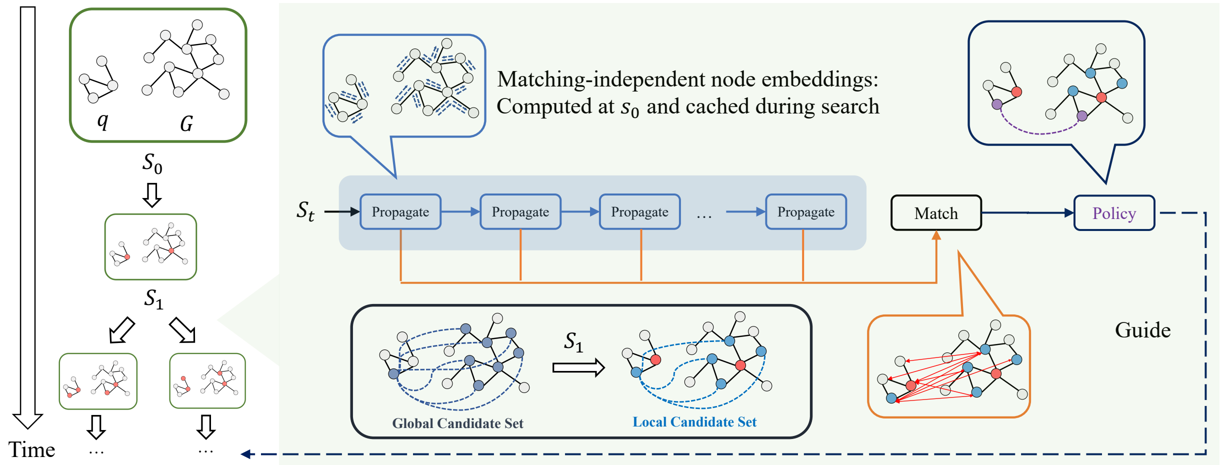


Figure 6.1: The overall process of subgraph matching is a search algorithm that matches one node pair at a time for the input query q and target graph G guided by a learned policy. Due to the large action space incurred by the large G in practice, we propose to train a policy network to guide the selection of local candidate nodes in G at each search state. A global candidate set is computed via a filtering algorithm (line 3 of Algorithm 6.1) before the search starts. In the figure, we illustrate the global candidate set for a particular node in q, which has 6 candidate nodes in G. At each search state, a local candidate set is computed to further reduce the candidate actions (line 8 of Algorithm 6.2). In the figure, the node in q has 4 local candidate nodes in G. From this figure, we can see that global and local candidate states directly reduce the action space by pruning node-node pairs that cannot be matched. The predicted policy guides the search by visiting one node in the local candidate set first, and when backtracked, the search will select another from the candidates, resulting in two branches below s_1. The goal of the policy is to guide the search so that a more promising node is visited first, leading to the early discovery of solutions to this NP-hard problem under a limited time budget.

our focus is to greedily reach the solution instead of reducing the search space, the policy distribution over $\mathcal{A}_{u_t}$, which determines the order node pairs are matched, is much more important than the ordering predetermined by ϕ, which orders query nodes to maximally reduce search space. Since each action is a target node in $\mathcal{A}_{u_t}$, the policy network must learn good node embeddings that capture the underlying subgraph matching to greedily arrive at a good solution before fully exhausting the search space, which will be shown in Section 6.4.3. Because subgraph matching requires matching all nodes in the query graph, we define our reward as $R = 1$ if executing an action returns the fully matched subgraph and $R = 0$ otherwise.

6.4.2 Encoder-Decoder Design for Policy Estimation

Given the efficacy the candidate set filtering step in the backbone search algorithm, line 3 of Algorithm 6.1, which prunes most unpromising actions using subgraph matching's isomorphism constraint and matching the whole query graph constraint, we conjecture unpromising actions conditioned on the current state may already pruned by the backbone search algorithm, and hence do not even appear in the action space, $\mathcal{A}_{u_t}$. In contrast, more general tasks, such as MCS, have more trouble constraining the action space, where the policy must learn to prune unpromising actions conditioned on the current state.

For this reason, we hypothesize a simple global policy network, $\pi_\theta(a_t)$, may be just as effective and easier to optimize than the more complicated state-dependent policy network, $\pi_\theta(a_t|s_t)$. Thus, we propose a simple global policy network, GPE.

Our policy networks follow the widely-used encoder-decoder paradigm [138, 139, 140, 141, 142, 143] to efficiently compute scores for many node-node pairs at once. Namely, the encoder, $\mathcal{E}_\theta$, computes state-independent node embeddings for both the query graph and the target graph, V_q and V_G, denoted as $\{\boldsymbol{h}_u\}_{u\in V_q}$ and $\{\boldsymbol{h}_v\}_{v\in V_G}$ respectively, and the decoder, $\mathcal{D}_\theta$, decodes pairs of node embeddings, $(\boldsymbol{h}_{u_t}, \boldsymbol{h}_{v_t})|v_t \in \mathcal{A}_{u_t}$, into the policy distribution, $\pi_\theta(a_t)$. The decoder is a single bilinear layer followed by a multi-layer perceptron.

6.4.2.1 Global Policy Encoder (GPE)

It is questionable whether a state-dependent encoder is really needed for subgraph matching, as the search algorithm already prunes many unpromising actions in the search state. We conjecture that the success of recent soft subgraph matching works [113, 114] implies a global state-independent policy may perform favorably. If this is the case, we hypothesize training a simpler model may result in better performance, following Occam's razor principle [144, 145, 146, 147].

Studying the design of GLSearch [22], we believe inter-graph matching operations are still largely beneficial in capturing node embeddings. Unlike GLSearch's value network, which restricts inter-graph message passing by carefully examining the search state, we run a full inter-graph message passing between the query graph and the target graph, following Graph Matching Networks [70], as detailed in Equation 6.2. Note, while there have been several related inter-graph message passing graph neural networks proposed in recent years [70, 105, 131, 148, 149, 150], we adopt Graph Matching Networks [70], because it is a canonical model that considers inter-graph messages between all pairs of nodes between the query and target graph.

$$\boldsymbol{h}_v = f_{msg}(\boldsymbol{h}'_v, AGG_{u \in V_q}\{f_{match}(\boldsymbol{h}'_v, \boldsymbol{h}'_u)\}) \tag{6.1}$$

$$\boldsymbol{h}_u = f_{msg}(\boldsymbol{h}'_u, AGG_{v \in V_G}\{f_{match}(\boldsymbol{h}'_u, \boldsymbol{h}'_v)\}) \tag{6.2}$$

In order to scale Graph Matching Networks [70] in our global policy network we notice some efficiency savings: (1) our policy is global, hence we do not need to recompute policy scores for already visited node-node pairs, and (2) the decoder only uses a small number of query and target nodes, and (3) we do not interleave intra- and inter-graph message passing. Hence, we only need to compute the inter-graph node embeddings for nodes in the local candidate set, $\{u_t\} \cup \mathcal{A}_{u_t}$, instead of all nodes, $V_q \cup V_G$, improving the complexity of Graph Matching Networks's inter-graph message passing [70] from $O(|V_q||V_G|)$ to

$O(|V_G|+|\mathcal{A}_{u_t}||V_q|)$. As shown in Figure 7.1, inter-graph message passing does not need to be recomputed as the encoder is state independent. By applying this trick, GRAPH MATCHING NETWORKS can scale to much larger datasets and efficient enough to be called many times through search.

Our global policy encoder, GPE, first constructs state-independent node embeddings, $\{\boldsymbol{h}'_u\}_{u\in V_q}$ and $\{\boldsymbol{h}'_v\}_{v\in V_G}$, using a standard intra-graph message passing [8, 10, 70] graph neural network, $g(\cdot)$. Then we apply inter-graph message passing detailed in Equation 6.2 to obtain our final node embeddings. This policy network encoder, $\mathcal{E}_\theta^{(gpe)}$, is global throughout search and performs full inter-graph message passing. We depict the whole policy network in Figure 7.1.

6.4.3 Loss Function (CGR-Loss)

To learn a policy network that captures underlying subgraph matching properties, we take inspiration from NMATCH [113], which uses a geometrically regularized loss function to capture transitivity, anti-symmetry, intersection set, and non-trivial intersection properties in subgraph matching, which is formulated in Equation 6.4, where P are node-node pairs in a ground truth mapping, N are randomly sampled node-node pairs from the query and target graph, $\mathcal{D}_\theta$ is a simple decoder, $BCE(\cdot)$ is binary cross entropy loss, α is the max margin hyperparameter, and $E(\boldsymbol{h}_u, \boldsymbol{h}_v) = ||max\{0, \boldsymbol{h}_u - \boldsymbol{h}_v\}||_2^2$ denotes an error bound.

$$L = \sum_{(u,v)\in P} BCE(\mathcal{D}_\theta(\boldsymbol{h}_u, \boldsymbol{h}_v), 1) + \sum_{(u,v)\in N} BCE(\mathcal{D}_\theta(\boldsymbol{h}_u, \boldsymbol{h}_v), 0) \tag{6.3}$$

$$+ \sum_{(u,v)\in P} E(\boldsymbol{h}_u, \boldsymbol{h}_v) + \sum_{(u,v)\in N} max\{0, \alpha - E(\boldsymbol{h}_u, \boldsymbol{h}_v)\} \tag{6.4}$$

We unify this loss under the reinforcement learning framework by reformulating it into Equation 6.7, where τ is a hyperparameter tuning what we consider a "large reward" and $G(u_t, v_t)$ returns the cumulative reward after executing action (u_t, v_t), which can be obtained

after the search process. If an action is not executed by search, we take the expected cumulative reward to be 0.

$$L = \sum_{v_t \in \mathcal{A}_{u_t}} BCE(\mathcal{D}_\theta(\boldsymbol{h}_u, \boldsymbol{h}_v), \mathbf{1}[\mathbb{E}_{\pi_\theta}[G(u_t, v_t)] > \tau]) \tag{6.5}$$

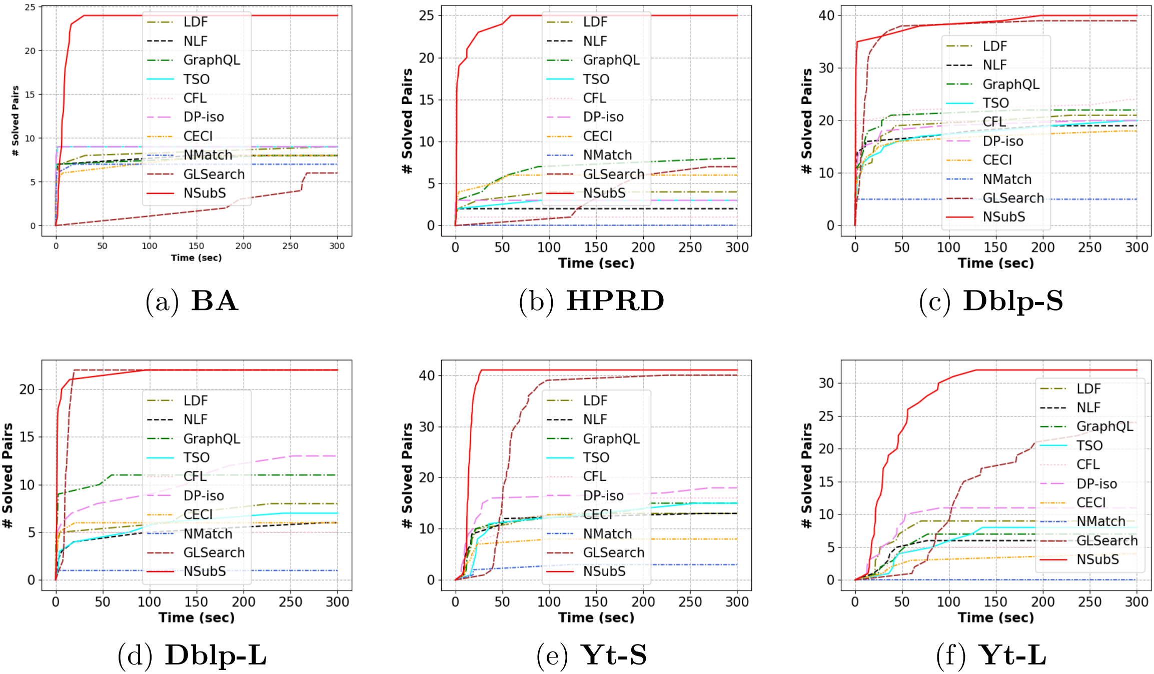
$$+ \sum_{\substack{v_t \in \mathcal{A}_{u_t} \\ \mathbb{E}_{\pi_\theta}[G(u_t,v_t)] > \tau}} E(\boldsymbol{h}_u, \boldsymbol{h}_v) \tag{6.6}$$

$$+ \sum_{\substack{v_t \in \mathcal{A}_{u_t} \\ \mathbb{E}_{\pi_\theta}[G(u_t,v_t)] \leq \tau}} max\{0, \alpha - E(\boldsymbol{h}_u, \boldsymbol{h}_v)\} \tag{6.7}$$

We make the crucial connection that ground truth node-node pairs in the original formulation correspond with node-node pairs that lead to "large reward" under the RL framework. Since RL can only pick actions from the aciton space, we extend the notion of negative sampling in Equation 6.4 to RL by sampling only from the action space. Negative samples can be drawn from both explored and unexplored actions throughout search, in contrast to methods like REINFORCE [151] which relies on receiving reward signals from sampled actions for learning. This works due to reward sparsity, where only a minority of actions will lead to large reward and the majority actions have little reward signal. As we do not want to punish good samples, we only consider negative samples that do not lead to "large reward". Most importantly, by making a direct connection to positive and negative pairs from Equation 6.4, we can apply the geometrically regularized max-margin loss under the RL setting. This encourages the node embeddings used by the policy network to capture desirable subgraph matching properties, as stated above. Concretely, Equation 5 extends the contrastive loss to RL; Equations 6 and 6.7 extend geometric regularization to RL.

To provide ample training signals to the neural network model, we adopt self-supervised learning, where we randomly sample subgraphs from the target graph and record the mapping between the nodes in the sampled subgraph and V_G. Therefore, after the search, we collect

the positive training signals at all states where the node-node pairs (u_t, v_t) lead to a solution. Note, unlike NMatch [113], our loss function is contingent on the search process, thus it can learn a better policy network that is tuned to the backbone search algorithm. Specifically, the policy network can differentiate between multiple actions that lead to different ground truths, by prioritizing the action which induces a smaller search space and hence more likely to produce a positive cumulative reward. As a consequence, Equation 6.7 also more naturally incorporates multiple ground truth solutions as the policy network automatically prioritizes solutions that are easier to reach within the search.



(a) **BA** (b) **HPRD** (c) **Dblp-S**

(d) **Dblp-L** (e) **Yt-S** (f) **Yt-L**

Figure 6.2: The growth in the number of input graph pairs where we find at least one solution. Notice, NSubS is able to outperform state-of-the-art solver baselines. Section 6.6.1 discusses this further. NSubS is also able to beat NMatch and GLSearch, models adapted from the representation learning and "learning to search" communities. Section 6.6.3 discusses this further.

Table 6.1: Target graph description. Details are shown in Section 6.5.1.

Dataset	Domain	$\lvert V_G \rvert$	$\lvert E_G \rvert$
BA	Synthetic	10,000	29,991
HPRD	Biology	9,045	34,853
Dblp	Citation	317,080	1,049,866
Youtube	Social	1,134,890	2,987,624

6.5 Evaluation Protocol

6.5.1 Dataset

We use one synthetic dataset BA and four real-world target graphs. For DBLP and YOUTUBE, we prepare two query sets of small and large query graphs, denoted as "-S" and "-L", respectively, i.e. DBLP-S, DBLP-L, and YT-S, YT-L. As shown in Table 6.1, the largest target graph, YOUTUBE, has over 1M nodes and 2M edges.

6.5.2 Baselines

We compare NSUBS against a series of baseline solvers whose source codes are provided by [115]: LDF [115], NLF [115], QUICKSI [118], GRAPHQL [119], TSO [120], CFL [121], CECI [122], and DP-ISO [123]. For HYBRID, we follow the recommendation by the authors of [115] by using DP-ISO for filtering, GRAPHQL for query node ordering, and LFTJ [124, 152] for local candidate computation. We report compare NSUBS with these state-of-the-art subgraph matching algorithms in Section 6.6.1.

We further evaluate NSUBS against several related works that tackle tasks similar to but not exactly subgraph matching: ISONET, NMATCH, and GLSEARCH. Note, without adaptations, these baselines cannot solve subgraph matching effectively. We discuss their shortcomings and compare them with NSUBS in detail in Sections 6.6.2 and 6.6.3.

Table 6.2: Average number of subgraph matchings found after 5 minutes on one graph pair in each dataset. Note, multiple subgraph matchings can exist for a single graph pair. For clarity and compactness, each result has been divided by 1000, i.e. each number is in the unit of 10^3. More details can be found in Section 6.6.1 and 6.6.3.

Method	BA	HPRD	Dblp-S	Dblp-L	Yt-S	Yt-L
LDF	1.21	0.69	4.82	1.51	2.73	1.75
NLF	1.29	0.37	4.26	1.06	2.52	1.06
GRAPHQL	1.42	1.24	5.23	2.55	2.73	1.23
TSO	1.41	0.50	4.36	1.36	2.57	1.24
CFL	1.56	0.18	5.70	1.13	2.84	0.90
DP-ISO	1.31	0.54	4.86	2.63	3.32	1.88
CECI	1.21	1.05	4.18	1.35	1.60	0.55
NMATCH	1.02	0.00	1.25	0.24	0.37	0.00
GLSEARCH	0.35	0.74	5.06	2.56	3.82	1.65
NSUBS	**1.95**	**4.27**	**8.90**	**4.29**	**7.27**	**5.01**

6.5.3 Experimental Setup

To examine the efficiency of each method and analyze the efficacy across time, we conduct the following evaluation. For each (q, G) pair, we record the time the method takes to find a solution, and accumulate the number of solved pairs across time. From $t = 0$ to $t = 300$ seconds, an increase at t indicates the method solves one graph pair at t. The earlier a method solves the graph pairs, the faster and better the method is.

In theory, given an infinite amount of time, every method adopting the backtracking search algorithm would be able to solve a graph pair. However, such assumption is not practical. Thus, the results in Section 6.6 examine the practical implication that under a reasonable amount of time budget, which method performs best at subgraph matching. Another observation is that the some baselines flatten towards the end of 5 minutes, indicating that they get stuck in unpromising search states that are unlikely to contain the solution, confirming the severity of the aforementioned challenges of solution and reward sparsity.

Table 6.3: Number of valid sugbraph matchings returned by IsoNet and search-based approaches on the BA dataset. More details can be found in Section 6.6.2.

Method	$\lvert V_q \rvert = 8$	$\lvert V_q \rvert = 16$	$\lvert V_q \rvert = 32$	$\lvert V_q \rvert = 64$
IsoNet	40%	26%	4%	0%
NSubS	100%	100%	98%	48%

6.6 Experimental Results

We experimentally verify that NSubS achieves state-of-the-art performance on subgraph matching by (1) adopting the "learning to search" paradigm which guides existing search algorithms to quickly find solutions, (2) using a global policy encoder, GPE, that is more suited for highly efficient search frameworks, and (3) incorporating subgraph matching properties through a contrastive geometrically-regularized policy gradient, CGR-Loss.

We show that NSubS achieves state-of-the-art performance in Section 6.6.1. We highlight the importance of using "learning to search" for subgraph matching in Section 6.6.2. We verify that GPE and CGR-Loss provide substantial performance improvement over both existing representation learning and "learning to search" methods in Section 6.6.3, 6.6.4, and 6.6.5. Finally, we provide further discussion on runtime analysis of NSubS in Section 6.6.7. We describe how we adapt NMatch and GLSearch in Section 6.6.3.

6.6.1 Outperforming State-of-the-Art Solvers

We first show NSubS drastically outperforms multiple strong combinatorial solver baselines, which is the state-of-the-art algorithm for subgraph matching. As shown in Figure 6.2, the performance bottleneck for subgraph matching on large graph pairs comes from an ineffective search policy rather than the search framework, which traditional solvers focus on. Due to the NP-Hard nature of subgraph matching, the search framework cannot hope to reduce the search space enough on large query and target graphs, such that naively enumerating node-node matchings could lead to a solution.

NSubS outperforms the state-of-the-art by applying the "learning to search" paradigm on the subgraph matching task, which allows NSubS to learn an effective policy that can

Table 6.4: NSubS Ablation Study on encoder and loss function design. More details can be found in Section 6.6.4, 6.6.5, and 6.6.6.

Loss Function	HPRD
NSubS (no-inter GMN)	34%
NSubS (no-inter SAGE)	20%
NSubS (dim = 8)	34%
NSubS ($\pi_\theta(a_t \mid s_t)$)	24%
NSubS (REINFORCE)	36%
NSubS (Contrastive Loss)	36%
NSubS (REINFORCE+GR)	36%
NSubS	50%

greedily arrive at a solution before exhausting the search space. This demonstrates the overall effectiveness of our model design and shows CGR-Loss will train an effective global policy network, GPE, on subgraph matching. We highlight these results are particularly impressive as the solver baselines run much faster than NSubS. Nonetheless, NSubS's smarter, albeit slower, policy network drastically improves the performance existing solvers by solving up to 3x more graph pairs. We provide further analysis on runtime in Section 6.6.7.

We observe that when the query graph size increase, all methods tend to show lower performance, which can be attributed to the exponentially growing search space. It is noteworthy that the survey chapter comparing existing solvers [115] uses query graphs up to 32 nodes, whereas we challenge all methods by testing on query graphs up to hundreds of nodes. The fact that NSubS is able to solve more graph pairs than baselines when the target graphs are large demonstrates good scalability of NSubS.

Given the nature of subgraph matching, we also evaluate the ability of each method to find as many solutions as possible. As Table 6.2 indicates, NSubS outperforms baseline methods on all datasets, which further supports the effectiveness of our overall model design and the "learning to search" paradigm as NSubS can not only ensure one solution is found, but also provide as many as and even more solutions compared against the faster baseline solvers.

6.6.2 Importance of Search-based Framework

We show NSubS drastically outperforms the state-of-the-art representation learning based approach to soft subgraph matching. Among representation learning based approaches, IsoNet [114] achieves state-of-the-art performance in ranking node-node and edge-edge pairs based on whether or not they appear in a true subgraph matching. IsoNet accomplishes this by asymmetrically computing edge-edge correspondence scores with a geometrically regularized loss function.

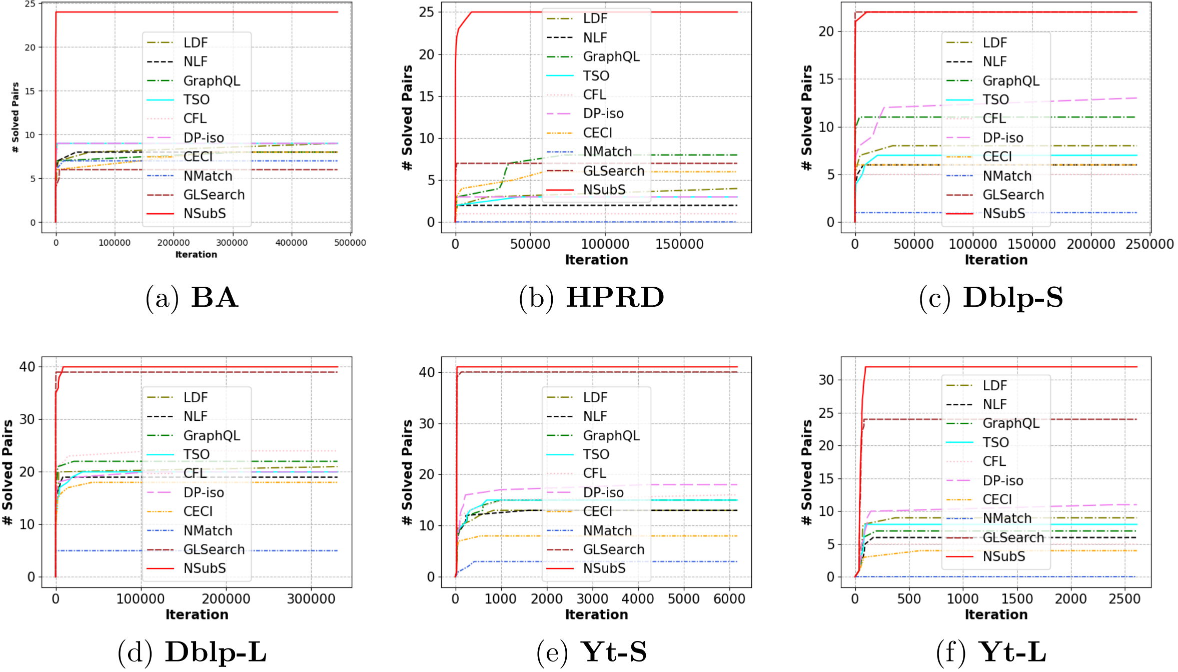
Note, IsoNet on its own cannot return an exact subgraph matching, only a soft edge-edge mapping. Thus, IsoNet proposes using the Hungarian algorithm to obtain a hard edge-edge mapping from its returned soft matching matrix, and show this approach can extract isomorphic subgraph matchings for small query and target graphs. We run the official IsoNet implementation[1] on our larger datasets and confirm this approach does not scale. As shown in Table 6.3, IsoNet [114] fails to query graphs with over 16 nodes on our smallest dataset in terms of edge count, BA. This is because the majority of subgraph matchings returned by running the Hungarian algorithm on IsoNet's soft edge-edge matching scores violate the isomorphism constraint and are hence invalid. Without properly intergrating representation learning models, such as IsoNet, under a rigorous search framework, they cannot guarantee the returned subgraph is isomorphic and hence cannot scale to large query and target graphs.

6.6.3 Outperforming GLSearch and NMatch

We now show by merging recent developments from representation learning and combinatorial search into the "learning to search" paradigm, we outperform existing works from both the representation learning and "learning to search" communities.

As shown in Section 6.6.2, representation learning approaches cannot scale without being integrated into a search framework. Because existing search frameworks iteratively match

[1]https://github.com/Indradyumna/ISONET



(a) **BA** (b) **HPRD** (c) **Dblp-S**

(d) **Dblp-L** (e) **Yt-S** (f) **Yt-L**

Figure 6.3: The growth in the number of input graph pairs where we find at least one solution with respect to iterations. More details can be found in Section 6.6.7.

node-node pairs, we compare NSubS against the state-of-the-art node representation learning model, NMatch, instead of IsoNet, which does edge-edge matching. Specifically, we adopt the same search framework as NSubS, but instead of running the policy network, we directly query a pretrained model provided by the official NMatch implementation[2] to rank input node-node pairs in the action space. The search policy selects the node-node pair with the highest rank first. We run the adapted NMatch in Figure 6.2 and Table 6.2.

For our "learning to search" baseline, we adapt the closely related GLSearch, which was designed for maximum common subgraph detection, for subgraph matching. This is done by simply putting GLSearch in the same search framework as NSubS, without modifying the model architecture or loss functions. Because, GLSearch also picks node-node pairs to match, it can be directly integrated into subgraph matching search frameworks. We run the adapted GLSearch in Figure 6.2 and Table 6.2. Note, we find GLSearch without this adaptation performs very poorly as it cannot take advantage of powerful subgraph matching

[2]https://github.com/snap-stanford/neural-subgraph-learning-GNN

search algorithms.

As shown in Figure 6.2 and Table 6.2, NSubS outperforms both the adapted NMatch and GLSearch models. Unlike NMatch, NSubS uses inter-graph message passing and trained using a reinforement learning loss, CGR-Loss, which in addition to geometrically encoding subgraph matching properties, considers both search dynamics and multiple ground truths. NSubS outperforms GLSearch as the overly complex state-dependent constraint encoder adopted by GLSearch is not needed for subgraph matching, hence NSubS's global encoder is both effective and easier to train. This agrees with our hypothesis that a global policy may be effective enough for highly efficient search algorithms, such as those for subgraph matching. Furthermore, NSubS uses a geometrically regularized RL loss that encodes subgraph matching properties, which GLSearch fails to consider. By leveraging advancements in representation learning, NSubS advances current approaches for "learning to search".

6.6.4 Effect of Global Policy Network

We verify our surprising claim that a simple global policy encoder can outperform a state-dependent encoder, denoted as $\pi_\theta(a_t|s_t)$ and detailed in Section 6.7.1.1. For this experiment, we use the exact same model and training setup as NSubS, but replace the global policy encoder with the state-dependent constraint encoder within the policy network. As seen in Table 6.4, this change indeed hurts the model performance. We believe this is due to the excessive search space pruning done by the backbone subgraph matching search algorithm, which allows even a simpler global policy to succeed. Thus, the inductive bias of our global policy encoder makes it outperform the state-dependent one.

6.6.5 Effect of CGR-Loss

To better analyze the effects of our proposed CGR-Loss, a contrastive geometrically-regularized RL loss, we compare against serveral reformulations. First, we compare CGR-Loss against standard REINFORCE [151] policy gradient loss. Next, we replace the contrastive learning component of CGR-Loss with a standard REINFORCE loss to form RE-

INFORCE+GR Loss. Finally, we remove geometric regularization from CGR-Loss to form Contrastive Loss. Apart from the loss function, we perform the exact same model and training setup as NSubS.

As seen in Table 6.4, we find our novel CGR-Loss both promotes improved representations in our policy network, by encoding subgraph properties, and improves training, by utilizing a contrastive learning approach to RL loss. Our results suggest both components of the CGR-Loss contribute equally to the NSubS's final performance.

6.6.6 Ablation Studies

We provide ablation studies on tuning the choice of graph neural network and dimension size. We first experiment with removing any inter-graph message passing from our encoder. Specifically, we first test our GPE encoder without the intergraph message passing layer and name this "no-inter GMN". Next, we replace GPE encoder with a GraphSAGE[8] model with equal layer and dimension size. Finally, we decrese the number of dimensions in GPE from 16 to 8.

As shown in Table 6.4, we find that removing the inter-graph message passing layer greatly hurts performance of the downstream model which confirms that GPE should include inter-graph message passing. We also show our choice of intragraph message passing layer is optimal. With smaller dimension size, GPE lacks the model capacity to learn good node embeddings.

6.6.7 Performance Across Iterations

We show the number of solved pairs across search iterations in Figure 6.3, similar to Figure 6.2 except for the x-axis which is replaced with search iterations. Note, as NSubS is implemented in Python and the baseline solvers can run for more iterations given the same runtime. These results suggest if NSubS can be accelerated by hardware acceleration, it can achieve even greater performance improvement.

6.7 Comparison with Related Works

6.7.1 Comparison to GLSearch

NSubS is different from GLSearch in several ways. First, NSubS adopts the policy network framework instead of deep value networks as it is more computationally efficient to compute. Second, NSubS uses a global policy network unlike the GLSearch's state-dependent value network, which we show is less effective within highly efficient search frameworks such as subgraph matching search algorithms. Lastly, GLSearch uses standard DQN training loss, whereas NSubS adopts CGR-Loss which uses contrastive learning and geometric regularization under a RL framework.

6.7.1.1 GLSearch-Style Policy Encoder $\pi_\theta(a_t|s_t)$

Following GLSearch, we design an encoder that captures state-dependent constraints. GLSearch captures this information by first running standard intra-graph message passing to form a set of state-independent node embeddings, $\{\boldsymbol{h}'_u\}_{u\in V_q}$ and $\{\boldsymbol{h}'_v\}_{v\in V_G}$, then aggregating node embeddings by bidomains, which intuitively capture a superset of matchable nodes pairs between 2 graphs by pruning provably unmatchable nodes pairs conditioned on the search state, s_t. This data structure can be represented by a set of query-to-target graph matchings, $\mathcal{M}(s_t) = \{V_q^{(b)} \rightarrow V_G^{(b)}|s_t\}_{b=1}^{|\mathcal{M}|}$, where each query-to-target matching defines a subset of query graph nodes, $V_q^{(b)} \subseteq V_q$ that can match a subset of target graph nodes, $V_G^{(b)} \subseteq V_G$. GLSearch computes embeddings for each query-to-target matching, $\mathbf{z}_b$ through aggregating node embeddings in bidomains, which are then used by the value decoder, $\mathbb{V}(s_t) = f(\{\mathbf{z}_b\}_{b=1}^{|\mathcal{M}|})$.

To construct a state-dependent constraint-based encoder, we make an analogy between the partially matched subgraph, M, local candidate set, $\mathcal{A}_{u_t}$, global candidate set, C, data structures for subgraph matching and the bidomain data structure for MCS. We define the concept of smallest set of matchable target nodes to a given query node, u, as (1) the currently matched target graph node, $M[u]$, if said node is in the partially matched subgraph, (2) the

local candidate set, $\mathcal{A}_u$, if said node selected by the local candidate set, and (3) the global candidate set, $C[u]$, if the said node is not currently matched and not selected by the local candidate set, i.e. the node will be matched in a future state in search. Thus, we can from the set of query-to-target graph matchings, $\mathcal{M}$, by taking the union of (1), (2), and (3). The resulting set of query-to-target graph matching is state dependent as the partially matched subgraph and local candidate set are both constraints conditioned on the current state, s_t.

The $\pi_\theta(a_t|s_t)$ encoder can be described as follows:

$$\boldsymbol{Z}_M = \{AGG^{(1)}_{v \in M[u]}\{f^{(1)}_{msg}(\boldsymbol{h}'_u, \boldsymbol{h}'_v)\}\}_{u \in M} \tag{6.8}$$

$$\boldsymbol{Z}_{\mathcal{A}} = \{AGG^{(1)}_{v \in \mathcal{A}_{u_t}}\{f^{(1)}_{msg}(\boldsymbol{h}'_u, \boldsymbol{h}'_v)\}\}_{u = u_t} \tag{6.9}$$

$$\boldsymbol{Z}_C = \{AGG^{(1)}_{v \in C[u]}\{f^{(1)}_{msg}(\boldsymbol{h}'_u, \boldsymbol{h}'_v)\}\}_{\substack{u \in V_q \\ u \neq u_t \\ u \notin M}} \tag{6.10}$$

$$\boldsymbol{Z} = \boldsymbol{Z}_M \cup \boldsymbol{Z}_{\mathcal{A}} \cup \boldsymbol{Z}_C \tag{6.11}$$

$$\boldsymbol{h}_v = \{AGG^{(2)}_{\boldsymbol{z} \in \boldsymbol{Z}}\{f^{(2)}_{msg}(\boldsymbol{h}'_v, \boldsymbol{z})\}\} \tag{6.12}$$

$$\boldsymbol{h}_u = \{AGG^{(2)}_{\boldsymbol{z} \in \boldsymbol{Z}}\{f^{(2)}_{msg}(\boldsymbol{h}'_u, \boldsymbol{z})\}\} \tag{6.13}$$

Our state-dependent constraint encoder, SDCE, follows GLSearch by first forming state-independent node embeddings, $\{\boldsymbol{h}'_u\}_{u \in V_q}$ and $\{\boldsymbol{h}'_v\}_{v \in V_G}$, using a standard intra-graph message passing graph neural network, $g(\cdot)$. Then, we inject state information by aggregating embeddings for each query-to-target matching through message passing, detailed in Equation 6.13, where $\mathbf{Z}$ denotes the aggregated embeddings. Then we inject the constraint embeddings back into the node embeddings through message passing. This policy network encoder, $\mathcal{E}^{(sdc)}_\theta$, implicitly captures the state-dependent constraints of the problem. Note, unlike GLSearch, we adopt the policy network framework, $\pi_\theta(a_t|s_t)$, as it is more computationally efficient than a value network.

6.7.2 Comparison with Dmpnn

Although Dmpnn [135] provides soft node scores in the target graph, it does not provide discrete subgraph matching. A naive solution is thresholding the soft scores to obtain a discrete subset of target graph nodes, but this cannot guarantee the extracted nodes are isomorphic to the query due to the lack of any way to guarantee that. Hence, a search framework is required to extract the discrete subgraph matching.

6.7.3 Comparison with RL-QVO

RL-QVO is orthogonal to NSubS, proposing a better query vertex-ordering scheme for subgraph matching. For large graphs, target-vertex ordering becomes the bottleneck instead of query vertex-ordering, as enumerating all target vertices for each query vertex becomes infeasible by the NP-Hard nature of subgraph matching.

Unlike RL-QVO, which uses random target vertex-ordering, NSubS learns a policy network to perform target vertex-ordering. In addition to RL-QVO 's single graph encoder, NSubS considers the query to target graph matching Unlike RL-QVO 's simple PPO loss, NSubS learns geometric node representations with lookahead loss.

6.8 Conclusion

In this chapter, we tackle the challenging and important task of subgraph matching, and present a new method, NSubS, for efficient and effective exact subgraph matching. NSubS effectively applies the "learning-to-search" paradigm to drastically outperform the state-of-the-art subgraph matching solvers. Particularly, we argue simple global policy network can perform surprisingly well on highly efficient search frameworks such as those used in subgraph matching. Inspired by recent works in representation learning for subgraph matching, we propose a contrastive geometrically regularized RL objective to unify loss functions from both reinforcement and representation learning. We experimentally show the utility of the proposed NSubS method on the important subgraph matching task. Specifically, NSubS

is able to solve more graph pairs than several existing subgraph matching solvers on one synthetic dataset and four large real-world datasets.

CHAPTER 7

ProgSG: Cross-Modality Representation Learning for Programs in Electronic Design Automation

7.1 Introduction

Over the past decades, there has been an increasing need for specialized computing systems that can speed up particular applications. As a result, domain-specific accelerators (DSAs) such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs) have emerged. DSAs are developed to enhance performance and energy efficiency by exploiting the characteristics of specific workloads. The Tensor Processing Unit (TPU) [153], Google's custom-designed ASIC, is a prominent example of a DSA that has been optimized for machine learning workloads and can deliver orders-of-magnitude faster performance and better energy efficiency than a CPU or GPU.

However, designing DSAs is more challenging than general-purpose hardware like CPUs and GPUs [154, 155, 156] because they are usually designed using hardware description languages (HDLs) at the register-transfer level (RTL) using Verilog and VHDL, which are only familiar to circuit designers. To address this challenge, high-level synthesis (HLS) [157, 158] was introduced. HLS raises the level of abstraction to C/ C++/ OpenCL/ SystemC, allowing designers to describe a high-level behavioral representation of their design rather than the transition of data in RTL.

Although HLS tools increase the level of design abstraction, they still require a significant amount of hardware design knowledge expressed through synthesis directives in the form of pragmas. These pragmas specify how computation is parallelized and/or pipelined, how

data is cached, how memory buffers are partitioned, etc. However, such architecture-specific optimization can usually only be done by hardware programmers and is beyond the reach of an average software programmer. Our objective is to automate and speed up the process of optimizing an integrated circuit (IC) design to make it more accessible for average software programmers.

There is a growing trend to apply machine learning to IC design automation [159]. For example, researchers have developed learning-based methods to predict the quality of HLS designs [160], explore the HLS design space intelligently for optimal resource allocation [161], etc. These methods fundamentally rely on an informative representation of an input design in order for the machine learning model to achieve good performance. We, therefore, focus on the fundamental task of representation learning for IC designs defined with HLS C/C++ (in short, we call them HLS designs) which are annotated with compiler directives/pragmas in this chapter. Specifically, we aim to design an encoder-decoder framework where the encoder provides powerful representations for the input HLS designs so that the designs' quality can be predicted accurately.

One limitation of existing representation learning methods for HLS designs is that they usually restrict to only using either the source code or the compiler-derived representation, but not both. For example, GNN-DSE [160] compiles the HLS code representing into assembly code and further transforms it into a graph derived from control data flow graph (CDFG), named ProGraML [162] and encodes it by a graph neural network (GNN). Meanwhile, [163, 164, 165, 166] directly apply a large language model (LLM) to the source code to obtain the representations that catch the semantics of the program.

However, we argue that only utilizing either one of the modalities is not good enough to obtain a comprehensive program representation. One one hand, the CDFG modality tends to ignore the semantic information in the source code, which is helpful to understand a program's behavior. For example, in CDFG, it is difficult for GNN to understand the functionality of a call site, particularly to standard libraries like `glibc`. What is worse, a statement such as "`A[i][j] *= beta;`" would be converted to a relatively large and complex

subgraph in the CDFG causing difficulty for the model to understand the semantic meaning. On the other hand, two source code programs with similar semantics and functionalities could have significantly different latency and communication requirements, whereas the lower-level control-flow structure of the programs can help.

In addition, the compiler translates the source code into assembly code and thus also provides fine-grained alignment information between source code tokens and CDFG nodes. None of the existing works for HLS design representation learning leverages such information. Last but not least, the labeled data is scarce due to the lengthy synthesis time for the HLS tools, and how to leverage unlabeled designs for our task remains a challenge.

In this chapter, we propose PROGSG (*Prog*ram representation learning combining *S*ource sequence and assembly code *G*raph) for a unified representation learning methods leveraging both the source code modality and the assembly code modality, with pre-training performed on both modalities. Specifically, PROGSG is an encoder-decoder framework allowing the naturally derived assembly instructions to be encoded together with the source code. To handle the interaction between source code and assembly code, we propose two novelties in the architecture: (1) An attention-summary architecture for coarse interaction between the two modalities; (2) A fine-grained node-to-token message passing mechanism to enable further collaboration between the two modalities. We also propose a novel pre-training method based on predicting node-node relationships for compiler analysis tasks for the GNN encoder to address the label scarcity issue.

Our contributions in this chapter can be summarized as follows:

- We are among the first to tackle the emerging problem of hardware design automation with the recently popular transformer model, and achieve state-of-the-art performance on two FPGA benchmark datadets.
- We propose a novel neural network architecture based on multi-modality learning, with graph summary augmentation and node-token alignment to enable collaboration between the two modalities efficiently and effectively.

- We propose a novel pretraining method for GNNs tailored for computer programs via CDFGs. To the best of our knowledge, we are among the first to explore pre-trained GNNs for assembly code represented as CDFGs.

7.2 Preliminaries

7.2.1 Background Introduction

The goal of this chapter is to train a neural network model to effectively predict the quality of the HLS design D specified by a C program P with a specific optimization pragma design ζ. The quality of a design, $\boldsymbol{y}$, is defined as a function of its performance, which is measured by its latency in cycle counts, and its area/resource utilization, such as the usage of block RAM (BRAM), digital signal processing (DSP), flip-flop (FF), and lookup-tables (LUT), which are the fundamental building blocks for implementing digital logic circuits in FPGA designs.
In this work, we specifically consider the optimization pragmas of the Merlin Compiler, a tool widely used for HLS designs. The Merlin Compiler provides three types of optimization pragmas, namely `PIPELINE`, `PARALLEL`, and `TILE` to define the desired microarchitecture [154].

```
void kernel_mvt(double x1[120], double x2[120], double
    y_1[120], double y_2[120], double A[120][120]) {
  int i, j;
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
  for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x1
    FACTOR=auto{__PARA__L2}
    for (j = 0; j < 120; j++) {
      x1[i] += A[i][j] * y_1[j];
    }}
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
  for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x2
    FACTOR=auto{__PARA__L3}
    for (j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
    }}}
```

Code 7.1: Code snippet of the MVT kernel (Matrix Vector Product and Transpose) with its 8 pragmas starting with "`#pragma`".

As shown in Code 7.1, these pragmas can be applied at the loop level and offer control over the type of pipelining, the parallelization factor, and the amount of data caching. If

designed by setting 3 out of the 8 pragmas properly to non-default parameters for proper parallelizing and pipelining the computation, the resulting accelerator is 10× faster than a single-core CPU. However, without any pragmas insertion, the resulting hardware is 13× slower than a CPU.

Table 7.1 summarizes the parameter space of these pragmas. For a given program P, any change in the option of any of the pragmas ζ results in a different design D with a unique microarchitecture. The "fg" option in pipelining refers to the case where all the inner loops are unrolled (parallelized with separate logic) and each parallel unit is pipelined. The "cg" option, on the other hand, results in coarse-grained processing elements (PEs) that are pipelined together. For example, it can create pipelined load-compute-store units. The `PARALLEL` and `TILE` pragma take numeric values that determine the degree of parallelization and loop tiling, respectively.

Table 7.1: Target pragmas with their options.

Pragma	Parameter Name	Parameter Space	Example Parameters
`PARALLEL`	factor	¡int¿	4, 8
`PIPELINE`	mode	"cg", "fg", off	'flatten' resulting in the "fg" mode
`TILE`	factor	¡int¿	2, 4

7.2.2 Problem Definitions

Task Definition The model $f(D_i)$ predicts $\hat{\boldsymbol{y}}_i$ for a given input design $D_i = (P_i, \zeta_i)$. The model is trained on a set of labeled designs $\mathcal{D}^{(\text{train})} = \{(D_i, \boldsymbol{y}_i)\}_{i=1}^{N}$ coming from a variety of programs with their pragmas[1].

Source Code and CDFG In this chapter, we follow GNN-DSE's [160] approach to compile the source code $C = (c_1, \ldots, c_I)$ where I denotes the sequence length via LLVM [167], and transform the assembly code further into a CDFG[2] denoted as G. $G = (V, E, l_V, l_E)$

[1]Due to the combinatotrial nature of design space, we follow [160] to exclude invalid designs from the training and testing set.

[2]Strictly speaking, it is a ProGraML graph with additional call relations between instructions and operands and with additional pragma nodes, but for convenience and without loss of generality, we use the term "CDFG" in this chapter.

where V denotes the node set, E denotes the edge set, and l_V (l_E) denotes the node (edge) labeling function that maps each node (edge) into a list of node (edge) attributes. The pragmas ζ are placed in the source code and can be transformed into the CDFG as nodes. Therefore, a design $D = (P, \zeta)$ can also be represented as a source code sequence C and a CDFG G, i.e., $D = (C, G)$.

Cross-Modality Alignment The LLVM compiler outputs the line and column numbers associated with some of the assembly instructions, which we use to construct a one-to-many alignment M mapping $v_k \in V$ to a set of source code tokens on that line $\{c_j\}$, i.e. $M(v_k) = \{c_j\}$. The reverse mapping M^{-1} maps a token c_j to nodes $\{v_k\}$ such that $c_j \in M(v_k)$. More details can be found in the supplementary material.

7.2.3 Related Work

Machine Learning for Electronic Design Automation Machine learning (ML) for electronic design automation (EDA) is a rising research area [159] with applications at various stages and levels of hardware design, such as design verification [168, 169], high-level synthesis (HLS) [160, 161, 170, 171], circuit design [172, 173, 174, 175], etc. In this work, we focus on how to represent HLS designs using both the source code and the assembly code under the design quality regression task for FPGA designs. Many works represent the input design/circuit as graphs [160, 170, 172], yet we are among the first to combine both the source code sequence and the CDFG modalities.

Representation Learning for Programs Based on the modality of data, current methods can be divided into source-code-based methods and data-structure-based methods. Source-code-based methods [163, 164, 165, 176] apply language models [84, 177, 178] on source code to perform various types of tasks, such as CuBert [164], CodeBert [165], and codet5 [163]. However, the language models cannot capture the underlying structure of how a program runs, and hence they may not be good at predicting the program's (that can describe a hardware design) runtime performance. The data-structure-based methods obtain the program embeddings from the data structure that represents a program. For example,

code2vec [179] extracts a collection of paths from AST to form embeddings.

Multi-modal Learning with Transformers Transformers on multi-modality data have been a popular research topic due to the expressive power of transformer [180] and its recent success across multiple domains. Modality-wise, transformers have been applied to cross-modality tasks spanning across vision [181, 182, 183, 184, 185, 186], language [187, 188], source code [189], knowledge graphs [190, 191], audio [192, 193], point clouds [194], etc. In fact, multi-modal learning using transformers has recently been considered for a foundation model [195] even possible for achieving generalist artificial intelligence in certain domains [196, 197]. Many of these works consider the alignment between modalities to further enhance representation learning [183, 186], but it is important to note that our CDFG is a natural derivative of the source code, and the alignment information is given by the compiler and does not need to be inferred. Instead, we aim to maximally utilize the given alignment for the prediction task.

Graph Neural Networks and Pre-training Most of the GNNs[7, 8, 198] fit into the message-passing framework where node representations are iteratively updated by aggregating the features of their neighbor nodes with a differentiable aggregation function. Existing self-supervised learning methods [199] can be divided into two categories: contrastive methods [200, 201, 202, 203] and predictive methods [91, 199, 204, 205, 206, 207, 208]. To our knowledge, we are the first to explore pre-training GNNs with CDFGs.

7.3 Proposed Method: ProgSG

In this section, we first describe the overall encoder-decoder neural network architecture of our proposed ProgSG. Then, we focus on our novel encoder with a graph summary augmented sequence representation, and a fine-grained node-to-token alignment for the unification of and the maximum collaboration between the two modalities for predicting the design quality.

7.3.1 Overall Architecture for Design Quality Prediction

The overall model $f(D) = f(C, G)$ first encodes the input source code and CDFG into a set of embeddings representing the design, and then transforms the embeddings into a multilayer perceptron (MLP) based decoder into $\hat{\boldsymbol{y}}$. As seen next and shown in Figure 7.1, depending on the encoder architecture, different embeddings are sent to the decoder. The loss function measures the mean squared error (MSE) between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, i.e. $\mathcal{L}_{\text{task}} = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$.

ProgSG-ca: A simple concatenation-based encoder Since one modality is the source code sequence, and the other is the CDFG, it is natural to adopt a transformer model on C and a GNN model on G, which produce token representations $\{\boldsymbol{h}_j \in \mathbb{R}^d | j \in \{1, \dots, I\}\}$ via the transformer's self-attention mechanism, and node representations $\{\boldsymbol{h}_k \in \mathbb{R}^d | k \in \{1, \dots, |V|\}\}$ via the message passing mechanism, respectively. d denotes the embedding dimension. The starting token c_1's embedding is then taken as the source code summary, $\boldsymbol{h}_{\text{src}} \in \mathbb{R}^d$, and a graph-level aggregation can be performed on the node embeddings serving as the CDFG summary, $\boldsymbol{h}_{\text{cdfg}} \in \mathbb{R}^d$. The encoder simply outputs the concatenation of the two modalities summaries, $\text{concat}(\boldsymbol{h}_{\text{src}}, \boldsymbol{h}_{\text{cdfg}})$, and let the MLP-based decoder handle the interaction between the two modalities.

7.3.2 ProgSG-si: Graph-Summary-Augmented Sequence Representation

One limitation of the simple ProgSG-ca encoder is the shallow and ineffective modeling of the interaction between C and G. We propose a novel yet simple way to address the issue, by making the following observation: The transformer operates on the sequence of tokens $C = (c_1, \dots, c_I)$ by enabling every token to pay attention to every other token, and thus the embedding of the starting token c_1 which is initialized as a special token such as "`[cls]`", can be treated as the representation of the entire C. Mathematically, we can formulate

$$\boldsymbol{h}_{\text{src}} = \boldsymbol{H}_{\text{src}}[1] = g_{\text{att}}\big(c_1, \dots, c_I\big) = g_{\text{att}}\big(\boldsymbol{h}_{c_1}^{(0)}, \dots, \boldsymbol{h}_{c_I}^{(0)}\big), \tag{7.1}$$

where g_{att} denotes the multi-layer self-attention encoder of a transformer model, capturing the interaction between pairwise source code tokens, $\boldsymbol{h}_{c_j}^{(0)}$ denotes the j-th token's initial embedding[3], $\boldsymbol{H}_{\text{src}} \in \mathbb{R}^{I \times d}$ denotes the final token embeddings, and $\boldsymbol{H}_{\text{src}}[1]$ denotes the final embedding of c_1 which is treated as the sequence-level embedding, $\boldsymbol{h}_{\text{src}}$.

Based on the above observation, we propose to append the CDFG summary $\boldsymbol{h}_{\text{cdfg}}$ to the beginning of the sequence, forming an augmented sequence representation $C^{(\text{aug})} = (\boldsymbol{h}_{\text{cdfg}}, c_1, \ldots, c_I)$ as input to the transformer[4]. At the output level, both the zeroth and first entries in $\boldsymbol{H}_{\text{src}}$ can be taken as the final sequence-level embedding, i.e. $\boldsymbol{h}_{\text{src}} = \boldsymbol{H}_{\text{src}}[0:1]$. Overall,

$$\boldsymbol{h}_{\text{src}} = \boldsymbol{H}_{\text{src}}[0:1] = g_{\text{att}}\big(\boldsymbol{h}_{\text{cdfg}}, c_1, \ldots, c_I\big) = g_{\text{att}}\big(\boldsymbol{h}_{\text{cdfg}}, \boldsymbol{h}_{c_1}^{(0)}, \ldots, \boldsymbol{h}_{c_I}^{(0)}\big). \tag{7.2}$$

We denote such an encoder as ProgSG-si (Summary Interaction), since it first performs GNN with L_1 layers on G to obtain a summary, and let the expressive transformer of L_2 layers handle the pairwise attention between tokens and that summary embedding, which efficiently allows cross-modality interaction. In other words, the CDFG is treated as a derivative of the source code whose summary embedding is used to augment the source code sequence. During training, the gradients back-propagate through $\boldsymbol{h}_{\text{cdfg}}$ to the GNN, updating both the GNN and the transformer.

It is noteworthy that we are not among the first to propose such a concatenation-based method for interaction between a graph modality and a sequence modality. GreaseLM [209], for example, updates the summary of a graph and the summary of a sequence first and then apply GNN and transformer again to the original graph and sequence, i.e. $\boldsymbol{h}_{c_1}^{(l+1)}, \boldsymbol{h}_{\text{cdfg}}^{(l+1)} = \text{split}\Big(\text{MLP}\big(\text{concat}(\boldsymbol{h}_{c_1}^{(l)}, \boldsymbol{h}_{\text{cdfg}}^{(l)})\big)\Big)$, and applies the next transformer's self-attention layer to $(\boldsymbol{h}_{c_1}^{(l+1)}, \ldots, \boldsymbol{h}_{c_I}^{(l+1)})$. Our ProgSG-si in contrast explicitly appends the summary embedding

[3]This is usually implemented by looking up from a dictionary mapping each token ID into a d-dimensional learnable vector representing the initial embeddings.

[4]This is equivalent to augmenting the initial embedding lookup dictionary with a special token initialized as the output of a GNN.

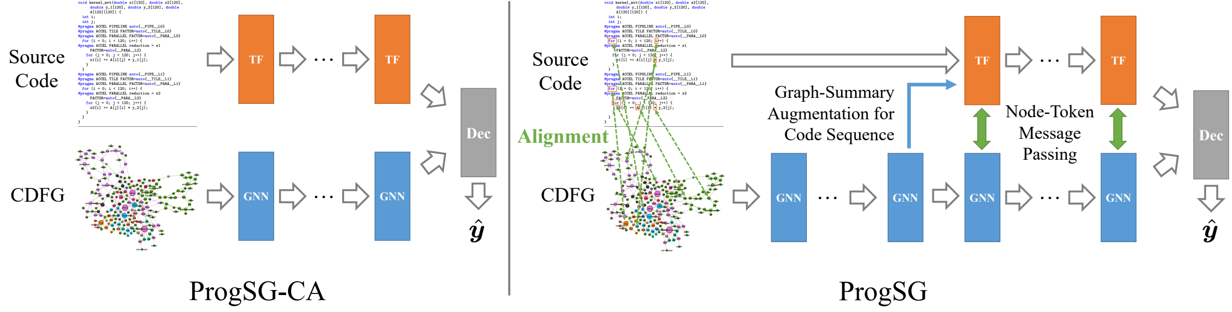


Figure 7.1: The overall diagrams of ProgSG-ca and ProgSG. "GNN", "TF", and "Dec" refer to Graph Neural Network Layer, Transformer Layer, and Decoder, respectively. For brevity, only some of the aligned token-node pairs derived from the assembly code are illustrated in dashed green arrows for ProgSG.

from the other modality as another token.

7.3.3 Full Model ProgSG: Leveraging Fine-grained Alignment

Since the CDFG is naturally derived from the source code but with more information about the control and flow data dependency information, we hypothesize that the alignment M between G and C contains useful information that helps the model obtain more expressive node embeddings $\boldsymbol{H}_{\text{cdfg}} \in \mathbb{R}^{|V| \times d}$ and token embeddings $\boldsymbol{H}_{\text{src}} \in \mathbb{R}^{I \times d}$.

Node-Token Message Passing To enable the message passing between the two modalities via M, we regard each aligned node-token pair as a special link, and propose the following cross-modality message passing mechanism inspired by message passing GNNs:

$$\begin{aligned} \boldsymbol{h}'_{v_k} &= \boldsymbol{h}_{v_k} + \text{MLP}_2\Big(\textstyle\sum_{c_j \in M(v_k)} \alpha_{k,j} \text{MLP}_1(\boldsymbol{h}_{c_j}) \Big), \\ \boldsymbol{h}'_{c_j} &= \boldsymbol{h}_{c_j} + \text{MLP}_2\Big(\textstyle\sum_{v_k \in M^{-1}(c_j)} \alpha_{j,k} \text{MLP}_1(\boldsymbol{h}_{v_k}) \Big), \end{aligned} \tag{7.3}$$

where the attention coefficients are computed via a dot product attention with learnable weight matrices $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d \times d}$, $\alpha_{k,j} = \text{softmax}\left(\frac{(\boldsymbol{W}_1 \boldsymbol{h}_{v_k})^\top (\boldsymbol{W}_2 \boldsymbol{k}_{c_j})}{\sqrt{d}} \right)$. Such cross-modality interaction enables ***fine-grained*** interaction between the two modalities so that more informative embeddings for the final prediction task can be generated. As an additional benefit, the interaction step has a time complexity that is linear with respect to the number of matched node-token pairs.

To allow ***deep*** cross-modality interaction, we perform the above node-token message passing L_2 times where L_2 is the number of transformer layers, e.g. 6 for the pre-trained CODET5 model used in our experiments. In each of the sequentially stacked L_2 layers, PROGSG performs the self-attention encoder on $C^{(aug)}$, executes GNN on G, followed by the node-token interaction.

7.3.4 Pretraining GNNs for CDFGs

A remaining challenge is the scarcity of the labelled data as generating ground-truth targets with HLS simulator is slow. Therefore, we want to utilize pre-training tasks to alleviate this issue. Although there exist many self-supervised tasks proposed to facilitate the training of GNN, they are too general, and we want to have tasks that could teach GNN the knowledge useful to our particular design quality regression task. Therefore, we propose to use data flow analyses as the self-supervised learning tasks. Data flow analysis is at the heart of modern compiler technology [162], where the capability of solving these tasks requires GNN to extract important information from a program's underlying structure. In addition, we can easily obtain a large set of labeled data to pre-train the GNN with non-ML techniques.

In particular, we select four data analyses tasks: (1) control reachability, (2) dominators, (3) data dependencies, and (4) liveness. The definitions of these tasks are given in the supplementary material. These tasks cover a full range of forward and backward analyses, and control and data analyses. In addition, these tasks focus on predicting the relationship between two nodes in a CDFG. Such node-level tasks help the GNN to learn meaningful node embeddings, which is the foundation of generating good graph embeddings.

Each task can be formulated as a binary classification problem. Therefore, given a pair of nodes v_i, v_j and a label y_{ij} which is a binary label indicating if the nodes have a particular relationship, we use the binary focal loss [210] to optimize the parameters.

$$\mathcal{L}_{\text{focal}} = -y_{ij}(1 - p(v_i, v_j))^{\beta} \log(p(v_i, v_j)) - (1 - y_{ij})(p(v_i, v_j))^{\beta} \log(1 - p(v_i, v_j)) \tag{7.4}$$

where $p(v_i, v_j)$ is the predicted probability that $y_{ij} = 1$ and $\beta \geq 0$ is a hyper-parameter. Focal loss is a good replacement for cross-entropy loss when the numbers of positive and negative samples are imbalanced. The modulated term allows the model to focus on hard misclassified examples.

Normally after pre-training, we would directly fine-tune the pre-trained GNN for the downstream task. However, we cannot do that because the graph schemas in the pre-training dataset and the fine-tuning dataset are different. In particular, the pre-training dataset does not contain any pragma nodes, which is important for predicting the quality of the HLS design. Therefore, instead of directly fine-tune the pre-trained model, we propose to use the pre-trained node embeddings as guidance to train a new (target) GNN for the downstream task.

Specifically, given a graph with pragma nodes, denoted as G, we would generate a corresponding graph without pragma nodes, denoted as G'. Then for a node v that appears in both G and G', we would compute its embedding in G' with the pre-trained GNN and compute its embedding in G with the GNN to be trained. Then, we would maximize the cosine similarity between the two embeddings with the following loss $\mathcal{L}_{\text{guide}} = 1 - \cos\langle g_{\text{cont}}(\boldsymbol{h}_{v,G}), \boldsymbol{h}_{v,G'}\rangle$ where g_{cont} is a continuous function (e.g., MLP, identity function). In this way, the target GNN would learn how to extract useful node-level information from the pre-trained GNN, which would in turn improve the quality of graph-level embeddings.

7.4 Experiments

We evaluate ProgSG on two sets of FPGA benchmark designs synthesized using two different versions of AMD/Xilinx HLS tools: SDx 2018.3 (v1) and Vitis 2020.2 (v2) (to show the robustness of our methodology). We compare ProgSG against six baselines: (1) code2vec [179] only encoding the source code level information represented as abstract syntax trees; (2) codeT5 [163] only encoding the source code sequence; (3) gnn-dse [160] only encoding the CDFG; (4) GreaseLM [209] leveraging both modalities without fine-grained

interaction; (5) ProgSG-ca which is a simple concatenation of the summary representations described in Section 7.3.1; (6) ProgSG-si which combines the two modalities without fine-grained interaction in Section 7.3.2.

7.4.1 Dataset and Evaluation Protocol

For the purpose of this study, we assembled a database of medium-complexity kernels that function as fundamental building blocks for larger applications. We selected a total of 40 kernels from two well-known benchmark suites, namely, the MachSuite benchmark [211] and the Polyhedral benchmark suite (Polybench) [212]. The kernels in the database were chosen to have a broad range of computation intensities, including linear algebra operations on matrices and vectors (e.g., BLAS kernels), data mining kernels (e.g., correlation and covariance), stencil operations, encryption, and a dynamic programming application.

Two sets of databases are generated by using different HLS tools to implement the design. These tools employ different optimization heuristics in generating the microarchitecture which impacts the final design's quality. Specifically, we used two distinct AMD/Xilinx HLS tools, namely, SDAccel SDx 2018.3 (v1) [213] and Vitis 2020.2 (v2) [214], with the Xilinx Alveo U200 as the target FPGA and a working frequency of 250MHz to generate the labels. For each design point, we recorded the `latency` in terms of cycle counts, as well as the resource utilization for `DSP`, `BRAM`, `LUT`, and `FF`. The statistics of the datasets are presented in Table 7.2. These datasets will be available upon chapter acceptance.

When evaluating, we test the model's ability to predict the quality of designs not only from the same Ps as in $\mathcal{D}^{(\text{train})}$, which we denote as the "transductive" setting (with the training, validation, and testing ratio being 70:15:15 and the root mean square error (RMSE) reported), but also from six unseen Ps that are not in $\mathcal{D}^{(\text{train})}$, which we denote as the "inductive" setting. For each unseen kernel, we perform an adaptation process which consists of an additional 10 steps of gradient updates of the trained model on 20 randomly labeled designs from that new program. Such adaptation is performed 5 times for each unseen kernel so that both the average and standard deviation of the RMSEs are reported for each kernel.

Table 7.2: Dataset statistics. "**#D**", "**#P**", "**A#P**", "**A#T**", "**A#N**", "**A#E**", and "**A#MP**" denote "# designs", "# programs", "avg # pragmas per program", "avg # tokens per program", "avg # nodes per program's CDFG", "avg # edges per program's CDFG", and "avg # matched node-token pairs", respectively.

Dataset	**#D**	**#P**	**A#P**	**A#T**	**A#N**	**A#E**	**A#MP**
SDx 2018.3 (v1)	8481	35	7.9	525.7	365.6	588.0	5859.1
Vitis 2020.2 (v2)	4337	26	7.3	530.5	328.1	525.0	3689.3

7.4.2 Model Setup and Hyperparameters

We follow [160] to generate the CDFGs. We adopt $L_1 = 8$ layers of TransformerConv [215] with a jumping knowledge network [216] as the final node embedding aggregation method. The embedding dimension $d = 512$. For the source code, we use codet5 [163] with $L_2 = 6$ layers to embed the source code[5]. AutoDSE defines a variable for each pragma, as shown in Code 7.1, that is a placeholder for the option of the pragma. Since the pragmas ζ must be reflected in the input source code, for each design, we add the pragma options to their respective variables, e.g. we change "`__PARA__L0__`" to "`__PARA__L0=1`", "`__PIPE__L2`" to "`__PIPE__L2=flatten`", etc. We set the maximum number of tokens to 64 for the tokenizer, and chunk each source code into multiple subsequences to handle the long input source code sequence. The summaries of all subsequences are aggregated into the final representation for the decoder. We report the full hyperparameters in the supplementary material.

7.4.3 Experimental Results

The overall results are shown in Table 7.3. In general, all the methods perform relatively better on the tranductive setting where the testing designs come from the seen programs. For the inductive setting, we observe the proposed full model ProgSG achieves the lowest overall error on both benchmark datasets. It is noteworthy that single-modality models including code2vec, codet5, and gnn-dse tend to perform poorly especially under the inductive setting (e.g codet5), suggesting that in order to generalize well to new programs,

[5] Specifically, we use codet5-small from `https://huggingface.co/Salesforce/codet5-small` to initialize the transformer encoder for source code, and fine-tune the whole model.

a more expressive model leveraging both modalities with more information is a promising direction.

Further breakdown of the prediction RMSE over individual unseen programs is reported in Tables 7.4 and 7.5. The full model ProgSG achieves the best performance on 3 out of the 6 unseen kernels, yet the best performance is consistently achieved by one of the three proposed models, ProgSG-ca, ProgSG-si, and ProgSG.

Table 7.3: Prediction RMSE on SDx 2018.3 (v1) and Vitis 2020.2 (v2).

Method	**SDx 2018.3 (v1)**		**Vitis 2020.2 (v2)**	
	Transductive	**Inductive**	**Transductive**	**Inductive**
code2vec [179]	2.7983	2.3037	2.5045	1.6134
codet5 [163]	**0.5476**	1.4480	0.4539	1.1837
gnn-dse [160]	0.7539	1.4359	0.6547	1.1216
GreaseLM [209]	0.7291	1.3855	0.3953	0.9016
ProgSG-ca	0.5603	1.3983	0.4120	0.9483
ProgSG-si	0.5505	1.2906	0.4326	0.8743
ProgSG	0.6211	**1.2806**	**0.3607**	**0.8199**

Table 7.4: Result breakdown on SDx 2018.3 (v1) on individual test kernels.

Method	doitgen-r	fdtd-2d	gemm-n	jacobi-2d	stencil-3d	trmm-opt
code2vec	1.7386±0.09	3.3187±0.12	5.0147±0.29	1.7448±0.07	0.8258±0.01	1.1798±0.10
codet5	0.9551±0.19	2.7643±0.12	2.8094±0.17	1.3274±0.10	0.4112±0.04	0.4206±0.04
gnn-dse	0.6717±0.12	2.4411±0.18	2.9897±0.52	1.3927±0.04	0.6251±0.03	0.4949±0.04
GreaseLM	1.0585±0.22	2.6807±0.20	2.7789±0.17	1.0331±0.14	0.3957±0.18	0.3659±0.03
ProgSG-ca	**0.5039±0.06**	2.5850±0.06	2.9542±0.11	1.4131±0.17	0.4589±0.09	0.4745±0.01
ProgSG-si	0.9417±0.21	2.5809±0.11	**2.3494±0.27**	1.1207±0.12	0.4636±0.18	**0.2875±0.01**
ProgSG	1.0884±0.10	**2.3259±0.25**	2.4837±0.26	**1.0612±0.07**	**0.3794±0.15**	0.3450±0.04

Table 7.5: Result breakdown on Vitis 2020.2 (v2) on individual test programs.

Method	covariance	fdtd-2d-l	gemm-n	gemm-p-l	symm	trmm-opt
code2vec	1.7627±0.11	2.1019±0.13	3.0219±0.09	1.3791±0.06	0.4736±0.03	0.9409±0.09
codet5	1.0556±0.02	2.3255±0.20	2.5189±0.36	0.5128±0.14	0.1958±0.03	0.4937±0.02
gnn-dse	1.1919±0.10	1.8424±0.07	2.0186±0.37	0.7733±0.05	0.3517±0.05	0.5517±0.08
GreaseLM	0.9224±0.10	1.4758±0.10	1.6650±0.10	0.6519±0.04	0.1960±0.05	0.4984±0.04
ProgSG-ca	1.0345±0.03	1.7138±0.07	1.9541±0.08	0.6753±0.07	0.1975±0.04	0.4506±0.05
ProgSG-si	**0.7157±0.07**	**1.2832±0.13**	2.0861±0.19	0.5206±0.06	0.2865±0.02	**0.3540±0.04**
ProgSG	0.9039±0.06	1.3323±0.11	**1.6294±0.12**	**0.5078±0.01**	**0.1672±0.03**	0.3791±0.02

7.4.4 Attention Visualization

To better understand if the transformer model learns to attend tokens that are relevant to HLS pragma configurations, we visualize the average attention scores of some of the pragma-related tokens in the GEMM-N kernel for the transformer before and after fine-tuning on VITIS 2020.2 (V2) for our regression task. The result is shown in Figure 7.2. We can see that 11 out of 15 tokens have higher attention scores after fine-tuning, which suggests the transformer model does learn to attend to the pragma-related tokens, which are important to predicting the quality of an HLS pragma configuration, even though these tokens are not seen in its pre-training stage.

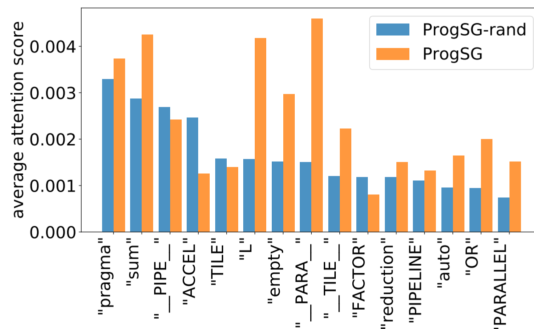


Figure 7.2: Bar plots of average attention scores of pragma-related tokens before (PROGSG-RAND) and after (PROGSG) being fine-tuned.

7.5 Conclusion

In this chapter, we propose PROGSG, a novel two-modality program representation learning method for IC design (defined with HLS C/C++) optimization. The key assumption is that there is critical information in both the source code modality and the assembly code modality, which must be captured jointly. To achieve that, we propose a graph-summary-augmented sequence representation for the source code transformer, a fine-grained alignment utilization method, and a novel pre-training method for the GNN encoder for the CDFG. Experiments on two benchmark datasets on FPGA design quality regression confirm the superiority of the proposed PROGSG over six baselines. PROGSG is currently tested on HLS designs for FPGA, yet we believe the core idea of using both modalities together with their alignment is general and can be adapted for other tasks.

CHAPTER 8

Conclusion

In this dissertation, we have introduced multiple works that focus on how to design neural network-based graph-level operators for graph similarity computation (Chapters 2 and 3), unsupervised graph-level embedding (Chapter 4), MCS detection (Chapter 5), subgraph matching (Chapter 6), and program representation learning for electronic designs (Chapter 7).

The contributions of this dissertation in advancing deep learning in neural network-based operator design for graph-level applications can be further summarized as follows:

- To design an effective and efficient neural network operator for graph similarity computation, careful design for representation learning is needed. Specifically, we employ a GNN-based encoder with a novel attention mechanism to produce one embedding per input graph, and a two-branch decoder that transforms the node and graph embeddings into the final output score denoting the graph-graph similarity.
- To tackle the more challenging NP-hard task of MCS detection, only using graph representation learning in the design of the neural network-based operator is not enough, as shown in the baseline models' results compared with GLSearch. In particular, we employ a branch-and-bound search algorithm as the backbone of the overall operator, and only use neural network in the DQN modeling which guides the search towards the MCS.
- For electronic design representation learning, the neural network operator we design incorporates both the CDFG modality that corresponds to the assembly code level

information and the source code modality which provides more semantic meaning for the neural network model. Additional pre-training and cross-modality interaction are also needed in order to obtain the best performance for the task of predicting the quality of these electronic designs.

In the end, this is by no means the end of the exploration neural network design for graph-level appplications. We also propose a few directions for future work:

- **More efficient and scalable operator for graph similarity and matching tasks.** As the size and amount of graph data continue to grow, the need for even faster and scalable neural network operators are needed. One promising direction is to consider hashing, which is commonly adopted in the database community, to provide speed-up with a potential trade-off in accuracy. For search-based methods such as GLSearch, we believe further leveraging optimization techniques in the search community is a promising direction. For example, randomized restart [217] combined with neural network can be a promising approach to accelerate the overall method.

- **Multi-task learning combining different yet related tasks.** Graph similarity and matching are inherently related albeit different tasks. Designing a more powerful neural network-based operator combining these inter-related tasks can be a promising direction, since these tasks may mutually benefit each other during the training process. For example, the GED metric and the MCS metric both are concerned with comparing two graphs, and designing a more expressive model that is jointly trained to perform both tasks could yield a more accurate model for both tasks.

- **Deeper understanding of task-specifics for proposing effective modeling.** We believe in general, it is a good idea to deeply understand the task itself in order to successfully design a neural operator, especially for graph-level tasks, because such tasks typically require some kind of comparing or reasoning over the entire graph instead of only focusing on a local part of the data. Using electronic design representation learning as an example, to further improve the accuracy of the proposed ProgSG model,

it can be beneficial to design pragma-specific neural operators that receive the local loop structure as well as the surrounding context that the pragma operates on as input. Such a design involves domain knowledge and is likely to yield more accurate prediction due to the more task-specialized way to design neural network operations.

[1] Xiaoli Wang, Xiaofeng Ding, Anthony KH Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *ICDE*, pages 210–221. IEEE, 2012. xii, 10, 18, 28, 29, 34

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999. xviii, 70

[3] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959. xviii, 70

[4] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998. xviii, 70

[5] Drahomira Herrmannova and Petr Knoth. An analysis of the microsoft academic graph. *D-lib Magazine*, 22(9/10):37, 2016. 1

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016. 1, 20, 27, 32, 35, 59

[7] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2016. 1, 11, 17, 27, 31, 35, 46, 49, 59, 65, 107

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017. 1, 27, 31, 36, 49, 54, 86, 96, 107

[9] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *ICLR*, 2018. 1, 27, 49, 65, 69

[10] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *ICLR*, 2019. 1, 31, 36, 49, 50, 86

[11] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection (2014). *URL http://snap. stanford. edu/data*, page 49, 2016. 1

[12] Guangyong Chen, Pengfei Chen, Chang-Yu Hsieh, Chee-Kong Lee, Benben Liao, Renjie Liao, Weiwen Liu, Jiezhong Qiu, Qiming Sun, Jie Tang, et al. Alchemy: A quantum chemistry dataset for benchmarking ai models. *arXiv preprint arXiv:1906.09427*, 2019. 1

[13] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *SIGKDD*, pages 1365–1374. ACM, 2015. 2, 18, 28, 46, 54, 57, 59

[14] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *NIPS*, 2018. 2, 27, 31, 32, 46, 59

[15] Xinyi Zhang and Lihui Chen. Capsule graph neural network. *ICLR*, 2019. 2, 46, 59

[16] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. *ICML*, 2019. 2

[17] Horst Bunke. What is the distance between graphs. *Bulletin of the EATCS*, 20:35–39, 1983. 2, 7, 28, 29, 47, 52

[18] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997. 2, 28

[19] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. *WSDM*, 2019. 3, 30, 32, 46, 47

[20] Yunsheng Bai, Hao Ding, Ken Gu, , Yizhou Sun, and Wei Wang. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. *AAAI*, 2020. 3

[21] Yunsheng Bai, Hao Ding, Yang Qiao, Agustin Marinovic, Ken Gu, Ting Chen, Yizhou Sun, and Wei Wang. Unsupervised inductive whole-graph embedding by preserving graph proximity. *IJCAI*, 2019. 3

[22] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. Glsearch: Maximum common subgraph detection via learning to search. In *ICML*, pages 588–598. PMLR, 2021. 3, 75, 76, 77, 79, 81, 82, 85

[23] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3-4):255–259, 1998. 7, 29, 47, 52, 60

[24] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009. 7, 10, 18, 28, 29

[25] David B Blumenthal and Johann Gamper. On the exact computation of the graph edit distance. *Pattern Recognition Letters*, 2018. 7, 19, 29

[26] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. A partition-based approach to structure similarity search. *PVLDB*, 7(3):169–180, 2013. 7, 10, 18, 28

[27] Yongjiang Liang and Peixiang Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *ICDE*, pages 783–794. IEEE, 2017. 7, 10, 18, 28, 52

[28] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *S+SSPR*, pages 163–172. Springer, 2006. 7, 8, 19, 20, 28, 55

[29] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959, 2009. 7, 8, 10, 19, 20, 28, 30, 36, 55

[30] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. Speeding up graph edit distance

computation through fast bipartite matching. In *GbRPR*, pages 102–111. Springer, 2011. 7, 8, 19, 20, 28, 30, 36, 55

[31] Sebastien Bougleux, Luc Brun, Vincenzo Carletti, Pasquale Foggia, Benoit Gaüzère, and Mario Vento. Graph edit distance as a quadratic assignment problem. *Pattern Recognition Letters*, 87:38–46, 2017. 7, 28

[32] Évariste Daller, Sébastien Bougleux, Benoit Gaüzère, and Luc Brun. Approximate graph edit distance by several local searches in parallel. In *ICPRAM*, 2018. 7, 28, 30

[33] Weiguo Zheng, Lei Zou, Xiang Lian, Dong Wang, and Dongyan Zhao. Graph similarity search with edit distance constraint in large graph databases. In *CIKM*, pages 1595–1600. ACM, 2013. 10, 18, 28

[34] Kaspar Riesen and Horst Bunke. Iam graph database repository for graph based pattern recognition and machine learning. In *S+SSPR*, pages 287–297. Springer, 2008. 10

[35] Andreas Fischer, Ching Y Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. A fast matching algorithm for graph-based handwriting recognition. In *GbRPR*, pages 194–203. Springer, 2013. 10

[36] Bing Xiao, Xinbo Gao, Dacheng Tao, and Xuelong Li. Hmm-based graph edit distance for image indexing. *International Journal of Imaging Systems and Technology*, 18(2-3):209–218, 2008. 10

[37] Kaspar Riesen, Sandro Emmenegger, and Horst Bunke. A novel software toolkit for graph edit distance computation. In *GbRPR*, pages 142–151. Springer, 2013. 10, 19, 33

[38] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *NIPS*, pages 926–934, 2013. 15

[39] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *NIPS*, pages 2042–2050, 2014. 16

[40] Hua He and Jimmy Lin. Pairwise word interaction modeling with deep neural networks for semantic similarity measurement. In *NAACL HLT*, pages 937–948, 2016. 16, 32

[41] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, pages 2014–2023, 2016. 16, 27, 37

[42] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. 19, 20, 37

[43] Roy Jonker and Anton Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. 19, 20, 37

[44] Rashid Jalal Qureshi, Jean-Yves Ramel, and Hubert Cardot. Graph based shapes representation and recognition. In *GbRPR*, pages 49–60. Springer, 2007. 19

[45] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015. 21, 70

[46] Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904. 21, 40

[47] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938. 21, 40, 55

[48] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *WSDM*, pages 459–467. ACM, 2018. 27

[49] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *SIGKDD*, pages 701–710. ACM, 2014. 27, 58

[50] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864. ACM, 2016. 27

[51] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015. 27, 46, 57, 58

[52] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *SIGKDD*, pages 1225–1234. ACM, 2016. 27

[53] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, pages 2224–2232, 2015. 27

[54] Xiaohan Zhao, Bo Zong, Ziyu Guan, Kai Zhang, and Wei Zhao. Substructure assembling network for graph classification. *AAAI*, 2018. 27

[55] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *Data Engineering Bulletin*, 2017. 27

[56] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016. 27

[57] John Boaz Lee, Ryan Rossi, and Xiangnan Kong. Graph classification using structural attention. In *SIGKDD*, pages 1666–1674. ACM, 2018. 27

[58] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. *ICLR*, 2018. 27

[59] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *ESWC*, pages 593–607. Springer, 2018. 27

[60] Tengfei Ma, Cao Xiao, Jiayu Zhou, and Fei Wang. Drug similarity integration through attentive multi-view graph auto-encoders. *IJCAI*, 2018. 27, 52

[61] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966. 28

[62] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *COLT*, pages 129–143. Springer, 2003. 28

[63] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *SIGKDD*, pages 158–167. ACM, 2004. 28

[64] Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Matching node embeddings for graph similarity. In *AAAI*, pages 2429–2435, 2017. 28, 30, 36

[65] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017. 29

[66] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636. ACM, 2003. 29

[67] Danai Koutra, Joshua T Vogelstein, and Christos Faloutsos. Deltacon: A principled massive-graph similarity function. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 162–170. SIAM, 2013. 29

[68] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. Learning graph distances with message passing neural networks. In *ICPR*, pages 2239–2244, 2018. 30, 32

[69] Sofia Ira Ktena, Sarah Parisot, Enzo Ferrante, Martin Rajchl, Matthew Lee, Ben Glocker, and Daniel Rueckert. Distance metric learning using graph convolutional net-

works: Application to functional brain networks. In *MICCAI*, pages 469–477. Springer, 2017. 30, 32, 47, 52

[70] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. *ICML*, 2019. 30, 32, 85, 86

[71] Andrei Zanfir and Cristian Sminchisescu. Deep learning of graph matching. In *CVPR*, pages 2684–2693, 2018. 30

[72] William L Hamilton, Payal Bajaj, Marinka Zitnik, Dan Jurafsky, and Jure Leskovec. Querying complex networks in vector space. *NIPS*, 2018. 32

[73] Frank Emmert-Streib, Matthias Dehmer, and Yongtang Shi. Fifty years of graph matching, network alignment and network comparison. *Information Sciences*, 346:180–197, 2016. 32

[74] Bhaskar Mitra, Fernando Diaz, and Nick Craswell. Learning to match using local and distributed representations of text for web search. In *WWW*, pages 1291–1299. International World Wide Web Conferences Steering Committee, 2017. 32

[75] John W Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002. 33

[76] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000. 36

[77] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *ICML*, pages 5694–5703, 2018. 37, 38

[78] Philippe Thévenaz, Thierry Blu, and Michael Unser. Image interpolation and resampling. *Handbook of medical imaging, processing and analysis*, 1(1):393–420, 2000. 38

[79] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *NeurIPS*, pages 5165–5175, 2018. 46

[80] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003. 46

[81] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifyingdeepwalk, line, pte, and node2vec. *WSDM*, 2018. 46, 54, 58

[82] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *KDD MLG Workshop*, 2017. 46, 54, 59

[83] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *NAACL*, 2018. 47

[84] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 47, 106

[85] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. 47

[86] Xifeng Yan, Philip S Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777. ACM, 2005. 52, 60

[87] Christopher KI Williams. On a connection between kernel pca and metric multidimensional scaling. In *Advances in neural information processing systems*, pages 675–681, 2001. 53

[88] Andreas Fischer, Ching Y Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. Approximation of graph edit distance based on hausdorff matching. *Pattern Recognition*, 48(2):331–343, 2015. 55

[89] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. 57

[90] Mark Heimann and Danai Koutra. On generalizing neural node embedding methods to multi-network problems. In *KDD MLG Workshop*, 2017. 58

[91] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *NIPS Workshop on Bayesian Deep Learning*, 2016. 59, 107

[92] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019. 59

[93] Younghee Park, Douglas S Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 39:419–430, 2013. 60

[94] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *2011 31st International Conference on Distributed Computing Systems*, pages 393–402. IEEE, 2011. 60

[95] Hans-Christian Ehrlich and Matthias Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011. 60

[96] Yan-li Liu, Chu-min Li, Hua Jiang, and Kun He. A learning based branch and bound for maximum common subgraph problems. *IJCAI*, 2019. 60, 63, 69, 75, 76, 81, 82

[97] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. *IJCAI*, 2017. 60, 63, 69

[98] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Learning combinatorial embedding networks for deep graph matching. *ICCV*, 2019. 60, 69

[99] Yunsheng Bai, Derek Xu, Ken Gu, Xueqing Wu, Agustin Marinovic, Christopher Ro, Yizhou Sun, and Wei Wang. Neural maximum common subgraph detection with guided subgraph extraction, 2020. 60, 69, 75, 81, 82

[100] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 61

[101] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *NeurIPS Deep Learning Workshop 2013*, 2013. 68

[102] Hongteng Xu, Dixin Luo, and Lawrence Carin. Scalable gromov-wasserstein learning for graph partitioning and matching. In *NeurIPS*, pages 3046–3056, 2019. 69

[103] Gabriel Peyré, Marco Cuturi, and Justin Solomon. Gromov-wasserstein averaging of kernel and distance matrices. In *ICML*, pages 2664–2672, 2016. 69

[104] Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. *arXiv preprint arXiv:1811.03508*, 2018. 69

[105] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 70, 75, 76, 85

[106] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012. 75

[107] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming subgraph isomorphism for rdf query processing. *arXiv preprint arXiv:1506.01973*, 2015. 75

[108] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203, 2009. 75

[109] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018. 75

[110] Hui Jiang, Yuxin Deng, and Ming Xu. Quantum circuit transformation based on subgraph isomorphism and tabu search. *arXiv e-prints*, pages arXiv–2104, 2021. 75

[111] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(7):1–13, 2013. 75

[112] Yunsheng Bai, Hao Ding, Yizhou Sun, and Wei Wang. Convolutional set matching for graph similarity. *NeurIPS Workshop on Relational Representation Learning Workshop*, 2018. 75, 76

[113] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020. 75, 76, 77, 78, 85, 86, 88

[114] Indradyumna Roy, Venkata Sai Baba Reddy Velugoti, Soumen Chakrabarti, and Abir De. Interpretable neural subgraph matching for graph retrieval. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8115–8123, 2022. 75, 76, 78, 85, 93

[115] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020. 75, 76, 77, 78, 80, 81, 82, 89, 92

[116] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. Reinforcement learning based query vertex ordering model for subgraph matching. *arXiv preprint arXiv:2201.11251*, 2022. 75, 76, 77, 78, 79, 81, 82

[117] Chang Liu, Runzhong Wang, Zetian Jiang, Junchi Yan, Lingxiao Huang, and Pinyan Lu. Revocable deep reinforcement learning with affinity regularization for outlier-robust graph matching. *arXiv preprint arXiv:2012.08950*, 2020. 75, 76, 81, 82

[118] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008. 76, 77, 89

[119] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418, 2008. 76, 77, 89

[120] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2013. 76, 77, 89

[121] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016. 76, 89

[122] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019. 76, 89

[123] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1429–1446, 2019. 76, 77, 89

[124] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020. 76, 89

[125] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*, pages 925–937, 2021. 78

[126] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015. 78

[127] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016. 78

[128] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112, 2019. 78

[129] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698, 2017. 78

[130] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Chunming Wu, and Shouling Ji. Hierarchical graph matching networks for deep graph similarity learning, 2020. 78

[131] Xin Liu and Yangqiu Song. Graph convolutional networks with dual message passing for subgraph isomorphism counting and matching. In *AAAI*, volume 36, pages 7594–7602, 2022. 78, 85

[132] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1959–1969, 2020. 78

[133] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. Can graph neural networks count substructures? *NeurIPS*, 33:10383–10395, 2020. 78

[134] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. Neural subgraph counting with wasserstein estimator. In *Proceedings of the 2022 International Conference on Management of Data*, pages 160–175, 2022. 78, 82

[135] Ye Yuan, Delong Ma, Aoqian Zhang, and Guoren Wang. Consistent subgraph matching over large graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2536–2548. IEEE, 2022. 78, 99

[136] Changjun Fan, Li Zeng, Yizhou Sun, and Yang-Yu Liu. Finding key players in complex networks through deep reinforcement learning. *Nature Machine Intelligence*, 2(6):317–324, 2020. 79

[137] Mingshuo Nie, Dongming Chen, and Dongqi Wang. Reinforcement learning on graphs: A survey. *arXiv e-prints*, pages arXiv–2204, 2022. 79

[138] Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. Superb: Speech processing universal performance benchmark. *arXiv preprint arXiv:2105.01051*, 2021. 84

[139] Priya Goyal, Dhruv Mahajan, Abhinav Gupta, and Ishan Misra. Scaling and benchmarking self-supervised visual representation learning. In *Proceedings of the ieee/cvf International Conference on computer vision*, pages 6391–6400, 2019. 84

[140] Linus Ericsson, Henry Gouk, and Timothy M Hospedales. How well do self-supervised models transfer? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5414–5423, 2021. 84

[141] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020. 84

[142] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020. 84

[143] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox. *arXiv preprint arXiv:1909.07930*, 2019. 84

[144] Matteo Hessel, Hado van Hasselt, Joseph Modayil, and David Silver. On inductive biases in deep reinforcement learning. *arXiv preprint arXiv:1907.02908*, 2019. 85

[145] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. 85

[146] Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. In search of the real inductive bias: On the role of implicit regularization in deep learning. *arXiv preprint arXiv:1412.6614*, 2014. 85

[147] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021. 85

[148] Eun-Sol Kim, Woo Young Kang, Kyoung-Woon On, Yu-Jung Heo, and Byoung-Tak Zhang. Hypergraph attention networks for multimodal learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14581–14590, 2020. 85

[149] Yupeng Hou, Binbin Hu, Wayne Xin Zhao, Zhiqiang Zhang, Jun Zhou, and Ji-Rong Wen. Neural graph matching for pre-training graph neural networks. In *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*, pages 172–180. SIAM, 2022. 85

[150] Can Qin, Handong Zhao, Lichen Wang, Huan Wang, Yulun Zhang, and Yun Fu. Slow learning and fast inference: Efficient graph similarity computation via knowledge distillation. *NeurIPS*, 34:14110–14121, 2021. 85

[151] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992. 87, 95

[152] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012. 89

[153] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. 101

[154] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022. 101, 104

[155] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Communications of the ACM*, 66(1):74–85, 2022. 101

[156] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE TCAD*, 2019. 101

[157] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. 101

[158] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022. 101

[159] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–46, 2021. 102, 106

[160] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 55–60, 2022. 102, 105, 106, 112, 114, 115

[161] Nan Wu, Yuan Xie, and Cong Hao. Ironman-pro: Multi-objective design space exploration in hls via reinforcement learning and graph neural network based modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022. 102, 106

[162] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021. 102, 111

[163] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *EMNLP*, 2021. 102, 106, 112, 114, 115

[164] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020. 102, 106

[165] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 102, 106

[166] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020. 102

[167] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on CGO*, 2004. 105

[168] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. Learning semantic representations to verify hardware designs. *NeurIPS*, 34:23491–23504, 2021. 106

[169] Peng Xu, Alejandro Salado, and Guangrui Xie. A reinforcement learning approach to design verification strategies of engineered systems. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3543–3550. IEEE, 2020. 106

[170] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate

operation delay prediction for fpga hls using graph neural networks. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020. 106

[171] Yunsheng Bai, Atefeh Sohrabizadeh, Yizhou Sun, and Jason Cong. Improving gnn-based accelerator design automation with meta learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1347–1350, 2022. 106

[172] Haoxing Ren, George F Kokai, Walker J Turner, and Ting-Sheng Ku. Paragraph: Layout parasitics and device parameter prediction using graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 106

[173] Haoyu Peter Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. Unsupervised learning for combinatorial optimization with principled objective relaxation. In *NeurIPS*, 2022. 106

[174] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 106

[175] Tai Yang, Guoqing He, and Peng Cao. Pre-routing path delay estimation based on transformer and residual framework. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 184–189. IEEE, 2022. 106

[176] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020. 106

[177] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 106

[178] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 106

[179] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019. 107, 112, 115

[180] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 107

[181] Wonjae Kim, Bokyung Son, and Ildoo Kim. Vilt: Vision-and-language transformer without convolution or region supervision. In *International Conference on Machine Learning*, pages 5583–5594. PMLR, 2021. 107

[182] Siqu Long, Feiqi Cao, Soyeon Caren Han, and Haiqing Yang. Vision-and-language pretrained models: A survey. *IJCAI*, 2022. 107

[183] Yen-Chun Chen, Linjie Li, Licheng Yu, Ahmed El Kholy, Faisal Ahmed, Zhe Gan, Yu Cheng, and Jingjing Liu. Uniter: Universal image-text representation learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXX*, pages 104–120. Springer, 2020. 107

[184] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. Align before fuse: Vision and language representation learning with momentum distillation. *NeurIPS*, 34:9694–9705, 2021. 107

[185] Peng Wu, Xiangteng He, Mingqian Tang, Yiliang Lv, and Jing Liu. Hanet: Hierarchical alignment networks for video-text retrieval. In *Proceedings of the 29th ACM international conference on Multimedia*, pages 3518–3527, 2021. 107

[186] Zhenyu Huang, Guocheng Niu, Xiao Liu, Wenbiao Ding, Xinyan Xiao, Hua Wu, and Xi Peng. Learning with noisy correspondence for cross-modal matching. *NeurIPS*, 34:29406–29419, 2021. 107

[187] Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. Ernie: Enhanced language representation with informative entities. *ACL*, 2019. 107

[188] Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. *arXiv preprint arXiv:2210.03347*, 2022. 107

[189] Yong Dai, Duyu Tang, Liangxin Liu, Minghuan Tan, Cong Zhou, Jingquan Wang, Zhangyin Feng, Fan Zhang, Xueyu Hu, and Shuming Shi. One model, multiple modalities: A sparsely activated approach for text, sound, image, video and code. *arXiv preprint arXiv:2205.06126*, 2022. 107

[190] Michihiro Yasunaga, Antoine Bosselut, Hongyu Ren, Xikun Zhang, Christopher D Manning, Percy S Liang, and Jure Leskovec. Deep bidirectional language-knowledge graph pretraining. *Advances in Neural Information Processing Systems*, 35:37309–37323, 2022. 107

[191] Jiahua Rao, Zifei Shan, Longpo Liu, Yao Zhou, and Yuedong Yang. Retrieval-based knowledge augmented vision language pre-training. *arXiv preprint arXiv:2304.13923*, 2023. 107

[192] Relja Arandjelovic and Andrew Zisserman. Look, listen and learn. In *Proceedings of the IEEE international conference on computer vision*, pages 609–617, 2017. 107

[193] Chuang Gan, Deng Huang, Hang Zhao, Joshua B Tenenbaum, and Antonio Torralba. Music gesture for visual sound separation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10478–10487, 2020. 107

[194] Mohamed Afham, Isuru Dissanayake, Dinithi Dissanayake, Amaya Dharmasiri, Kanchana Thilakarathna, and Ranga Rodrigo. Crosspoint: Self-supervised cross-modal contrastive learning for 3d point cloud understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9902–9912, 2022. 107

[195] Amanpreet Singh, Ronghang Hu, Vedanuj Goswami, Guillaume Couairon, Wojciech Galuba, Marcus Rohrbach, and Douwe Kiela. Flava: A foundational language and vision alignment model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15638–15650, 2022. 107

[196] Michael Moor, Oishi Banerjee, Zahra Shakeri Hossein Abad, Harlan M Krumholz, Jure Leskovec, Eric J Topol, and Pranav Rajpurkar. Foundation models for generalist medical artificial intelligence. *Nature*, 616(7956):259–265, 2023. 107

[197] Gengchen Mai, Weiming Huang, Jin Sun, Suhang Song, Deepak Mishra, Ninghao Liu, Song Gao, Tianming Liu, Gao Cong, Yingjie Hu, et al. On the opportunities and challenges of foundation models for geospatial artificial intelligence. *arXiv preprint arXiv:2304.06798*, 2023. 107

[198] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. 107

[199] Yaochen Xie, Zhao Xu, Jingtun Zhang, Zhengyang Wang, and Shuiwang Ji. Self-supervised learning of graph neural networks: A unified review. *IEEE Trans. Pattern Anal. Mach. Intell.*, 45(2):2412–2429, 2023. 107

[200] Fan-Yun Sun, Jordan Hoffman, Vikas Verma, and Jian Tang. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. In *International Conference on Learning Representations*, 2019. 107

[201] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5812–5823. Curran Associates, Inc., 2020. 107

[202] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep Graph Infomax. In *International Conference on Learning Representations*, 2019. 107

[203] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. Gcc: Graph contrastive coding for graph neural network pre-training. *arXiv preprint arXiv:2006.09963*, 2020. 107

[204] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. Gpt-gnn: Generative pre-training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2020. 107

[205] Zhen Peng, Yixiang Dong, Minnan Luo, Xiao-Ming Wu, and Qinghua Zheng. Self-supervised graph representation learning via global context prediction. 03 2020. 107

[206] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. Self-supervised graph transformer on large-scale molecular data. *Advances in Neural Information Processing Systems*, 33, 2020. 107

[207] Ke Sun, Zhouchen Lin, and Zhanxing Zhu. Multi-stage self-supervised learning for graph convolutional networks on graphs with few labeled nodes. In *AAAI*, pages 5892–5899, 2020. 107

[208] Zhihui Hu, Guang Kou, Haoyu Zhang, Na Li, Ke Yang, and Lin Liu. Rectifying pseudo labels: Iterative feature clustering for graph representation learning. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, CIKM '21, page 720–729, New York, NY, USA, 2021. Association for Computing Machinery. 107

[209] Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. Greaselm: Graph reasoning enhanced language models. In *International conference on learning representations*, 2022. 109, 112, 115

[210] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017. 111

[211] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *IISWC*, 2014. 113

[212] Tomofumi Yuki and Louis-Noël Pouchet. Polybench/c. 113

[213] AMD/Xilinx Vivado HLS. `https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis`. 113

[214] AMD/Xilinx Vivado HLS. `https://docs.xilinx.com/v/u/2020.2-English/ug1416-vitis-documentation`. 113

[215] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. *IJCAI*, 2021. 114

[216] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *ICML*, 2018. 114

[217] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *Theory and Applications of Satisfiability Testing–SAT 2008: 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings 11*, pages 28–33. Springer, 2008. 118