Subgraph Pattern Matching on Graphs with Deep Representations

Yue Zhang, Ziyi Yang
Department of Mechanical Engineering
Stanford University
Stanford, CA 94305
yzhang16, ziyi.yang@stanford.edu

Zhaoyu Lou
Department of Computer Science
Stanford University
Stanford, CA 94306
zlou@stanford.edu

Abstract

Subgraph matching is useful yet hard to scale to large graphs. In this project, we propose subgraph matching model based on graph convolutional networks (GCNs). We reformulated the problem as node classification by encoding nodes in the search graph and query graphs into vector representations to predict if the search graph nodes match center node in query graphs. We explored two architectures, multi-layer perceptron (MLP) and order embedding. Both models achieves outstanding performance on single and multiple query graphs experiments and the second method has very low computational cost. Randomized training and curriculum raining are performed to push the model to get better generalization results.

1 Introduction

Graphs have always been a powerful tool to represent data for problems in real-life. They have been widely used especially in social network, web, bioinformatics and astrophysics applications. In current big data era, the size of graphs are growing larger and larger. For facebook, there are over 2.38 billion monthly active users as of March 31, 2019. Thus if we represent each user as a node, we will get a gigantic graph representing the social network.

In the meantime, subgraph matching problem is of special interest in graph problems, owing to the wide ranging applications from the functional characterization of proteins and their interactions to structural modeling of social networks to semantic structure discovery in knowledge graphs [3, 7]. The problem can be described as:

Definition: Given a dataset of graphs $G = \{G_1, \ldots, G_N\}$ and a query graph Q, identify the set $\mathcal{I}_H$ of all subgraphs in G which are isomorphic to Q.

Subgraph matching has a been a extensively studied problem in Graph theory. Recent work includes [4] that casts the matching problem as as a cross-network node similarity problem. TALE in [10] approximates matching and large query graphs by proposing NH-index (Neighborhood Index). Although inspiring progress has been made, most of the proposed methods attempt to solve the problem for graphs at size of around 50 to 100 nodes. Since the problem is NP-complete, it's hard to scale the existing graph-theory based methods to larger graphs, let alone those in current big data era. Thus we need to propose some new methods.

Graphical neural network (GNN) is an emerging representation learning method for graph structured data [6, 12], based on deep learning. The idea is that given an adjacency matrix G and a feature vector f_v for each node v, GNNs are able to learn a node embedding h_v for each node in the graph. It has been proved to be effective in encoding graph structures and has many successful examples in solving graph-related problems. In this project, we propose methods based on graph convolutional networks to attempt this problem. We expect that this neural network based method has better potential solving subgraph matching problems with larger size.

2 Dataset Description

In the current stage for algorithm development, we use synthetic graph data. The synthetic data is generated by the following steps. All graphs are generated using networkx package.

(a) Generate a large random base graph with m nodes, using Barabási-Albert preferential attachment model.

(b) Draw the subgraph motif we want to query and generate n such subgraphs.

(c) Attach each of the subgraph to the base graph by randomly setting up edges between the query graph nodes and the base graph nodes.

(d) Perturb the generated graph by randomly deleting some edges with certain probability.

This generation of graph ensures that the graph does have query graphs in it and the introduced randomness helps the query graphs to mix with the base graph, so that the task will not be complicated enough for our current stage. The figure below illustrates the resulting graph up to the 3rd step.

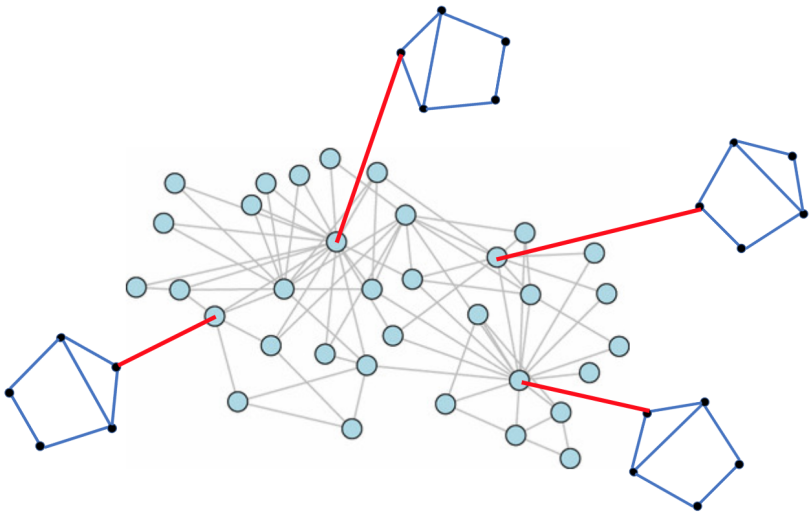

Figure 1: An example of synthetic graph. Green nodes and grey edges are base graph. Black nodes and blue edges are query graphs. The red edges serve to attach query graphs to the base graph.

3 Experiment Setup

3.1 Problem reformulation

We reformulate the subgraph matching problem as a node (neighbourhood) classification. Given a search graph and a query graph, we run a GCN on them to generate embeddings for each node. The graph convolution layer in the GCN (eq. (1)) aggregates information from each node's one-hop neighborhood to create new feature vectors. Details for embedding generation will be discussed in the next subsection.

Our model runs the embedding network on every node of the query graph as well as every node of the search graph. The embeddings of the query graph node and search graph node should be able to incorporate the neighborhood information and the graph structures. Then based on these two embedding information, we will do a binary classification to determine whether this node in the search graph is a matching with the center node in the query graph. Therefore, this becomes a binary classification problem for all the nodes in the search graph.

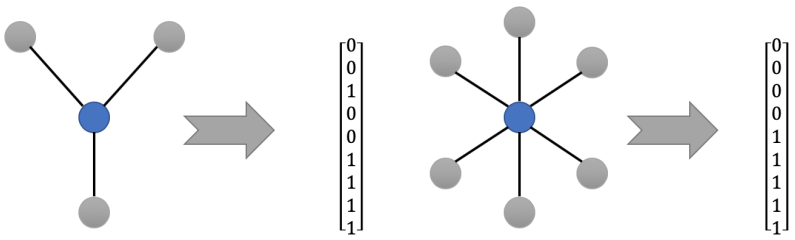

Figure 2: Our preprocessing step with $N = 5$, $M = 4$. The blue node on the left has degree 3, so the third element is 1. The blue node on the right has degree $6 > N$, so the N-th element is 1. The last M elements all vectors are constant 1.

3.2 Graph Convolutional Neural Network

We first encode the test and query graph by Graph Convolutional Network (GCN). The generated embedding vector for each node should bear information about the underlying structure of the neighbourhood of the node. Given a graph represented by a adjacency matrix G, we first initialize a feature vector for each node v. The embeddings are are generated by recursively aggregating each node's neighborhood information and features. At the t-th iteration, the information aggregation is defined as:

$$h_v^t = \text{ReLU}(\sum_{u \in S(v)} W^t h_u^{t-1} + b^t) \tag{1}$$

where $S(v) = N(v) \cup v$, h_v^t is the embedding for v at the t-th iteration and h_v^0 is the feature vector for node v. The final node representation is h_v^T from the last iteration T. The embedding network consists of T successive graph convolution layers, which aggregates information from the T-hop neighborhood. In our experiment, we choose $T = 3$, the size the size of node feature is 10 and the final output embedding size is 20.

3.3 Preprocessing

Before we can run the model, we need to assign each node an initial vector representation (feature vector). There are a few approaches to doing this. The most principled approach is to use identical features for every node - this forces the model to learn the matching task based purely on the graph topology. However, this is prone to slow starts due to the symmetry of the representations, and is somewhat artificial - in practice we will have node features.

Another approach which address the symmetry problem is to generate random feature vectors for each node. This alleviates the slow start problem, but nonetheless makes the problem needlessly difficult since in practice we have access to the graph structure so we can design more useful features.

A simple way to incorporate nontrivial node features is to simply use the node degree of every node as an embedding. This provides some initial information to warm start the model's learning. However we find that using a single real number (the node degree) as the feature results in a problem space too small to learn the task, so we instead use a one hot encoding where each element of the embedding is 1 if the node has some specific number of nodes, up until some maximum N, with the N-th element indicating a node degree of N or greater. To further expand the dimensionality of our problem space we concatenate these one hot degree embeddings with constant M dimensional feature vectors. This process is illustrated in Figure 2.

4 Methods

In this section, we describe in details the methods we experimented. At a higher level, we experimented two different methods. The baseline method is based on using cross entropy loss with concatenated embeddings through MLP. The other method we proposed is based on order embedding and a self-defined loss function, we leave the details in the following sections.

4.1 Baseline method

The baseline model is described as below. We first extract the embeddings of the center node of each query graph as well as the embeddings for each node in the search graph. Then for each node embedding in search graph, we concatencate the query node embedding vector. This concatenated feature vector is then passed through a small MLP which outputs final logits, indicating whether the node in the search graph corresponds to the node in the query graph. The model is trained then using binary cross entropy loss. Despite the fact that the method is intuitive, in our experiments we experimented with different training schemes, hoping to get the best generalization results on any subgraph query. In subsections below we describe the schemes we experimented, it roughly correpsonds to different levels of generalization.

4.1.1 Multiple Query Training

In problem reformulation section, we discussed how we reformulate the subgraph matching problem to a node classification problem. However, a major limitation of the aforementioned model is that it trains on a single query graph at a time. This is not a practical approach, since it would require retraining the model for every query graph of interest. We are interested in showing performance which is generalizable to many different queries, with the end goal of having a single model generalize to arbitrary query graphs.

The first step in this direction is to allow the model to train on multiple query graphs at once. The obvious way to do this is simply by batching the query graphs - since the graph convolutional network is essentially a series of matrix multiplications it should be possible to batch query graphs and train on all of them at once. However, we found that this was not the case in practice - batching query graphs lead to substantially degraded performance, likely due to correlations in query graph structure.

An alternative approach is alternating training, where for each minibatch we randomly choose one query graph to train on, populate the minibatch with examples which are either positive examples of the chosen query or negative for all queries, and train on that minibatch as a binary classification problem. The problem then is recast into a series of m binary classification problems, where m is the number of query graphs. We find that this works quite well in practice.

4.1.2 Randomized Training

In seeking to achieve generalized performance on arbitrary graphs, we sought to train the network on randomly generated query graphs, rather than the relatively small query graphs we had previously been working. Our first approach to this was taking random subgraphs of the large search graph. To do so, we randomly select a node in the search graph to use as our center node. We then do a breadth first traversal of the graph starting from the center node, adding each edge we traverse with a given probability (we used a probability of 0.7). The search terminates at some specified distance from the center node, in our case 3 hops. The resulting subgraph is used as a query graph. However, this approach suffers from the fact that we only have a single positive example for any given query, which allows the model to simply memorize positive examples.

The obvious solution to this issue would be to leverage the semisupervised approach we used with the original deterministic queries. We can randomly attach more instances of the generated queries to the graph by adding random edges between the search graph and the nodes of the generated queries. While this would work, we noticed that the search graph is not actually necessary when we move to the randomized setting, and so we took a different approach. We discarded the search graph altogether and created a dataset of query graphs. Query graphs were generated by starting from a single node, and choosing a random number of neighbors for that node. Each of these neighbors would have this process repeated on them, until a 3 hop query graph had been generated. Finally, a number of extra edges would be added between randomly selected nodes. Then for each of these query graphs, we would create a number of positive neighborhoods by adding a number of extra nodes and connecting them randomly to nodes of the query graph, and adding extra edges between these nodes. Thus each query graph had a number of positive neighborhoods and a number of negative neighborhoods, which would be any neighborhood not generated as a positive neighborhood for that query. We then trained the model on this randomized dataset.

4.1.3 Curriculum Training

We found that this randomized dataset was, unsurprisingly, much more difficult than the original dataset to learn. Accordingly, we introduced a new curriculum training scheme, where the model is first trained on a small number of easy queries and is then trained on successively more complex and more numerous queries while still forced to retain performance on the old ones. More specifically, for the first 200 epochs of training the model is trained on 2 one hop query graphs. For the next 300 epochs, the model is trained on 4 one hop queries, and then for the next 500 it is trained on 4 two hop queries in addition to the 4 one hop queries. For the next 800 epochs it is trained on the previous queries in addition to 4 three hop queries. After this the model is trained on the previous queries plus another 4 three hop queries, which is as far as we have tried curriculum training thus far.

4.2 Order Embeddings Method

In previous sections, we solve the subgraph matching problem by concatenating the embeddings for query node and target graph node and pass it through MLP to do node classification. However, if the search graph size is huge or we have too many queries to make, the computational cost could not be negligible. In this section, we motivate a new method that is more computationally efficient. The key idea is that instead of concatenating the embeddings, we want to compare the trained embeddings directly with some operations that have negligible cost.

To realize the idea, we adapt the idea of order embedding, which was first proposed in [11] and has been used in image caption generation and several other problems successfully. The algorithm can be described as,

Order Embedding: Given query node embedding $\vec{x}$ and target graph embedding $\vec{y}$, we compare the numerical values of the two embeddings elementwise. For all embedding dimensions N, if we have

$$\wedge_{i=1}^{N} x_i < y_i$$

then we say that the query node is a subgraph node of the search graph, otherwise it is not.

To train this set of embeddings, we adjust our loss functions accordingly. We use the following function to penalize the order violation, where

$$E(\vec{x}, \vec{y}) = ||\max(0, \vec{y} - \vec{x})||^2 \tag{2}$$

It is clear to see that, for positive examples, the function above is minimized when all the elements in query node embedding $\vec{x}$ is less than the corresponding elements in search graph node embedding $\vec{y}$. Thus for positive examples we use this as the loss function, as shown in the first term below. For negative cases, we introduce the max-margin loss, and the margin is α, which is a tuning parameter. Then the total loss function can be written as

$$\sum_{(\vec{x},\vec{y})\in P} E(\vec{x}, \vec{y}) + \sum_{(\vec{x},\vec{y})\in N} \max\{0, \alpha - E(\vec{x}, \vec{y})\} \tag{3}$$

In experiments, we find that enforcing all dimensions to follow the inequality order can be too strict and may lead to training instability. We thus relax the problem and introduce another hyperparameter β to describe the fraction of dimensions that we allow to violate the threshold. β then should not be a large value. In our experiment with embedding dimension set to 60, we find that setting $\beta = 0.2$, which means we allow at most 12 dimensions to violate. This really helps to stabilize the training and test process.

As we see from above, the computational cost for doing classification is just comparing the numbers for each element, which should be much faster than base model when we scale up. It also makes the neural network and order embeddings more interpretable and we will show the interpretation in the result section on order embedding.

5 Results

5.1 Baseline model

5.1.1 Single Query Training

Training on single queries, the model quickly learns to classify with near perfect performance, as can be seen from Figure 3. This experiment was run using a house shaped graph as the query graph.

Happily, this performance also generalizes quite well to the test set - it takes a few more iterations to generalize to the test set, but overall the model converges very quickly to very high performance.

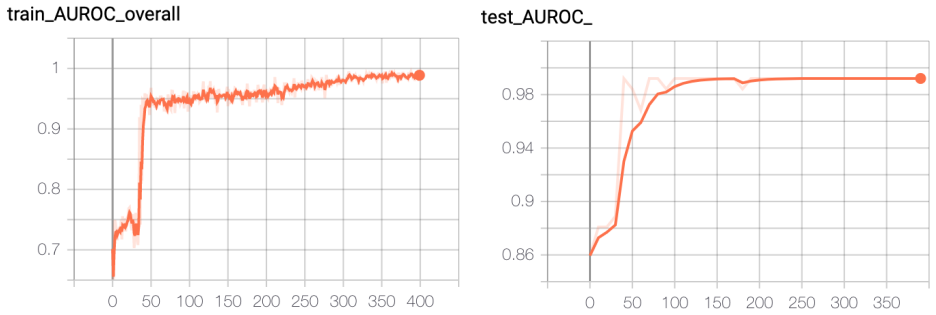

Figure 3: Train and Test AUROC for single query classification.

5.1.2 Multiple Query Training

Before showing the results from multiple query training, we show the geometries of graphs we studied.

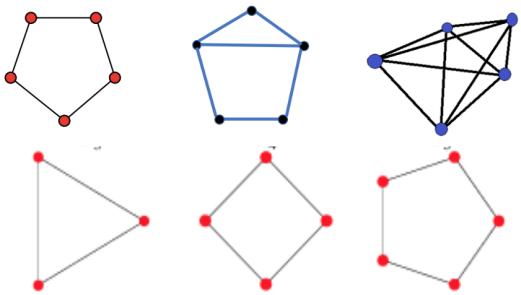

Figure 4: The geometry of queries in the experiment. The three graphs in the first row are named as 'cycle 5', 'house' and 'clique', from left to right. The three graphs in the second row are named as 'cycles' with different lengths, shown are lengths of 3, 4 and 5, from left to right.

Query Families The next step in getting generalization in the model is training on multiple, similar query graphs. We chose to experiment on cycle graphs, which are simple, relatively easy to learn, and readily define a family of graphs (cycles of different lengths). Figure 5 shows performance when the model is trained on cycles of length 2, 3, 4, 5, and 6. We can see that again the model is able to quickly learn to classify all these query graphs with near perfect accuracy; in fact, since cycle graphs are simpler to learn than house graphs, the convergence is even faster.

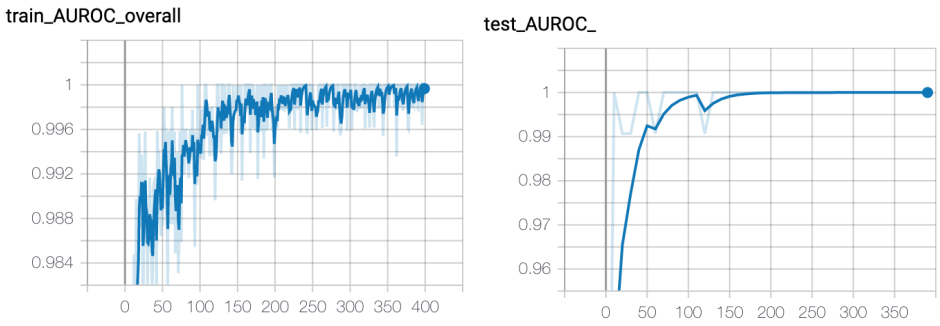

Figure 5: Train and Test AUROC for multiple cycle classification.

Next, we sought to achieve generalization even to graphs that we had not trained on. Continuing with the cycle graphs, we trained on two cycle graphs (lengths 3 and 4) and tested on cycles of length 2 through 6. The results are shown in 6. The test performance takes longer to converge than the previous case when we trained on all the cycles we test on, but nonetheless the model is able to generalize to classifying even the unseen queries close to perfectly.

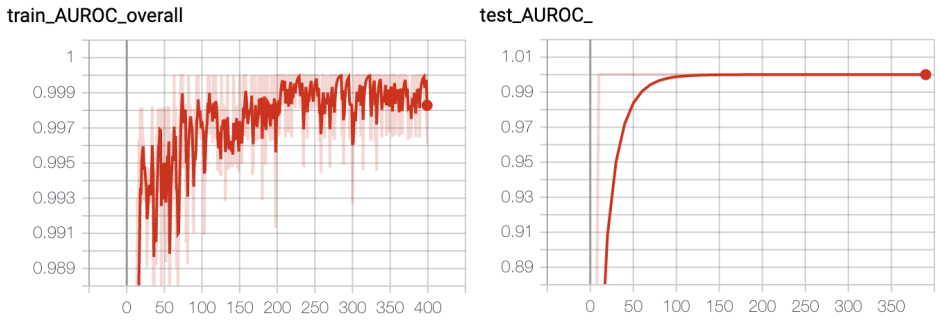

Figure 6: Train and Test AUROC for unseen cycle classification.

General Queries Given the model's strong performance on sets of similar queries, a natural question was whether or not the model would perform as well when the query graphs were not deliberately selected to be similar to each other. Figure 9 shows the model's performance when the query graphs are a house, a cycle of length 6, and a fully connected clique of 5 nodes. Here, we begin to see difficulties in the model. First, the convergence is much slower, taking on the order of a few hundred epochs instead of the < 100 epochs it took on previous tasks. We also see that the model is beginning to exhibit overfitting on this task - the test performance plateaus at an AUROC of around 0.83. While the slower convergence is not surprising - after all, the problem is an NP complete problem - the overfitting is concerning and will need to be addressed. We are still looking into methods to alleviate this overfitting and improve test performance.

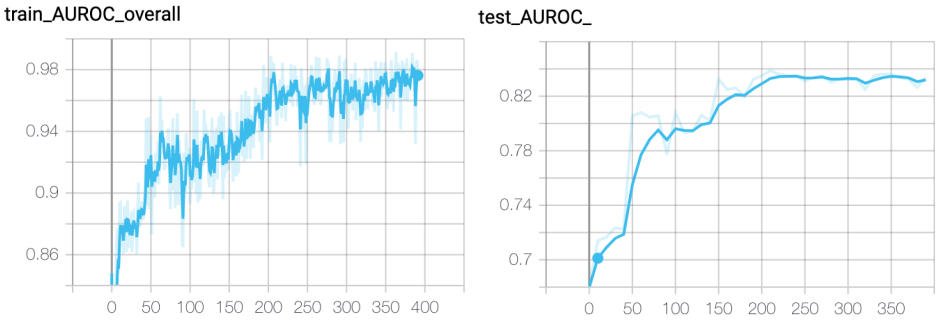

Figure 7: Train and Test AUROC for dissimilar query classification.

5.1.3 Randomized and Curriculum Training

The performance of curriculum training in the original randomized training setting is shown in Figure 8. We see that performance drops sharply with each new introduction of query graphs, but that the model is able to recover it's original performnace and generalize to the new queries given time. We are still working on extending this to the new randomized dataset we created.

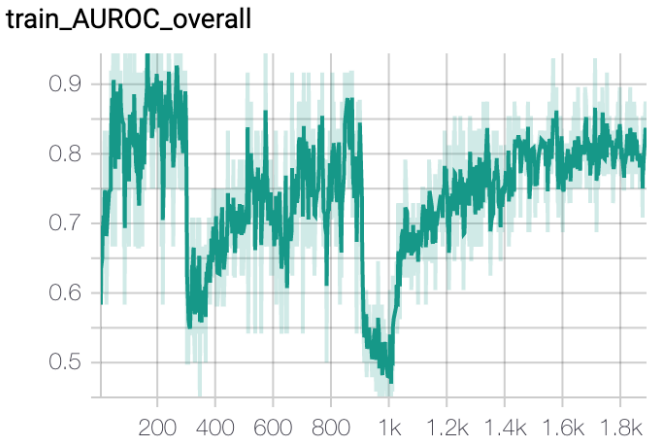

Figure 8: AUROC for Curriculum Training.

5.2 Order Embeddings

Performance: In Figure 7, we report the performance of order embedding training. The training is based on multiple queries and we are achieving very similar performance in comparison with MLP method. The computational time, however, is only half of the baseline method. The multiple query experiments are carried out here. These include 'cycles' with 3,4 and 5 nodes, also included are 'house' and 'clique'. The embedding dimension is 60 and the violation tolerance threshold is 0.2. The training and test are both quite stable.

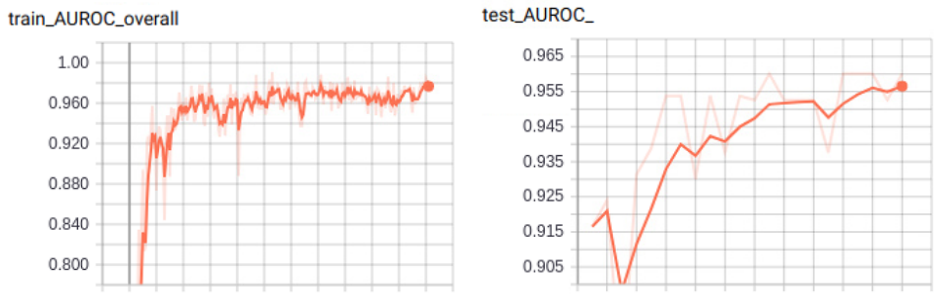

Figure 9: Train and Test AUROC using order embedding training

Interpretation: To better understand the training process and the resulting embedding. We check the embeddings for each of the following graphs from the model. For the query series of 'cycle 5', 'house' and 'clique', we find that 'cycle 5' is a subgraph of 'house' and 'house' is a subgraph of 'clique'. This can be visualized from the query geometry figure above. Thus, we should expect the embedding of 'cycle 5' to be smaller than 'house' and embedding of 'house' smaller than 'clique'. The embedding dimension we used is 60 and we set a order violation threshold to be 0.2 in training. Thus, we expect at most 12 dimension violations in the embedding. The table below shows the embedding violation for the comparison above, all the violations are less than 12. In comparison, we compare the node embeddings of 'cycle 3', 'cycle 4' and 'cycle 5' and we find that the violations are much larger than 12, indicating that these should not have subgraph relationship. Since these relationship are not hardcoded in the loss function, we can conclude that the trained order embeddings do capture the torpological relationship to some extent.

Graphs	# of dimension violations
'cycle 5' vs 'house'	$0 < 60 \times 0.2$
'house' vs 'clique'	$9 < 60 \times 0.2$
'cycle 3' vs 'cycle 4'	$35 > 60 \times 0.2$
'cycle 4' vs 'cycle 5'	$32 > 60 \times 0.2$

6 Conclusion

We proposed a subgraphs matching scheme based on graph convolutional neural network. By reformulating the matching problem as a node (neighborhood) classification problem, we show that our proposed method can successfully address both single and multiple query graph problem, showing great generalization potential. Besides muti-layer perceptron (MLP) prediction, we further propose computation-efficient order embedding scheme that can directly predict the matching by numerical comparison.

7 Future Work

In this work we explored graph with only one type of edges. The future work includes exploring network with multi-category edges, for example, knowledge graph [8]. Furthermore, we plan to test on real-world daatset, include protein-protein interactions (PPI) network [9]. We also consider applying our new methods on some molecule benchmarks as subgraph pattern matching is a useful tool in cheminformatics. One good candidate for molecular dataset is a dataset studied in [5]. The data set consists of a collection of 1235 SMARTS substructure expressions and selected molecules from the ZINC database and was used in a substructure search study. Other common datasets for this molecular modeling task include the BIND and ASTRAL datasets, the former containing protein interaction graphs and the second containing 3D protein structures [1, 2]. If we need additional data in different domains, it should be easy to obtain data by e.g. scraping the link structure of Wikipedia.

References

[1] C. Alfarano, C. Andrade, K. Anthony, N. Bahroos, M. Bajec, K. Bantoft, D. Betel, B. Bobechko, K. Boutilier, E. Burgess, et al. The biomolecular interaction network database and related tools 2005 update. *Nucleic acids research*, 33(suppl_1):D418–D424, 2005.

[2] J.-M. Chandonia, G. Hon, N. S. Walker, L. Lo Conte, P. Koehl, M. Levitt, and S. E. Brenner. The astral compendium in 2004. *Nucleic acids research*, 32(suppl_1):D189–D192, 2004.

[3] D. G. Corneil, I. Jurisica, and N. Pržulj. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 02 2006.

[4] B. Du, S. Zhang, N. Cao, and H. Tong. First: Fast interactive attributed subgraph matching. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1447–1456. ACM, 2017.

[5] H.-C. Ehrlich and M. Rarey. Systematic benchmark of substructure search in molecular graphs - from ullmann to vf2. *Journal of Cheminformatics*, 4(1):13, Jul 2012.

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

[7] T. A. B. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological Methodology*, 36(1):99–153, 2006.

[8] R. Speer, J. Chin, and C. Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[9] D. Szklarczyk, J. H. Morris, H. Cook, M. Kuhn, S. Wyder, M. Simonovic, A. Santos, N. T. Doncheva, A. Roth, P. Bork, et al. The string database in 2017: quality-controlled protein–protein association networks, made broadly accessible. *Nucleic acids research*, page gkw937, 2016.

[10] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *2008 IEEE 24th International Conference on Data Engineering*, pages 963–972. IEEE, 2008.

[11] I. Vendrov, R. Kiros, S. Fidler, and R. Urtasun. Order-embeddings of images and language. *arXiv preprint arXiv:1511.06361*, 2015.

[12] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.