SUBGRAPH MATCHING PROBLEMS IN A NUTSHELL

PROJECT REPORT

Ji Zhang
19300180021@fudan.eud.cn

Xinfeng Yuan
19307110525@fudan.edu.cn

Lesi Chen
19307130195@fudan.edu.cn

ABSTRACT

In our final work, we investigate a series of problems related to subgraph matching, and make improvement based on existing algorithms. In some parts, we also come up with some innovative algorithms. Besides, we implement several code package and did comparative experiments. Our exploration of subgraph matching problem begins with the original base subgraph matching and then makes extension to dynamic subgraph matching and frequent subgraph mining. We also consider extending subgraphs to similar subgraphs and work on similar subgraph mining and frequent subgraph mining problems.

1 Base Problem: Subgraph Matching

As we believe that subgraph matching is the key problem in all of the following problems, we first study the algorithms on subgraph matching. We will see that the techniques and ideas of the classic algorithm in this problem inspires for our solution to other subgraph relevant problems.

1.1 Problem Definition

Given a query graph $Q(V_Q, E_Q, L_Q)$ and a data graph $G(V_G, E_G, L_G)$ (with V the set of vertexes, E the set of edges and L the set of vertex labels), we say Q is subgraph isomorphism or subgraph matching to G , if and only if there exists a function $f : V_Q \to V_G$, such that: $\forall v \in V_Q, L_Q(v) = L_G(f(v))$ and $\forall v, u \in V_Q, (v, u) \in E_Q, (f(v), f(u)) \in E_G$,which is one of the well-known NP-hard problems.

1.2 Existing Methods

DP-iso or DAF Han et al. [2019] is a representative work in the domain of subgraph matching. DP-iso uses dynamic programming (DP) between a directed acyclic graph (DAG) and a graph, adaptive matching order with DAG ordering, and pruning by failing sets to get the performance of state-of-the-art. We believe that the ideas in DP-iso may have universal value in solving the subgraph relative problems, such as dynamic subgraph matching, similar subgraph matching and frequent subgraph mining.

1.3 Filter, Order and Failing Set

Filter The representative methods of subgraph matching (including CFL,CECI,DP-iso, QuickSI, RI and VF2++ Sun and Luo [2020]) use either label filtering and degree filtering(LDF) or neighbor label frequency filtering (NLF). The label and degree filtering (LDF) generates the candidates of vertex u, $C(u)$ based on the label of that vertex $L(u)$ and the degree of that vertex $d(u)$: $C(u) = \{v \in V(G) \mid L(v) = L(u) \wedge d(v) \geqslant d(u)\}$. All existing algorithms adopt LDF. The neighbor label frequency filtering (NLF) utilizes the neighbors of a vertex $N(u)$ to prune $C(u)$ as follows: given $v \in C(u)$, if there exists $l \in L(N(u))$ such that $|N(u,l)| > |N(v,l)|$ where $L(N(u)) = \{L(u') \mid u' \in N(u)\}$ and $N(u,l) = \{u' \in N(u) \mid L(u') = l\}$, then remove v from C(u) . CFL, CECI and DP-iso utilize NLF.

We believe the idea of filtering is valuable and we adopt filtering in Frequent Subgraph Mining. And we will also see that in our attention-based graph neural network, the design of attention mechanism has a close relationship with the idea of candidate and filter.

Order Order does matter in searching. QuickSI proposes an infrequent-edge first ordering method; GraphQL proposes a left-deep join based method; CFL proposes a path-based ordering method, and puts core vertices at the beginning of the matching order; CECI uses the BFS traversal order of query graph Q as the matching order; DP-iso proposes an adaptive ordering method that dynamically selects the next query vertex during the enumeration. A good order may be the key of a good algorithm.

In the data structure proposed by us, the Success Set, order also does matter. Even we have not design a new order technique for Success Set, we know that the Success Set must choose a good order to be more effective so ordering method still need to be taken into careful consideration.

Failing Set DP-iso proposed the failing sets pruning method, which utilizes the information obtained from the exploration of the search subtree rooted at a partial result M to rule out some invalid partial results, especially the siblings of M in the search tree. We find that this optimization method cannot only work with DP-iso but also other algorithms. In addition, maybe the idea of failing sets can be useful in other searching problems or constrain satisfaction problems. In our algorithm for dynamic subgraph matching, we modify and improve the failing sets into Failure Set.

1.4 Experiment

We implement the state-of-the-art subgraph matching algorithm DP-iso as a base algorithm with C++. In our experiment, we study the effect of different types of filters, orders and whether the technique of failing set contributes to the efficiency of searching. It is worth noticing that the problem of subgraph matching is the basic of all other problems, so we need to implement an algorithm of subgraph matching first. When solving the problem of dynamic subgraph matching, DP-iso is a perfect baseline to show the importance of the design of incremental algorithm. When mining frequent subgraphs or similar subgraphs, we generate candidate graphs and we need to use subgraph matching algorithm to verify the candidate graph is exactly a subgraph of the data graph. Plus, when calculating the frequency of a pattern, we need to use subgraph matching to find the subgraph embeddings. In all, we use the algorithm DP-iso implemented by us as a sub-program in other programs solving related problems. The results of comparative experiments are showed as below, where the engine here refers to the enumeration methods using when searching, and LFTJ is the naive method which is used in QuickSI, RI and VF2++, while GraphQL (or GQL for short) and DP-iso develops more advanced enumeration methods.

Table 1: Comparative experiments on Subgraph Matching

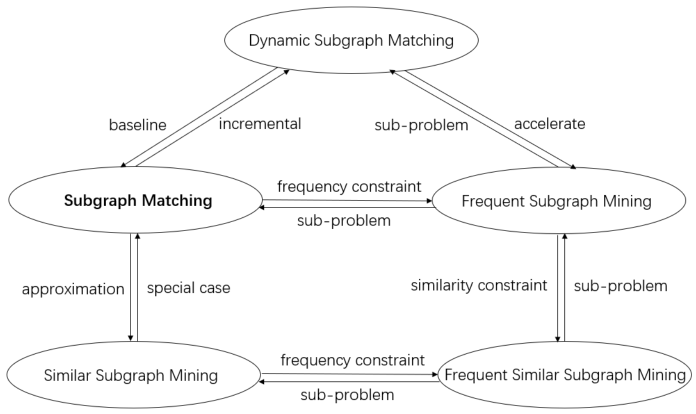
filter	order	engine	failing set	mean total time(s)
DPiso	DPiso	DPiso	×	1e-05
			✓	9e-06
GQL	DPiso	DPiso	×	1e-05
			✓	1e-05
NLF	DPiso	DPiso	×	2e-05
			✓	1e-05
LDF	DPiso	DPiso	×	1e-05
			✓	1e-05
LDF	DPiso	LFTJ	×	1e-05
			✓	9e-06
LDF	GQL	LFTJ	×	9e-06
			✓	1e-05
LDF	DPiso	DPiso	×	1e-05
			✓	1e-05
LDF	DPiso	GQL	×	3e-05
			✓	3e-05

These experiments are conducted on HDRP dataset. And the results reveal the effectiveness of filter that a good filter could decrease the time needed. Therefore we also apply filter on follow-up related problems. The experiments also show the effect of failing set which motivates us to come up with failure set which will be introduced later. It is worth noticing that sometimes failing set does not bring us a better performance, we think when the query is simple to deal with, the failing set needs to spend time on maintaining indexes. In our failure set, we design an backtrack algorithm with less redundant index maintain and the same pruning power, which may be a solution of the problem we found in our experiments of failing set. The HDRP dataset is available at https://github.com/RapidsAtHKUST/SubgraphMatching.

In conclusion the DP-iso performs best on Subgraph Matching and we implement it as the core code for matching in following problems.

1.5 Related Problems

The following figure shows all the problems related to subgraph matching as a base problem and illustrates the relationship between them:



Dynamic subgraph matching is an extension of the original static subgraph matching problems, where the data graph G evolves over time. A naive method to solve dynamic subgraph matching problem is to solve a subgraph matching sub-problem at every time step t. Hence, algorithms for subgraph matching problems can be a baseline for dynamic subgraph matching. However, this naive method does not explore the information from the evolution of data graph G, so we need to design incremental algorithm to deal with each vertex or edge update over data graph G and report the change of subgraph matches.

Frequent subgraph mining aims at finding the frequent patterns on a single large graph. It is worth noticing that the frequent patterns need to be a subgraph first. Hence, on the one hand, the problem of frequent subgraph mining can be considered as the original subgraph matching problem added frequency constraint. On the other hand, each algorithm for frequent subgraph mining is required to solve the subgraph matching problem as a sub-problem. Therefore, the efficiency of subgraph matching may become the bottleneck of frequent subgraph mining. Furthermore, using a Gspan Yan and Han [2002] like framework, we can see that sub-problem can be transformed into the problem of dynamic subgraph matching. Then we can apply the algorithms which solve dynamic subgraph matching for acceleration of frequent subgraph mining.

Similar subgraph matching relax the base subgraph matching problem. The solutions of similar subgraph mining are unnecessary to be exactly isomorphic to a given query graph Q, the subgraphs similar enough to Q are also accepted. Hence, the solutions of similar subgraph mining can also be an approximation of base problem. If we use graph edit distance (GED) as a measurement for similarity, then subgraph matching is exactly a special case of similar subgraph mining where we want to find all the subgraphs whose GED is less than 1 ,i.e. equal to 0.

Frequent similar subgraph mining requires to find out all the subgraph which are both frequent and similar to a given pattern graph P. It is a natural extended problem because it can be regarded as the base problem added both frequency and similarity constraint. However, the combination of constraints make the new problem much more difficult to solve, so we need to use more techniques in this new problem (eg. a sample based framework which is also used by some frequent subgraph mining algorithms).

Remark Based on our observation and the relationship between the above subgraph matching related problems. We put all the problems together, and try to solve them separately to our best. Due to the similarity and relationship with the problems, we adopt the techniques from one problem for solving another related problems.

2 Dynamic Subgraph Matching

2.1 Problem Definition

Given a data graph G_t at time stamp t and the matches of a query graph Q (denoted by $M(Q)$), the task is finding the matches of Q over G_{t+1} based on the previous matches where $G_{t+1} = G_t + \Delta G$ and ΔG is a set of update operations including vertex additions, vertex deletions, edge additions and edge deletions. As we know that static subgraph matching is a NP-hard problem, we can also proof by contradiction that dynamic subgraph matching is also a NP-hard problem.

Remark Since deleting a vertex u with neighbors can be composed as first deleting all the edges between the u can its neighbors and then deleting vertex u (now u is an isolated vertex). We just need to consider the vertex operations on isolated vertexes,i.e. to delete an isolated vertex or add an isolated vertex, which are easier to deal with.

2.2 Motivation

Related work, such as SJ-Tree(Choudhury et al. [2015]), Turboflux(Kim et al. [2018]), Symbi(Min et al. [2021]) use different methods to solve the problem of dynamic subgraph matching. However, we found that these methods neither store too little nor too much information of previous matching. To be more specific, SJ-Tree partition the whole query graph Q into small structures(i.e. one-edge structures or tow-edge structures) and store the partial matching of the structures, leading to huge memory cost for storage. Turboflux tried to solve this problem by employing a concise representation of intermediate results based on spanning tree, with a well-designed edge transition model. Symbi outperforms Turboflux by maintaining an auxiliary data structure based on a directed acyclic graph (DAG) instead of a spanning tree, which also maintains the intermediate results of bidirectional dynamic programming between the query graph Q and the dynamic graph G.

However, we found that essentially both Symbi and Turboflux maintain the “mark" of filtering (or the filter path) , as Turboflux maintains the filter path on a spanning tree and Symbi maintains the filter path on a DAG). The filter path can be regarded as a concise representation of the previous subgraph matching results. However, this way of concise representation may lose much information left by previous subgraph matching as we know that filtering is merely a pre-pruning technique before searching, i.e. even after a tight filter, we still need to spend exponential time on searching is we are concerned on the results. Thus, even Symbi and Turboflux outperform SJ-Tree in space complexity, they may not perform as well as SJ-Tree in time complexity. Based on the above considerations, we are aiming at designing a more effective and efficient index data structure for the storage of the "mark" of searching (or the search path) instead of just the "mark" of filter. In this way, we can utilise the information producing when searching the results of subgraph matching on G_0 to G_t to guide the search of the subgraph matching on G_{t+1}.

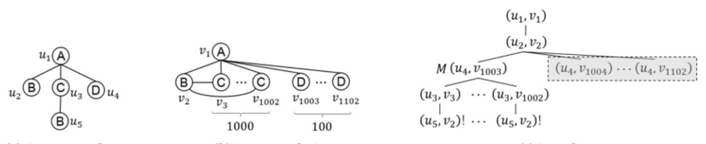
2.3 Method Overview

Our solution to the above problems is to store the common knowledge producing by searching. Based on the observation that there exist many similar structures both on the query graph Q and the data graph G, as a consequence of which, there exist many similar matches even G evolves a lot as time goes by. To be more specific, both the successful searching and the unsuccessful searching can be similar on similar structures. We maintain a Success Set for recording the successful searching, and when the data graph changes, there is no need for the program to search from sketch, but it can continue the search based on the previous search results in the success set. Also, a failure set is maintained to record the similar causes of failure inspired by the failing set proposed by Han et al. [2019]. While the success set allow us to continue the promising search, the failure set allow us to prune the unpromising search. In short, both success set and failure set allow us to avoid redundant computation when searching.

2.4 Failure Set

The motivation of failure set is that we know that the failures in searching result from the conflicts of constrains between vertexes. Then we can conclude that if we do not succeed in a previous search due to certain conflicts , we will fail due to the same reason if we do not try to solve the conflicts.

The following figure illustrates the motivation of Failure Set:



(a) Query graph q **(b) Data graph G** **(c) Search tree**

Suppose (c) is a search tree for query graph q in Figure (a) and data graph G in Figure (b), and also suppose that we just came back to node $M = \{(u_1, v_1), (u_2, v_2), (u_4, v_{1003)}\}$ after the exploration of the subtree rooted at M. Recall that we use the mapping function M to represent a node as well as a partial embedding. During the exploration, we have tried all possible extensions to map u_3 and u_5, and all attempts to map u_5 have failed by the conflicts in the mappings of u_2 and u_5. However, u_4 is not relevant to any of the failures because u_4 is not involved in the conflicts. It means that no matter how we change the mapping of u_4 in the current mapping M, it will never lead to an embedding of q because all possible extensions will end up with failures in the same way. Thus, we need not extend all other siblings of node M (depicted by a dashed box in (c).

In the above example, we know that Failure Set have a close relationship with the conflict set. Based on that, DP-iso uses conflict-class and Failing Set to prune redundant nodes in the search tree. DP-iso maintains a Failing Set for each node in the search tree and uses the following Lemma for pruning.

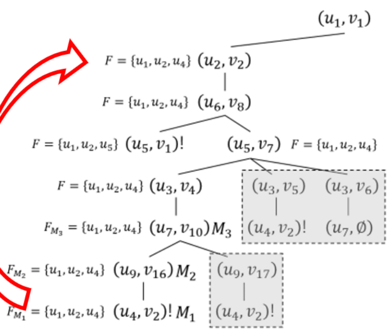
Failing Set Lemma Let M be a search tree node labeled with (u, v) (i.e. M is a partial embedding in which the last mapping is (u, v), and F_M be a non-empty failing set of node M. If $u \in F_M$, then all siblings of node M (including itself) are redundant.

However, we find out that there is no need to maintain a Failing Set for each node in the search tree as DP-iso does. Every time DP-iso uses the above Failing Set Lemma for pruning, we can know that it is meaningless to maintain the nodes being pruned. Based on this motivation, we modify the original Failing Set used in DP-iso, and we call our data structure Failure Set, which is similar but slightly different from Failing Set. The key of the modification comes from the following technique, which is called Leapfrog Backtrack.

Leapfrog Backtrack Before we formally introduce the technique of Leapfrog Backtrack, we first simplify and abstract a concept called Conflict Set from Failing Set of DP-iso. The Conflict Set is defined the same as the Failing Set of each leave node in DP-iso, that is to say, the Conflict Set of a vertex u maintains the other vertex which has constrains on u. For example, in Figure (a) , we can conclude that u_4 would not be in the Conflict Set of vertex u_2 for they have different label and there is no edge between them and they have no common ancestors either.

With the Conflict Set, we can illustrate Leapfrog Backtrack clearly. When we reach a leaf node $v$in the search tree, we initialize the Failure Set of v with the Conflict Set of v, which is the same as what DP-iso does. However, when backtracking, instead of backtracking towards the direct parent of node v, we just need to backtrack towards the nearest ancestor u of node v which is in the Failure Set of node v. After we reach ancestor u, we absorb the Failure Set of node v into the Conflict Set of node u, forming the Failure Set of u. And the procedure is applied recursively.

We can see an example for Leapfrog Backtrack:


(u_1, v_1)
$F = \{u_1, u_2, u_4\}$ (u_2, v_2)
$F = \{u_1, u_2, u_4\}$ (u_6, v_8)
$F = \{u_1, u_2, u_5\}$ $(u_5, v_1)!$
(u_5, v_7) $F = \{u_1, u_2, u_4\}$
$F = \{u_1, u_2, u_4\}$ (u_3, v_4)
(u_3, v_5)
(u_3, v_6)
$F_{M_3} = \{u_1, u_2, u_4\}$ $(u_7, v_{10}) M_3$
$(u_4, v_2)!$
$(u_7, \emptyset)$
$F_{M_2} = \{u_1, u_2, u_4\}$ $(u_9, v_{16}) M_2$
(u_9, v_{17})
$F_{M_1} = \{u_1, u_2, u_4\}$ $(u_4, v_2)! M_1$
$(u_4, v_2)!$

After we reach the leaf node $M_1 = (u_4, v_2)$ in the search tree, we initialize the Failure Set of node M_1 with the Conflict Set of vertex v_4 as $\{u_1, u_2, u_4\}$. Now backtracking happens, and we directly jump to the nearest ancestor of vertex u_4 in the Failure Set of node M_1 and reach the node (u_2, v_2). Then, node (u_2, v_2) absorb the Failure Set of node M_1 (i.e. $\{u_1, u_2, u_4\}$) into the Conflict Set of node (u_2, v_2) (i.e.$\{u_1, u_2\}$, forming the Failure Set of $\{u_1, u_2, u_4\}$. Then, the procedure is applied recursively and (u_1, v_1) will be reached in the next backtrack.

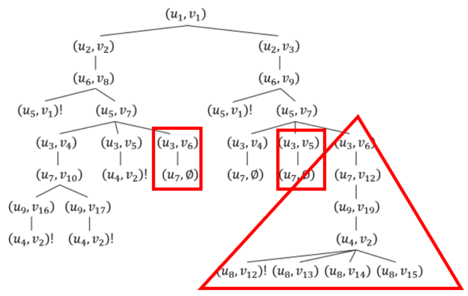
With the help of Leapfrog Backtrack, we avoid the redundant computation and storage of the Failing Set of the node (u_9, v_16), (u_7, v_10), (u_3, v_4), (u_5, v_1), (u_6, v_8) and the examination of whether the condition in Failing Set Lemma is satisfied and whether pruning can be applied in node (u_7, v_10) and node (u_5, v_7). Hence, our method will be more efficient than the Failing Set of DP-iso, without undermining the same pruning effect. In summary, the Failure Set proposed by us would be a good modification and improvement of the original Failing Set.

2.5 Success Set

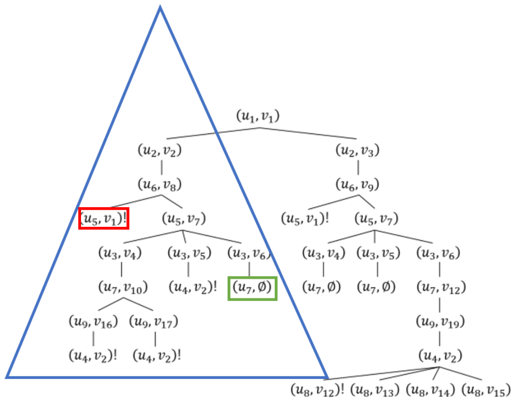
We first illustrate the definition of Success Set. Suppose a query graph Q with m vertexes and an order given by our algorithm (eg. the order using by DP-iso) which is also the order we use in searing. In this way, we can define m Success Sets, corresponding with the partial embedding in subgraph matching. The largest Success Set (i.e. the m_{th}

Success Set) stores all the results of subgraph matching, and other Success Sets stores the partial result when searching with the given order. When we add an edge in the data graph G, we examine the embedding in the i_{th} Success Set and check whether it can be put into the $\{i+1\}_{\text{th}}$ Success Set. Similarly, when we delete an edge in G, we examine whether the embedding in the i_{th} Success Set need to be put into the $\{i-1\}_{\text{th}}$ Success Set. What we need to do is just to maintain the Success Set, and we will show that Success Set can be maintained on the search tree.

Update on Search Tree We use the search tree for the implementation and maintain of Success Set. It is clear that all the nodes and their ancestors in the search tree of depth i is exactly the nodes in i_{th} Success Set. Hence, we just need to update the search tree and we can maintain the Success Set. Consider the following example, suppose we have the search tree in the figure, when the data graph G add or delete an edge (v_5, v_6), we find the subtree whose root is related to u_5 or v_6 (i.e. the red subtree in the figure) and re-search in those subtrees.



Partial Update Practically, there is no need to re-search the whole subtree, since the search at time stamp $t+1$ will not differ from the search at time stamp t a lot. Consider another example, that we add the edge (v_1, v_2) and the blue subtree is one of the subtrees that we need to re-search. However, consider the green leaf node in the following figure, we can find that the search $\{(u_2, v_2), (u_6, v_8), (u_5, v_7), (u_3, v_6), (u_7, \phi)\}$ will be the same as previous search as all the node in the partial embedding remain unchanged. Only the read leaf node in the following figure need to re-search for we need to examine the previous failure at the leaf node (u_5, v_1) will change, which requires a re-search. In conclusion, we can proof that when we only need to re-search the leaf node which is related to v_1 in the subtree whose root is related to v_2 or vice versa, the leaf node which is related to v_2 in the subtree whose root is related to v_1. Similarly, if we delete the edge (v_1, v_2), we only need to re-search the leaf node which is related to v_1 in the subtree whose root is related to v_2 or vice versa, the leaf node which is related to v_2 in the subtree whose root is related to v_1. In summary, partial update of some subtrees of the whole search tree is enough. Thus, our algorithm would not be too time-consuming in practice.



2.6 Comparison of Complexity

In this section, we compare our method with the other algorithms for dynamic subgraph matching, including Turboflux,Symbi and SJ-Tree.

The following table shows the comparison of space complexity:

Table 2: Comparison of Complexity

Method	Information Stored	Space Complexity
SJ-Tree	partial matches	$O(\|V(Q)\|\|E(G)\|^{\|E(Q)\|})$
Turboflux	mark of filter on spanning tree of Q	$O(\|V(Q)\|\|E(G)\|)$
Symbi	mark of filter on DAG of Q	$O(\|E(Q)\|\|E(G)\|)$
Ours	mark of search with Success Set and Failure Set	$O(\|V(G)\|^{\|V(Q)\|})$

For our method stores the search tree, the space complexity is the same as the size of the search tree, which is $O(|V(G)|^{|V(Q)|})$ since every vertex in Q, it has at most $|V(G)|$ candidates. From the table, we can though our method stores more than Turboflux and Symbi, the space complexity is less than that of SJ-Tree. Plus, for we store more information than both Turboflux and Symbi, we tend to believe that our algorithm will require less running time than both Turboflux and Symbi, even though they have the same worst case time complexity.

2.7 Extend Gcoding to Dynamic Graph

Gcoding, which is designed by Zou et al. [2008], is an effective spectral encoding method for pruning in subgraph matching. An advantage of Gcoding is that this method has no conflicts with the method based on filter as Symbi and Turboflux, also has no conflicts with the method proposed by us with success set and failure set. Hence, we can apply Gcoding to our method for better performance. However, since Gcoding requires the eigenvalues of the adjacent matrix of a data graph G and that of the Level-n Path Tree. If we directly apply the algorithm in Gcoding for pruning, it would be quite time-consuming since the calculation of eigenvalues takes a lot of time. In this section, we propose a method which does not require the re-computation of the eigenvalues for time efficiency.

In order to solve the above problem, we recall the Interlacing Theorem used in Gcoding.

Interlacing Theorem Given a graph G with n vertices and a graph G' with m vertices $(n \geq m)$, their adjacency matrices are denoted as A and B respectively. For matrix A , its eigenvalues are $\lambda_1(G) \geq \lambda_2(G) \geq \ldots \geq \lambda_n(G)$. For matrix B , its eigenvalues are $\lambda_1(G') \geq \lambda_2(G') \geq \ldots \geq \lambda_m(G')$. If G' is subgraph of G , then $\lambda_{n-m+i}(G) \leq \beta_i(G') \leq \lambda_i(G), (i = 1, \ldots, m)$.

In addition, as in our problem setting, the data graph G_t evolves as time goes by, we are concerned about the change of eigenvalues of the adjacent matrix of the next time stamp G_{t+1} compared to that of G_t. Since $G_{t+1} = G_t + \Delta G$, we study the change of eigenvalue of the adjacent matrix of G_{t+1}. As is discussed before, the vertex operation is only applied to isolated vertexes which are easy to deal with. As for vertex addition, since the added vertex has connection with none of other vertexes, the operation only add an eigenvalue of 0 to the the eigenvalues of adjacent matrix of G_t. Similarly, as for vertex deletion only reduce an eigenvalue of 0 to the the eigenvalues of adjacent matrix of G_t. Hence, we only need to consider the cases of edge operation. The following theorem study the eigenvalues of ΔG in edge operation.

Eigenvalues of ΔG in Edge Operation In terms of edge addition and edge deletion, let n be the size of matrix, the eigenvalues of ΔG consists of multiple 0 of a number of $n - 2$, an unique 1 and an unique -1.

The proof of this theorem can be done by definition of eigenvalues.

With the eigenvalues of ΔG, even if we can not directly know the eigenvalues the adjacent matrix of G_{t+1}, it is easy for us to know that the change on eigenvalues would not be large.

Change of Eigenvalue In terms of edge addition and edge deletion, the change on eigenvalues must satisfy:

$$\sum_{i=1}^{n} (\lambda_i(G_{t+1}) - \lambda_i(G_t))^2 \leq \|\Delta G\|_F^2 = 4$$
$$\max_i \lambda_i(G_{t+1}) - \lambda_i(G_t) \leq \|\Delta G\|_2 = 1$$

The above theorem is important for it tells us that though we can not know the exact value of $\lambda(G_{t+1})$, it will not differ from those of $\lambda(G_t)$ too much. The proof can be done with Hoffman–Wielandt Inequality and Young Theorem.

With the above observations and theorems, we can establish our algorithm for dynamic Gcoding update. The key idea behind the algorithm is that there is no need to maintain the exact values of eigenvalues, just inexact values are enough and the estimation of the eigenvalues would only slightly differ from the exact value. That is to say, we only maintain the lower bound $m_i(G_t)$ and upper bound $M_i(G_t)$ for $\lambda_i(G_t)$, where $m_i(G_t) \le \lambda_i(G_t) \le M_i(G_t)$. With the above estimation of eigenvalues, we can still apply the same Interlacing Theorem on Level-n Path Tree as in Gcoding.

Estimation of Eigenvalue The upper bound and lower bound can be calculated by:

$$m_i(G_{t+1}) = \max\{\lambda_i(G_t) - 1, \lambda_{i-n+1}(G_t) + 1, \max_{1<j<n} \lambda_{i-j+1}(G_t)\}$$
$$M_i(G_{t+1}) = \min\{\lambda_{i+n-1}(G_t) - 1, \lambda_i(G_t) + 1, \min_{1<j<n} \lambda_{i-j+1}(G_t)\}$$

The proof can be done by Wely Inequalities with $\lambda_{i-j+1}(G_t) + \lambda_j(\Delta G) \le \lambda_i(G_{t+1}) \le \lambda_{i+j}(G_t) + \lambda_{n-j}(\Delta G), \forall i, j$

Eigenvalue Disc One problem brought by our method can be that even we guarantee the change of eigenvalues would be slight, the slight error can accumulated when the data graph G evolves for a long time. Hence, we use the eigenvalue disc as a trust region, as we know that:

$$\lambda(G_t) \in \cup_{i=1}^{n} R(G_{t_{ii}}, \rho(G_t) - 1)$$

Thus, when our estimation deviate the trust region given by our eigenvalue disc, we may need to re-compute the eigenvalues of the adjacent matrix. Otherwise, the pruning power of the Interlacing Theorem may greatly discount.

3 Frequent Subgraph Mining

3.1 Problem Definition

Given a single data graph G and a threshold s , the task of frequent subgraph mining is to find all the subgraph Q in data graph G whose support is higher than s. While way of the calculation of support varies by different definition of "support", as long as it conforms to anti-monotonicity. One of the popular definition of support is based on computing maximum independent sets (MIS) in overlap graphs, containing a NP-hard sub-problem. Thus, MNI (Minimal Image based support) is proposed by Bringmann and Nijssen [2008], which is more time-efficient than MIS based support. Hence, in our project, we use MNI for the definition of support.

3.2 Method Overview

Our method is based on Gspan Yan and Han [2002], a famous algorithm for frequent subgraph mining over graph dataset. Gspan builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, Gspan adopts the depth-first search strategy for mine frequent subgraphs efficiently. However, the problem is that our problem is to mine the frequent subgraphs over a single large graph instead of over a graph dataset with multiple graphs. Hence, if we apply the same method of Gspan to our problem, we can not get the correct answer as we expect.

Luckily, if we notice the relationship between the two problems, we will find that we only need to modify Gspan slightly then it can support our problem frequent subgraph mining over a large single data graph G. To be more specific, if we regard the graph dataset consists of only a single data graph G and still apply Gspan on this graph dataset with a threshold of 1, the program in Gspan will "grow" the pattern with DFS code, where all the patterns are one subgraph matching over G. All we need to do is to calculate the MNI and verify if it is higher than our threshold of MNI s. If so, we let the pattern "grow" to a bigger pattern and verify again; If not, we can stop the procedure of pattern growing in this search.

In short, in each depth-first search, we get a pattern as in Gspan. And we our algorithm differ from Gspan is that we need to regard the pattern as a query graph Q and calculate MNI with subgraph matching algorithms. The modification may seem slight but it may make a big impact of the algorithm for we know that subgraph matching is a NP-hard problem. Hence, we need to reduce the time complexity in this problem, rather than simply apply any subgraph matching problem such as DP-iso.

3.3 Introduce Filter Into Subgraph Mining

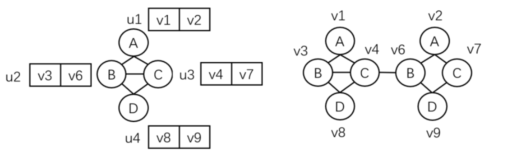
As is known, filter is an important pruning technique in subgraph matching problem and we also see the relationship between subgraph matching and frequent subgraph mining. However, we find out that most of the existing methods for frequent subgraph mining do not use the technique of filter as in subgraph matching, instead, they focus on the procedure of searching and pattern growing such as Gspan. As related work analyses above problems a lot, we focus on how to introduce filter into the problem of frequent subgraph mining and we wonder the effect of filter on this problem.

The idea of filter is to prune some candidates based on local check. And more importantly, when one candidate is pruned, it may make it possible for us to prune another candidate. Thus, the process can be applied recursively until we only prune any of the candidate based on our check. The filter is quite similar to the Arc-Consistent Check in CSPs (Constrain Satisfaction Problems), since we know that the problem of subgraph matching is just a special case of CSP.

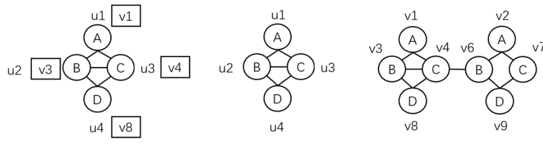
Simple Filter When searching, we can notice that the same filter which is used in subgraph matching can be directly applied to our problem since subgraph matching is just a sub-problem of it. However, this simple filter may be sub-optimal to this problem for it only consider the constrains given by subgraph matching , but ignore the constrains given by frequency. That is to say, if we only use the same filter in subgraph matching, the effect of filter will not differ with different settings of s.

Support Filter In consideration if the constrains given by frequency, we propose a filter called support filter which is based on the relationship of MNI and the number of candidates of a vertex in query graph or pattern graph Q. The support filter is that we can examine each $u \in V_Q$ and the number of its candidates $N(u)$, when we find that $N(u) < s$, we can know that the MNI must also be less than s and we can terminate the growth of the pattern. The idea of support filter is simple but it may be quite effective since it can terminate all the search in subgraph matching as it prunes all the candidates of the vertex u at the same time while the simple filter can only prune one of them each time.

The following example demonstrates the effect of Filter and Support Filter with the MNI threshold $\tau = 2$:



Without Filter Graph



With Filter With Support Filter Graph

We can see that with the filter used in subgraph matching, we can prune the candidate pairs $(u_1, v_2), (u_{2v6}), (u_3, v_7)$ and (u_4, v_9) before searching the results of subgraph matching, i.e. half of the data graph G can be ignored in the following search. And we can even go further with support filter proposed by us, with which we can continue to prune the candidate pairs (u_1, v_1), (u_2, v_3) , (u_3, v_4) and (u_4, v_8), i.e. the whole data graph G can be pruned or the search can be terminated, reducing a lot of redundant computation.

Algorithm 1 Subgragh Mining With Filter

```
Input: A data graph $G$ and threshold $s$
Output: All the subgraphs $Q$ of $G$ whose support is greater than $s$
1: results = $\emptyset$
2: $E$ = GenerateFrequentEdges($G, s$)
3: for $e \in E$ do
4:     results = results $\cup$ SubgraphExtensionWithFilter($e, G, s, E$)
5: end for
6: return results
```

3.4 Introduce Dynamic Subgraph Matching

It is worth noting that the algorithm of dynamic subgraph matching can be directly applied to the method based on Gspan. Consider the following example, we know that whenever a pattern "grows", it is equivalent to the case that an edge is added or both a vertex and an edge are added in dynamic subgraph matching. In the example, when we

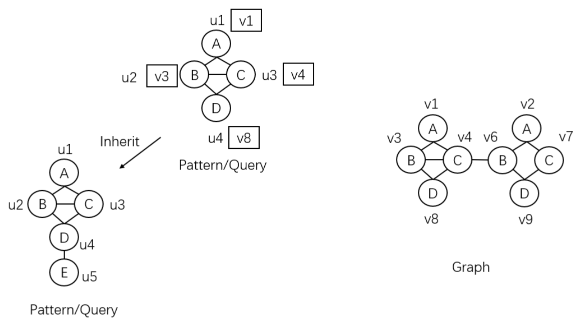
Algorithm 2 Subgraph Extension With Filter

```
Input: A subgraph $Q$ of data graph $G$, support threshold $s$, the set of frequent edges $E$
Output: All the subgraphs of $G$ as an extension of $Q$ whose support is greater than $s$
 1: candidates = $\emptyset$
 2: for $e \in E, u \in V(Q)$ do
 3:     if $e$ can be used to extend $u$ then
 4:         new_candidate = Extend($e, u, Q$)
 5:         if HaveGenerate(new_candidate) is false then
 6:             candidates = candidates $\cup$ new_candidate
 7:         end if
 8:     end if
 9: end for
10: results = $\emptyset$
11: for $Q \in$ candidates do
12:     $t$ = CalculateFrequencyWithFilter($Q, G$)
13:     if $t > s$ then
14:         results = results $\cup$ SubgraphExtensionWithFilter($Q, G, s, E$)
15:     end if
16: end for
```

Algorithm 3 Calculate Frequncy With Filter

```
Input: A query graph $Q$ and a data graph $G$.
Output: The frequency $s$ of $Q$ over data graph $G$
 1: for $u \in Q$ do
 2:     candidates[$u$] = InitCandidates($u, Q, G$)
 3: end for
 4: queue = $\emptyset$
 5: while isEmpty(queue) is false do
 6:     candidates[$u$] = Pop(queue)
 7:     if Filter(candidates[$u$],$Q, D$) is true then
 8:         if sizeof(candidates[$u$]) < $s$ then
 9:             return 0 // use the support filter
10:         end if
11:         for $v \in$ Neighbor($u$) do
12:             Push(queue, $v$) // recursively use the simple filter
13:         end for
14:     end if
15: end while
16: embedding = BacktrackSearch(candidates)
17: $s$ = CalculateMNI(embedding)
18: return $s$
```

calculate MNI through subgraph matching, we can inherit the subgraph matching results from the previous pattern and then add vertex u_5 and edge (u_4, u_5) in the new pattern. Thus, we can use any one of the algorithm for dynamic subgraph matching, including SJ-Tree, Turboflux, Symbi or our method with Success Set and Failure Set to solve this problem. However, it is worth noticing that here the problem is slightly different from the original dynamic subgraph matching problem since now the query graph Q evolves over time instead of the data graph G. Therefore, we need to modify the algorithms to meet new needs here.



However, while the problem may be a little different, the key techniques are all the same. It is a pity that we did not implement the code of subgraph mining with dynamic acceleration, for we use Python for subgraph mining problem but the existing codes for dynamic subgraph matching are written with C++. To compensate that pity, We put the Pseudo code as follows:

Algorithm 4 Subgragh Mining With Dynamic Acceleration

```
Input: A data graph $G$ and threshold $s$
Output: All the subgraphs $Q$ of $G$ whose support is greater than $s$
1: results = $\emptyset$
2: $E$ = GenerateFrequentEdges($G, s$)
3: for $e \in E$ do
4:     $t_0$ = CalculateEdgeFrequency($e, G$)
5:     results = results $\cup$ SubgraphExtensionWithDynamicAcceleration($e, G, s, E, t_0$)
6: end for
7: return results
```

Algorithm 5 Subgraph Extension With Dynamic Acceleration

```
Input: A subgraph $Q$ of data graph $G$, support threshold $s$, the set of frequent edges $E$, the frequency $t_0$ of $Q$
Output: All the subgraphs of $G$ as an extension of $Q$ whose support is greater than $s$
1: results = $\emptyset$
2: candidates = $\emptyset$
3: for $e \in E, u \in V(Q)$ do
4:     if $e$ can be used to extend $u$ then
5:         new_candidate = Extend($e, u, Q$)
6:         if HaveGenerate(new_candidate) is false then
7:             $t$ = $t_0$ + DynamicSubgraphMatchingFrequency($e, Q, G$)
8:             if $t > s$ then
9:                 results = results $\cup$ SubgraphExtensionWithDynamicAcceleration($Q, G, s, E, t$)
10:            end if
11:        end if
12:    end if
13: end for
```

3.5 Experiment

We implement a Gspan-like algorithm with Python for frequent subgraph mining over a single large data graph with both simple filter and support filter. We study the effect of both simple filter and support filter on YEAST dataset which is available at https://github.com/RapidsAtHKUST/SubgraphMatching. YEAST dataset consists of a single large graph with 3112 vertexes and 12519 edges, where the average degree of each vertex is 8. The following figure shows the running time of algorithm with the support threshold $s = 470$.

Table 3: Experiment results on filters

Simple Filter	Support Filter	Running Time
×	×	25.09s
×	✓	18.45s
✓	✓	**13.95s**

From the table, we can see the two filter proposed by us do work and they both improve the performance of frequent subgraph mining a lot.

4 Similar Subgraph Mining

4.1 Problem Definition

Given a single data graph G and a threshold τ and a pattern graph P , the task of similar subgraph matching (or similar subgraph mining) is to find all the subgraphs Q in data graph G whose graph edit distance (GED) with P is less than τ,i.e. which are similar enough to pattern graph P. In some scenarios, the number of all the subgraphs Q which satisfy the condition is too large, so the problem may be relaxed as to find out the top-k subgraphs Q whose graph edit distance is less than τ.

4.2 GED Computation with GNN

Our method of computing GED is inspired the related work Noah Yang and Zou [2021], which uses the neural-optimized A* search algorithm to compute GED, we take the advantage of neural network by directly training the GNN to learn the GED between two graphs.

The architecture of the network are as follows. Firstly the graph pairs(e.g.G_1, G_2) in the training set are embedded simply by the information of vertices, nodes and labels, then each graph and cross-graph information are sent into GIN(graph isomorphism network) and self-attention part. Then the features are sent into MLP and get a score of the similarity between the graph pairs. To ensure the robustness of the network and avoid the effect of the graph scale, the similarity score is a value between 0 and 1, then the score is multiplied by the upper bound of GED between the graph pair, which is the sum of the maximum number of edges and the maximum number of nodes within these two graphs. Thus we get the predicted GED between the graph pairs.

In general the network is trained to get the accurate GED, thus the mean square error between the predicted GED and the ground truth is applied for loss to supervise the learning of the network.

However in some scenarios which we only need the top-k results which means the rank is rather important than the accuracy of the GED value. Inspired by the work of RankNet Burges C [2005], in previous work, we've made attempt which uses the rank loss in Ranknet, which leads the network to pay attention to learning the rank, to supervise the learning of the network.

In our previous experiment in project 3, we implemented the code of similar algorithm which is conducted on data base. On AIDS dataset, Spearman rank correlation is improved by the combination of Ranknet to 0.58 with the original network just get the correlation of 0.09939. The experiment shows the effectiveness of the application of Ranknet thus lead our network to better predict the rank of GED.

4.3 Method Overview

Our idea is to break this problem into smaller parts we've solved. The first stage is to find the candidates of the subgraph in data graph. As the labels significantly increase the complexity of finding the subgraph, we omit labels in the process of getting candidates and only consider nodes and vertices. We consider the subgraphs of the pattern graph whose GED is less than τ compared to original pattern graph. As the pattern graph is always small and the labels are omitted, the number of subgraphs satisfying our condition will be not too much. Then we use the subgraph matching algorithm mentioned in the first section to find the subgraphs of data graph which is isomorphic with these subgraphs of pattern graph and get our candidates. Thus these candidates' GED with pattern graph are all less than τ in condition of ignoring labels.

The reason why we don't find the subgraph candidates directly but find the subgraph of pattern graph and get their matching subgraph is that the matching process is more precise and faster than finding the subgraph candidates, which is to search with a vague condition of GED but without the precise whole restriction of nodes and vertices.

And in the second stage the GED between pattern graph and the candidates are computed by the network introduced in last part. The application of neural network greatly speed up the computation of GED. Thus we get the subgraphs whose graph edit distance is less than τ. Or if the top-k subgraphs is wanted in case the number of all the subgraphs Q which satisfy the condition is too large, we could use the rank loss to supervise the network and get the top results.

Algorithm 6 Similar Subgraph Mining

```
Input: A query graph $Q$ and a big graph $G$.
Output: All subgraphs of $G$ whose GED to $Q$ is less than $k$.
1: Firstly ignore all labels of $Q$ and $G$.
2: Generate all subgraph of $Q$ whose GED with $Q$ is less than $k$. After this process, we get graph database $D$.
3: for each graph $g_1$ in $D$ do
4:     Use subgraph isomorphism algorithm to search $g_1$ in $G$
5: end for
6: We get all similar subgraph candidates called database $C$.
7: Restore labels of $G$ and graphs in $C$.
8: Use GNN network+mse loss to find all answers.
```

Algorithm 7 Top-k Similar Subgraph Mining

```
Input: A query graph $Q$ and a big graph $G$.
Output: Top $k$ subgraphs of $G$ similar to $Q$.
1: Firstly ignore all labels of $Q$ and $G$.
2: Generate all subgraph of $Q$ whose GED with $Q$ is less than $k$. After this process, we get graph database $D$.
3: for each graph $g_1$ in $D$ do
4:     Use subgraph isomorphism algorithm to search $g_1$ in $G$
5: end for
6: We get all similar subgraph candidates called database $C$.
7: Restore labels of $G$ and graphs in $C$.
8: Use GNN network+rank loss to find all answers.
```

5 Frequent Similar Subgraph Mining

5.1 Problem Definition

Given a single large data graph D and a pattern graph P, the task of frequent similar subgraph mining is to find out all the subgraphs Q which are similar to P (or equivalently whose graph edit distance is less than τ) and meanwhile whose support is greater than s.

Note that the problem of frequent similar subgraph mining is a new problem proposed by us.

In this new problem, need to find the similar and frequent subgraphs over G at the same time. This problem can be regarded as a combination of similar subgraph mining and frequent subgraph mining. However, we know that both these two problems are NP-hard. The combination of them are extremely hard! In this section, we show our method can reduce the new problem into much simpler original problems.

5.2 Method Overview

Our idea is to reduce the new problem into original problems. On the one hand, We know that the calculation of support is relatively much easier than the calculation of GED. On the other hand, our method in similar subgraph mining is able to give the results of all the similar subgraphs Q over the data graph G. Based on the above analysis, the naive method can be a two-stage method: First, find out all the similar subgraphs Q to be candidates with similar subgraph mining method. Second, given the candidates found out by stage-one, calculate the frequency of occurrence of each candidate Q and the ones with a support greater than s are the outputs.

However, the naive two-stage method may be too time-consuming because the number of candidates Q generated by stage-one may be too large. Additionally, as stage-one is equivalent to the task of similar subgraph mining, we focus on accelerate stage-two to improve the performance of the algorithm. The improvement is mainly given by the two following techniques:

First, as the counting of support contains a maybe NP-hard problem: graph matching or graph isomorphism, since we need to verify whether two candidates Q are the isomorphic. Using classical algorithms for graph isomorphism requires a lot of time. Motivated by existing work which use graph neural networks to calculate an approximate isomorphism to save time and the observation that the problem of problem of graph isomorphism is exactly the same problem of judging whether the graph edit distance between two graphs are 0. Hence, we use the same graph neural networks for GED to test whether two graphs are isomorphism.

Second, though we can use graph neural network to reduce the time, the number of subgraphs Q would be unbearable. With the aim of improving the efficiency of our algorithm, we use a sample based framework. We show that it is unnecessary to enumerate all the candidates Q for the calculation of support, all what we need is to sample a number of Q with a much smaller sample size m which also guarantees a approximately support calculation.

5.3 Sample Based Framework

Our sample based framework relies on the firm foundation of statistical learning theory of VC-dimension (or Vapnik–Chervonenkis dimension) and ϵ-sample. The ϵ-sample theory tells us the sample size m which can guarantees a approximately support calculation of absolute error less than ϵ. Since the sample size m calculated by ϵ-sample theory is dependent on the complexity of hypothesis space of the neural networks we use to test the graph isomorphism. We also need to calculate the VC-dimension of the networks for a measurement of the complexity of hypothesis space, which is easy to be done using the relationship of VC-dimension and the growth function of hypothesis space.

Algorithm 8 Sampled Based Frequent Similar Subgraph Mining

```
Input: A data graph $G$ , support threshold $s$, GED threshold $\tau$, a pattern graph $P$, number of parameters of GNN $N$
Output: All the subgraphs $Q$ of $G$ whose support is greater than $s$ and GED with $P$ less than $\tau$
1: candidates = SimilarSubgraphMining($P, G, \tau$)
2: m = CalculateSampleSize($N, \delta, \epsilon$)
3: sampled_candidates = DrawSample(candidates, $m$)
4: Calculate the frequencies of each candidate among all the sampled_candidate graphs
5: Let the results be the graphs which are frequent in the sample
6: return results
```

5.3.1 Definition and Lemma

Growth Function

Before introducing the VC-dimension, the definition of the growth function needs to be given, which is defined as the maximum possible number of labels that can be given by the hypothesis space $\mathcal{H}$ for the sample set D of size m. In question, the token is $+1, -1$, and the growth function is defined as:

$$\Pi(m) = \max_{|D|=m} |\mathcal{H}_D| = \max_{|D|=m} [|(h(x_1), h(x_2), ...h(x_m))|, h \in \mathcal{H}]$$

VC-Dimension

If the hypothesis space can achieve all possible labeling outcomes for a sample set of size m, it is called Shattering the sample set. The VC dimension is defined as the maximum size of the sample set that the hypothesis space can be broken up, which is:

$$VC(\mathcal{H}) = \max_m [\Pi(m) = 2^m]$$

Sauser's Lemma

Sauser's Lemma gives the relationship between the growth function and the VC-dimension, because in many cases we pay more attention to the VC-dimension or the upper bound of the VC-dimension, but the VC-dimension is more difficult to calculate while the computation of the growth function may be simple by definition.

$$\Pi(m) \leq \sum_{i=0}^{d} \binom{m}{i}, \forall m, d = VC(\mathcal{H})$$

Furthermore, when $m \geq d$, that is, when the sample size is greater than the VC-dimension, the upper bound of the growth function can be obtained as:

$$\Pi(m) \leq (\frac{em}{d})^d \leq (em)^d$$

The proof can be done using mathematical induction. First, assume that the above formula holds for $m-1$, and consider the data set D corresponding to the growth function that makes the sample size m, and the data set corresponds to all possible The number of markers in can actually be represented by the number of markers in two sub-datasets of size $m-1$. Then, it is known that $\mathcal{H}_D = \{h(x_1), h(x_2), ..., h(x_m)\}$, for the case where $h(x_m)$ has only one value, define $\mathcal{H}_{D_1} = \{h(x_1), h(x_2), ..., h(x_{m-1})\}$ .And for $h(x_m)$ there are two values, define accordingly $\mathcal{H}_{D_2} = \{h(x_1), h(x_2), ..., h(x_{m-1})\}$. For $\mathcal{H}_{D_2}$, since it satisfies the restriction that the last dimension of the sample set is broken up, its VC-dimension should be smaller than the original, and at most $d-1$, according to which induction can be done by:

$$\begin{aligned}
\Pi(m) &= |\mathcal{H}_D| \\
&= |\mathcal{H}_{D_1}| + |\mathcal{H}_{D_2}| \\
&\leq \sum_{i=0}^{d} \binom{m-1}{i} + \sum_{i=0}^{d-1} \binom{m-1}{i} \\
&= 1 + \sum_{i=1}^{d} \binom{m-1}{i} + \sum_{i=1}^{d} \binom{m-1}{i-1} \\
&= 1 + \sum_{i=1}^{d} (\binom{m-1}{i} + \binom{m-1}{i-1}) \\
&= 1 + \sum_{i=1}^{d} \binom{m}{i} \\
&= \sum_{i=0}^{d} \binom{m}{i}
\end{aligned}$$

If $m \geq d$, i.e. when the sample size is greater than the VC-dimension, the upper bound of the growth function can be obtained,

$$\begin{aligned}
\Pi(m) &\leq \sum_{i=0}^{d} \binom{m}{i} \\
&\leq (\frac{m}{d})^d \sum_{i=0}^{d} \binom{m}{i} (\frac{d}{m})^i \\
&\leq (\frac{m}{d})^d \sum_{i=0}^{m} \binom{m}{i} (\frac{d}{m})^i \\
&= (\frac{m}{d})^d (1 + \frac{d}{m})^m \\
&\leq (\frac{m}{d})^d e^d \\
&= (\frac{em}{d})^d \\
&\leq (em)^d
\end{aligned}$$

5.3.2 VC-Dimension of Neural Networks

In this section, we focus on the VC-dimension of the networks we used to test graph isomorphism. We begin at the VC-dimension of linear hyperplane, which is also the basic unit in feed forward neural networks or multi-layer linear perceptions, i.e. MLP.

Linear Hyperplane Classifier

The expression of linear hyperplane classifiers (eg. Logistic Regression, SVM) can be:

$$h(x) = \text{sign}(w^T x + b)$$

whose VC-dimension can be obtained according to the definition:

On the one hand, there is a dataset D with sample size $d+1$ such that it can be broken up by a linear hyperplane, where e_i is a vector where only the i_{th} element is 1 while the others are all 0, and (x_i, y_i) represents the pair samples from dataset D:

$$\begin{aligned} \text{Given } x_0 = 0, x_1 = e_1, x_2 = e_2, ...x_d = e_d \\ \forall y_0, y_1, ...y_d, \text{Let } w = [y_1, y_2, ..., y_d]^T, b = y_0 \\ \text{Then } h(x_i) = \text{sign}(w^T x_i + b) = y_i \end{aligned}$$

On the other hand, for any dataset D with a sample size of $d+2$, even if the zero vector is removed, there will still be at least $d+1$ linearly dependent vectors in the d dimension space, The following proves that for any $x_0, x_1, ..x_{d+1}$, it must not be broken up by the linear hyperplane, let's assume $x_0 = 0$, and then use the proof by contradiction. The proof begins with the fact that since all the x_is are linear dependent, there exist a set of nonzero α_is and then x_1 can be expressed by a linear combination of $x_2, x_3, ...x_{d+1}$:

$$\begin{aligned} x_1 &= \sum_{i=2}^{d+1} a_i x_i, \exists a_i \neq 0, \\ w^T x_1 + b &= \sum_{i=2}^{d+1} a_i (w^T x_i + b) \\ \text{Let } y_1 &= -1, y_i = \text{sign}(a_i), \forall 2 \leq i \leq d+1 \\ \text{sign} h(x_1) &= \text{sign}(w^T x_1 + b) = \text{sign}(\sum_{i=2}^{d+1} a_i (w^T x_i + b)) = 1 \neq -1 \end{aligned}$$

Therefore, we prove that there is a dataset D with sample size $d+1$ such that it can be broken up by a linear hyperplane, and any dataset D with sample size $d+2$, its It must not be broken up by the linear hyperplane, that is, the VC-dimension of the linear hyperplane classifier is $d+1$.

MLP

Though the network we use in our algorithm is much more complex than MLP. To be more precise, it is a combination of graph neural network units (eg. GCN, GIN, GraphSARE) , attention mechanisms and MLP. The last layer, which outputs the approximate GED, is still the MLP. Thus, we can fix the other layers before MLP layer and fine-tune with merely the MLP layer, which reduce the complexity of model. In this way, we can only calculate the VC-dimension of MLP to represent the VC-dimension of the whole network.

In practice, the MLP uses Relu as the activation function in our network, however, our analysis is based on MLP activated by sign function. It is worth mentioned that this simplification is reasonable because the proof of universal approximation theorem of neural networks also deeply relies on this simplification. In addition, at the limit, classical activation function, such as Sigmoid function also has the same behavior with sign the function.

The MLP $\mathcal{F}$ activated by the sign function is actually a composite function, assuming that it has a total of h layers, each layer has n_i neurons, and each neuron is a unit which can be expressed by $\text{sign}(w^T x + b)$. Then, MLP is equivalent to the composition of h functions, and the range of each layer function h_i is equivalent to the Cartesian product of n_i

neurons. Hence, the MLP can be decomposed into the combination of the Cartesian product and the composite function of each neuron. The following proof tells us the VC-dimension with he Cartesian product and the composite function of neurons , where d_{ij} represents the number of parameters of the j neuron in the i layer, and N represents the total number of parameters of the entire neural network, m the maximum sample size that MLP can shatter:

$$\begin{aligned}
VC(\mathcal{F}) &= 2^m \\
&= \Pi_{f_1 \circ f_2 \circ \ldots \circ f_h}(m) \\
&\le \prod_{i=1}^{h} \Pi_{f_i}(m) \\
&= \prod_{i=1}^{h} \Pi_{g_{i,1} \times g_{i,2} \times \ldots \times g_{i,n_i}}(m) \\
&\le \prod_{i=1}^{h} \prod_{j=1}^{n_i} \Pi_{g_{ij}}(m), \text{With } g_{ij}(x) = \text{sign}(w_{ij}^T x + b_{ij}) \\
&\le \prod_{ij} (em)^{d_{ij}}, \text{With } d_{ij} = VC(g_{ij}) \\
&= (em)^N, \text{Let } N = \sum_{ij} d_{ij}
\end{aligned}$$

The inequality about its VC dimension m can be obtained,

$$\begin{aligned}
2^m &\le (em)^N \\
m &\le N \log em \\
em &\le eN \log em \\
\frac{em}{eN \log em} &\le 1
\end{aligned}$$

By contradiction, according to the analysis of the upper bound of complexity, we can get $em = O(N \log N$, otherwise, $em > KN \log N, \forall K, N \to \infty$. Then we know that,

$$\begin{aligned}
1 &\ge \frac{em}{eN \log em} \\
&> \frac{KN \log N}{eN \log K + eN \log N + eN \log \log N} \\
&> \frac{KN \log N}{2eN \log N} \\
&= \frac{K}{2e} > 1, \text{Let } K > 2e
\end{aligned}$$

which is impossible. Therefore, the upper bound of the VC-dimension of MLP can be obtained as $O(N \log N)$

5.3.3 ϵ-Sample Theorem

Calculation of the actual occurrence frequency of each candidate graph Q among a large number of candidates can be time-consuming. However, we know that if we draw a sample from the candidate set and calculate the occurrence frequency of each candidate graph Q in the sample, the sampled occurrence frequency will converge to the actual occurrence frequency when the sample size m is large enough. Further, the ϵ-sample theorem tells us that a ϵ-approximation of occurrence frequency can be maintained with a relatively small m.

In this section, we give the formula which tells us the relationship between m, ϵ and the VC-dimension of network d. Given a candidate graph Q, we can get a hypothesis function $h(Q)$, denoted by h in the following proofs, and $h(X)$ tells whether another candidate graph X is isomorphic to Q. The ϵ-sample theorem tells us for all the hypothesis h in the hypothesis space $\mathcal{H}$ (i.e. for all the candidate graphs Q), we can approximate the occurrence frequency $R(h)$ among all the candidates with the frequency $\hat{R}(h)$ calculated using a small sample with a size m.

The proof is given by previous work as in Vapnik and Chervonenkis [2015], however, the original proof is quite complex and technical and here we provide a much simpler proof using the same symmetry lemma:

Symmetry Lemma

In order to associate the upper bound VC-dimension of the ϵ-bound , it is necessary to associate it with the growth function, which can be done using the following symmetry lemma.

The data set to be analyzed is D, and if there exists an independent and identically distributed data set D' (which is also called a ghost dataset), the following conclusions can be drawn:

$$P(\sup_{h\in\mathcal{H}} |R(h) - \hat{R}_D(h)| > \epsilon) \leq 2P(\sup_{h\in\mathcal{H}} |\hat{R}_{D'}(h) - \hat{R}_D(h)| > \frac{\epsilon}{2}), \text{With } m\epsilon^2 \geq 2$$

Let h to be the hypothesis which achieve the supremum of the lef formula and now we only need to prove that:

$$P(|R(h) - \hat{R}_D(h)| > \epsilon) \leq 2P(\sup_{h'\in\mathcal{H}} |\hat{R}_{D'}(h') - \hat{R}_D(h')| > \frac{\epsilon}{2})$$

According to the triangle inequality,

$$I[|R(h) - \hat{R}_D(h)| > \epsilon]I[|\hat{R}_{D'}(h) - R(h)| \leq \frac{\epsilon}{2}] \leq I[|\hat{R}_{D'}(h) - \hat{R}_D(h)| > \frac{\epsilon}{2}]$$

Where I denotes the indicative function. Now take the expectation of the above formula and we can get,

$$P(|R(h) - \hat{R}_D(h)| > \epsilon)P(|\hat{R}_{D'}(h) - R(h)| \leq \frac{\epsilon}{2})) \leq P(|\hat{R}_{D'}(h) - \hat{R}_D(h)| > \frac{\epsilon}{2}))$$

And according to Chebyshev's inequality,

$$\begin{aligned} P(|\hat{R}_{D'}(h) - R(h)| \leq \frac{\epsilon}{2})) &\geq 1 - \frac{4Var[\hat{R}_{D'}(h)]}{\epsilon^2} \\ &= 1 - \frac{4Var[R(h(x))]}{m\epsilon^2} \\ &= 1 - \frac{4E[R^2(h(x))] - 4E[R(h(x))]^2}{m\epsilon^2} \\ &\geq 1 - \frac{4E[R(h(x))] - 4E[R(h(x))]^2}{m\epsilon^2} \\ &= 1 - \frac{4E[R(h(x))](1 - E[R(h(x))])}{m\epsilon^2}, \text{By } R(h(x)) \leq 1 \\ &\geq 1 - \frac{1}{m\epsilon^2} \\ &\geq \frac{1}{2}, \text{By } m\epsilon^2 \geq 2 \end{aligned}$$

Finally we can prove the lemma,

$$P(|R(h)-\hat{R}_D(h)|>\epsilon)\leq 2P(|\hat{R}_{D'}(h)-\hat{R}_D(h)|\leq\frac{\epsilon}{2}))\leq P(\sup_{h'\in\mathcal{H}}|\hat{R}_{D'}(h')-\hat{R}_D(h')|>\frac{\epsilon}{2})$$

ϵ-Bound

Combined Hoeffding's inequality, the above symmetric lemma and Sauser's lemma of VC-dimension, we can conclude the ϵ-bound:

$$\begin{aligned}
P(\sup_{h\in\mathcal{H}}|R(h)-\hat{R}_D(h)|>\epsilon) &\leq 2P(|\hat{R}_{D'}(h)-\hat{R}_D(h)|>\frac{\epsilon}{2}))\\
&\leq 2P(\sup_{h\in\mathcal{H}}|\hat{R}_{D'}(h)-\hat{R}_D(h)|>\frac{\epsilon}{2})\\
&\leq 2\sum_{h\in\mathcal{H}}P(|\hat{R}_{D'}(h)-\hat{R}_D(h)|>\frac{\epsilon}{2})\\
&= 2\Pi(2m)P(|\hat{R}_{D'}(h)-\hat{R}_D(h)|>\frac{\epsilon}{2})\\
&\leq 4\Pi(2m)\exp(-\frac{m\epsilon^2}{2}),\text{By Hoeffding's Inequality}\\
&\leq 4(\frac{2em}{d})^d\exp(-\frac{m\epsilon^2}{2}),\text{By Sauser's Lemma}
\end{aligned}$$

Therefore,

$$P(|R(h)-\hat{R}(h)|\leq\sqrt{\frac{2d\log\frac{2em}{d}+2\log\frac{4}{\delta}}{m}})\geq 1-\delta,\forall h\in\mathcal{H}$$

Hence, given δ, the following formula can be satisfied with a high probability of $1-\delta$, for all $h\in\mathcal{H}$:

$$|R(h)-\hat{R}(h)|\leq\sqrt{\frac{2d\log\frac{2em}{d}+2\log\frac{4}{\delta}}{m}}$$

And we know that as for the network, the VC-dimension d is $O(N\log N)$ where N is the number of parameters in the MLP layer.

If we want to get an ϵ-bound, we need m be large enough , given by,

$$m\geq\max\{\frac{2e}{d},\frac{4}{\delta},\frac{\epsilon^2\log m}{4d+2}\}$$

Even we need to compare m and $\log m$ in the formula, it is easy to verify since we know the function $f(x)=\frac{\log(x)}{x}$ is monotonically decreasing. Therefore, we can use binary search to find out m which satisfy the condition.

Then, with the chosen sample size m, we can have,

$$|R(h)-\hat{R}(h)|\leq\sqrt{\frac{2d\log\frac{2em}{d}+2\log\frac{4}{\delta}}{m}}\leq\sqrt{\frac{(4d+2)\log m}{m}}\leq\epsilon$$

Which implies the ϵ-bound can be obtained with the sample size m.

6 Conclusion

In a nutshell, we have carried out a series of extensions around the subgraph matching problem, and made improvements based on research and existing methods, implemented code in some parts and conducted comparative experiments. The table below summarizes what we have done on each issue. There is still a lot that we can improve and enrich, and we also believe that this entire series of issues of subgraph matching deserves more exploration and innovation.

Table 4: Conclusion of our finished work

Section	Issues	Research and Algorithm	Code and Experiment
1	Subgraph Matching	✓	✓
2	Dynamic Subgraph Matching	✓	
3	Frequent Subgraph Mining	✓	✓
4	Similar Subgraph Matching	✓	
	GED computation with GNN	✓	✓
5	Frequent Similar Subgraph Mining	✓	

References

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.

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.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.

Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 411–426, 2018.

Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. Symmetric continuous subgraph matching with bidirectional dynamic programming. *arXiv preprint arXiv:2104.00886*, 2021.

Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 181–192, 2008.

Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 858–863. Springer, 2008.

Lei Yang and Lei Zou. Noah: Neural-optimized a* search algorithm for graph edit distance computation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 576–587. IEEE, 2021.

Renshaw E Lazier A Deeds M Hamilton N Hullender G Burges C, Shaked T. Learning to rank using gradient descent. In *InProceedings of the 22nd international conference on Machine learning*, pages 89–96. ICML, 2005.

Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pages 11–30. Springer, 2015.