Tackling Subgraph Isomorphism: Implementation and Optimization of Algorithms

Shuchen Li
1800012789

Wenhao Tang
1800013088

Chang Wang
1800017793

Abstract

As a classical problem in graph theory, testing graph isomorphism has been extensively studied since it came out in 1970s, concerning about both its theoretical complexity and practical algorithms. However, no polynomial-time algorithm exists so far for it. Therefore, there are various previous studies trying establishing efficient useable algorithms for this test. In this project, we implement a number of algorithms for graph (sub)isomorphisms that are claimed to be efficient for different tasks and conditions. We propose possible optimization routines to accelerate these algorithms. Finally, we conduct thorough experiments to evaluate the performances of existing algorithms and the effects of our proposed optimizations.

1 Introduction

As a classical problem in graph theory, testing graph isomorphism has been extensively studied since it came out in 1970s, concerning about both its theoretical complexity and practical algorithms. Testing (sub)graph isomorphism also shows great importance in practical scenarios, such as bioinformatics and electronic design automation. Unfortunately, it is still an open problem whether testing graph isomorphism is NP-complete or not, and no existing polynomial-time algorithm is known, though Babai (Babai, 2015) claims to find a quasi-polynomial-time algorithm for GI. On the other hand, subgraph isomorphism is easily proven to be NP-complete. Therefore, practically useable algorithms for these two problems are proposed, analysed and tested. Most of the existing algorithms are based on backtracking, with introduction of numerous pruning techniques and heuristics for searching.

Now, we shall first formally introduce the problem of (sub)graph isomorphism and the problems that we are concerned throughout this report.

Definition 1 (Graph Isomorphism)**.** An isomorphism of graphs with vertex labels G and H is a label-preserving bijection f between the vertex sets of G and H, such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

The problem associated to Definition 1 is testing whether two graphs are isomorphic to each other.

Definition 2 (Subgraph Isomorphism)**.** If there is a graph isomorphism mapping between a graph G and a graph H's certain subgraph, we say there is a subgraph isomorphism between G and H.

The problem associated to Definition 2 is testing whether a graph is subgraph-isomorphic to another graph. Besides concerning the binary relationships between two graphs, we are also interested in the following problem: given a database of graphs and a query graph, find out all the graphs that the query graph is subgraph-isomorphic to.

We have read some papers about subgraph isomorphism (Ren and Wang, 2015; Shang et al., 2008; Lee et al., 2012; Zhao and Han, 2010; Ullmann, 1976; Cordella et al., 2004), and implemented most of them. We found that most of the algorithms follow a basic backtracking routine and apply

Project report for Seminar of Algorithm Analysis, Spring 2020.

various optimizations to speed up the searching process. We shall point out that for different practical problems, different algorithms should apply.

This report is briefly organised as follows. We sketch the frameworks of our proposed optimizations, and then analyse them, along with presentation of the algorithms that inspire us. The existing algorithms and proposals are carefully examined in our experiments where we run them on various data-sets of distinct nature. We conclude from the experiments with effects of these algorithms and optimizations.

We can summarize these optimizations into three key ideas:

1. To reduce the size of the candidate sets of the nodes in the query graph. (Section 2)
2. To find an optimal or near optimal searching order of the query graph. (Section 3)
3. To reduce the number of duplicated structures of data graphs. (Section 4)

2 Reducing the Size of the Candidate Set

Suppose we have two graphs P and G and we want know if P is subgraph isomorphic to G. For every vertex u of P, we have a candidate set $\Phi(u) \subset V(G)$ (where $V(G)$ is the set of vertices of G) which is the set of candidate vertices of G that u may be mapped to.

Therefore, the total search space of the subgraph isomorphism problem is $\Phi(u_1) \times \cdots \times \Phi(u_{|V(P)|})$. The initial candidate set of $u \in V(P)$ is $\Phi(u) = \{v \in V(G) \mid \text{label}(v) = \text{label}(u)\}$.

A simple idea of optimization is to reduce the size of $\Phi(u)$. Two optimization methods are proposed in the work of GraphQL (He and Singh, 2008). We will illustrate these optimizations and our improvement in the following sub-sections.

2.1 Method 1: Local Reduction of Candidate Sets

Firstly we consider how to reduce $\Phi(u)$ locally, i.e. without using any other information of $\Phi(v)$, $v \neq u$. Despite the existence of labels, it is obvious that we can eliminate the vertices in $\Phi(u)$ whose set of labels of neighbors does not contain the set of labels of neighbors of u as a subset. Using $NS(u)$ to denote the neighbors of vertex u, this optimization can be formalized as $\Phi(u) = \{v \in G \mid \text{label}(v) = \text{label}(u) \wedge \text{label}(NS(u)) \subset \text{label}(NS(v))\}$, where $\text{label}(S) = \{\text{label}(s) \mid s \in S\}$.

Furthermore, we can also consider the neighbors which are at most two units of distances away from u (denoted as $NS^{(2)}(u)$), and three or more. Considering the neighbors at a farther distance may improve the ability of optimization but may result in greater time consumption.

To implement it, we need to check if a set of size m is a subset of another set of size n, which can be done by dynamic programming in $O(mn)$ time.

2.2 Method 2: Joint Reduction of Candidate Sets

Then we consider how to reduce the size of $\Phi(u)$ taking advantage of other $\Phi(u')$. Consider $v \in \Phi(u)$ after the optimization above; if there exists $u' \in NS(u)$ such that there is no $v' \in NS(v)$ with $v' \in \Phi(u')$, then we can conclude that u can never be mapped to v. Thus we can use an algorithm for maximum bipartite graph matching to see if $\forall u \in NS(u)$ there $\exists v' \in NS(v)$ satisfying $v' \in \Phi(u')$. We can see that if we iteratively do this l times, the subtree of u with height less than l is contained by the subtree (vertices may be reused) of v. This method was first proposed by (He and Singh, 2006).

2.3 A Weaker but Faster Improvement of Method 2

The main problem of method 2 is that computing maximum bipartite graph matching is relatively slow. We can speed up the process of Method 2 by using a faster approximation algorithm to reject bipartite graphs which has no semi-perfect matching. To guarantee the correctness of our algorithm, we require the algorithm for semi-perfect bipartite graph to reject if there is definitely no semi-perfect matching. But it can also accept bipartite graphs which have no semi-perfect matching. In

other words, we want a sufficient condition that there is no semi-perfect matching for a bipartite graph.

First, it is easy to see that if for u_0 and $v_0 \in \Phi(u_0)$, $|\bigcup_{u \in NS(u_0)} \Phi(u) \cap NS(v_0)| < |NS(u_0)|$, then there is definitely no semi-perfect match between neighbors of u_0 and neighbors of v_0. We can generalize this condition by considering a subset of $NS(u_0)$. If for $|S \subset NS(u_0)|$, the condition $|\bigcup_{u \in S} \Phi(u) \cap NS(v_0)| < |S|$ holds, then there is also no semi-perfect match between neighbors of u_0 and neighbors of v_0.

Therefore, we can enumerate every $u_i \in \Phi(u_0)$ and maintain the set $C_k = \bigcup_{i=1}^{k} \Phi(u_i) \cap NS(v_0)$. If there is some k such that $|C_k| < k$, we may assert that there is no semi-perfect matching. Note that the order of enumeration does affect the results. To achieve a better ability of pruning, we sort $u \in \Phi(u_0)$ in ascending order of $|\Phi(u)|$.

Theoretically, we can analyze the improvement with asymptotic complexity. The bipartite graph has $N = |NS(u_0)| + |NS(v_0)|$ vertices and $M = \sum_{u \in NS(u_0)} |\Phi(u)|$; the best asymptotic complexity is $O(\sqrt{n}m)$ by the Hopcroft-Karp algorithm. Using the approximation algorithm above, the asymptotic complexity is $O(m \log(m+n))$. What's more, the constant of the approximation algorithm is smaller. Though the pruning ability of the approximation algorithm is weaker, we can see that the improvement on running time is more significant in the experiment section.

3 Finding Optimal Searching Orders

As we have mentioned before, most of the existing algorithms are using a DFS framework. For the query graph and data graph, the searching order of nodes is arbitrary but the searching process may have different performances with different searching orders. Hence, a vital improvement of naive DFS is to find a good searching order to reduce the breadth of the searching tree. However, finding such optimal order is intractable.

3.1 QI-Sequence

In QuickSI (Shang et al., 2008), a heuristic method is used to find an approximately optimal searching order. We assign a weight to each of the edges and vertices according to the number of its appearances in the graph database. We then compute a minimum spanning tree using these weights by Prim's algorithm. For the edges with the same minimum weight, we choose the edge with an induced subgraph after adding this edge of less size. If the sizes are still equal, we choose the edge with less summation of the degrees of the nodes on that edge. If they are still equal, then we choose an edge at random. Thus, we can let the searching order be the order that the nodes added into the minimum spanning tree, which is called QI-Sequence. Then we can perform DFS on the data graph following the order of QI-Sequence.

3.2 Modifying the Order by the Size of the Candidates Set

However, the QuickSI algorithm can't handle some graphs that are large and dense. Fortunately, we found that the candidate set described in Section 2 is relatively small, especially in some graphs with high density, where the candidate sets are mostly empty or have only one node on the data graph. Inspired by this, we modified the process of computing QI-Sequence, choosing edges with nodes that have smaller size of candidate set. Although this modification indeed has significant efficiency improvement on large graphs, we found that this improvement is likely to mainly come from the reduction of candidate sets, but not from improving the searching order, since for a few graphs that the candidate sets are not so small as of size zero or one, the algorithm still can't run with high efficiency. We propose that perhaps this is because that the QI-Sequence must be a valid DFS order of the query graph, and thus a node with small candidate set may be connected with a node with large candidate set. Therefore, when the size of candidate set matters, the modified QuickSI algorithm can't beat GraphQL, which simply searches in the increasing order of the size of candidate sets.

4 Reducing the Duplicate Structure of Data Graphs

All of the algorithms described above focus on the query graph and the searching process, but not the data graph. For data graphs with high density, some local structures may occur multiple times in different positions of the graph. Actually, we can make use of these duplicate structures to reduce some redundant computation.

4.1 Exploiting the Similar Structure to Boost Searching

In some dense graphs, there may exist some nodes that has the same neighbors. Intuitively, if a node in the data graph fails to be mapped by certain node in the query graph, then mapping will also fail for the nodes with the same neighbors.

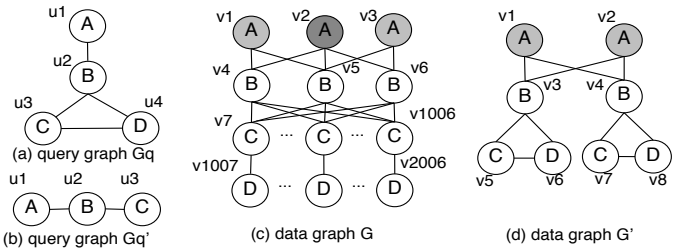

Figure 1: Example Query Graphs and Data Graphs

For example, consider the graph G' in Figure 1, observe that the neighbors of the nodes v_1 and v_2 are the same, so if we fail on v_1 then we don't have to search on v_2. Moreover, not only the equality of the neighbors helps; we can also exploit the containment relations of neighbors. For example, in graph G, the set of neighbors of both v_1 and v_3 are subset of that of v_2, so if we failed on v_2, then we don't have to consider v_1 and v_3. Motivated by this, the algorithm BoostIso (Ren and Wang, 2015) makes use of these redundant structures to compress the data graph, improving the efficiency in searching.

More specifically, the algorithm verifies four types of relations between the nodes in the data graph, namely *Syntactic Containment*, *Syntactic Equivalence*, *Query-Dependent Containment*, and *Query-Dependent Equivalence*. Given two nodes v_i, v_j in a data graph, we say v_i syntactically contains v_j (denoted by $v_i \succeq v_j$) if $\text{label}(v_i) = \text{label}(v_j)$ and $\text{Adj}(v_j) \backslash \{v_i\} \subseteq \text{Adj}(v_i) \backslash \{v_j\}$. We can see that this relation is transitive. We can then define Syntactic Equivalence of v_i and v_j (denoted by $v_i \simeq v_j$) if $v_i \succeq v_j$ and $v_j \succeq v_i$. Thus, if $v_i \succeq v_j$, then replacing v_j with v_i in any mapping will result a new mapping, if v_i is unused. And if $v_i \simeq v_j$, then we can replace in both directions, and we can swap the image of the nodes in the query graph that are mapped to v_i and v_j. These two relations are independent on the query graph, however, we can also use the information in the query graph to further explore the similarities. And Query-Dependent Containment and Query-Dependent Equivalence, are similarly defined but only consider the neighbors whose labels appeared as labels of us neighbors in the query graph, denoted by $\succeq_{(G_q,u)}$ and $\simeq_{(G_q,u)}$, where u is a node in the query graph G_q.

Also, it's easy to verify some properties of these relations (e.g., $\simeq$ is a equivalence relation), and we can use these properties to compress the data graph, i.e., to construct a *adapted hypergraph*. Given a data graph $G = (V, E, \text{label})$, the corresponding hypergraph is $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \text{label}_{sh})$, where $V_{sh} = \{\overline{v} : v \in V\}$, E_{se} is a set of undirected edges $(\overline{v_i}, \overline{v_j})$ that $(v_i, v_j) \in E$, E_{sc} is the smallest set of directed edges such that a path from h to h exists iff $h \succeq h$, and $\text{label}_{sh}(\overline{v}) = \text{label}(v)$. Then we can search on the compressed hypergraph to reduce redundant computations.

Inspired by BoostIso, we can compute the equivalence relations between nodes in data graphs and use these relations to reduce the candidate sets during the searching process. That is, if we find that node v in the query graph fails to be mapped to node u in the data graph, then we can remove nodes that are equivalent to u from the candidate set of v. For implementation, we represent the equivalence relations by a disjoint-set data structure. Then the time complexity for computing equivalence is $O(n^2 d \log d)$, where d is the maximum degree of the nodes (the logarithm term comes from checking

whether the sets of neighbors is equal). However, the running time on n can be much lower than n^2, by the transitivity of equivalence relation. Unfortunately, we found that the equivalence relations are very isolated on the datasets we tested, and thus this modification won't improve too much.

4.2 The Weisfeiler-Lehman Isomorphism Test and Graph Neural Network

In fact, if we consider the neighbors of u as the nodes that the distance to u is less or equal to k, instead of its adjacent nodes, we are actually checking whether two vertices are equivalent in the view of graph or not, i.e. whether two graphs are isomorphic or not. This idea is quite similar to the Weisfeiler-Lehman isomorphism test. Indeed, it is similar to a reverse version of Weisfeiler-Lehman isomorphism test.

Another way to find equivalent vertices is to use the Weisfeiler-Lehman isomorphism test (Weisfeiler and Lehman, 1968). The main idea of it is to produce a canonical form for each graph. If the canonical forms of two graphs are not equivalent, then the graphs are not isomorphic. However, it is possible that two non-isomorphic graphs to share a canonical form or the calculation for a canonical form never converges. Despite these, it is a good probabilistic algorithm for graph isomorphism testing. Since it calculates a feature for each vertex, we can just use the features to check if two vertices are equivalent.

However, calculating the features of Weisfeiler-Lehman isomorphism test requires a lot of time. One method to optimise is to use the Graph Neural Network. Xu et al. (2019) proposed that the most powerful Graph Neural Network is as powerful as the Weisfeiler-Lehman isomorphism test. And they proposed a Graph Isomorphism Network which can efficiently calculate a feature for vertices which has a good ability in distinguishing different vertices.

But since it is a probabilistic method, the results may be wrong if we use the Graph Isomorphism Network to find equivalent vertices. That's what we don't want. And due to our limit time, we haven't test the speed improvement with the Graph Isomorphism Network.

5 Experiments and Discussions

5.1 Datasets

We use two different datasets for the experiments.

The first is the AIDS Antiviral Screen Data from wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data. It contains tens of thousands of compounds which have been checked for evidence of anti-HIV activity. Since they are all chemical formulas, the graphs in this dataset is relatively small and sparse. On average, each graph has 25.4 vertices and 27.3 edges. The size of the largest graph is of less than 300 vertices. What's more, most of the labels are C, O, H, N, S.

The other dataset is the DBLP collaboration network and ground-truth communities dataset from the Stanford Large Network Dataset Collection. We choose 100 communities whose size are about 500 vertices as the database. The maximum size of the graphs is 592 and the minimum size of the graphs is 433. Each graph has 1368 edges on average. Since there are no labels initially, we random generate labels for vertices from a Guassian distribution $N(15, 5)$ truncated in $[0, 30]$.

5.2 Generating Query Sets

There are six query sets $Q_4, Q_8, Q_{12}, Q_{16}, Q_{20}, Q_{24}$ for each of the two datasets. Q_i contains 100 graphs of size i. Each graph in Q_i is a random sub-graph of some graph in the datasets. The random sub-graph is generated by depth-first-search from a random vertex and choosing each edge with the probability $p = 0.6$.

5.3 Experiments

For each dataset and each query set of it, we do subgraph isomorphism test between each pair of 100 graphs from the dataset and 100 graphs from the query set, for 10000 times in total. We use the

average time to measure the efficiency of each algorithm. We also show the average reduction of the size of $\Phi(u)$ to show the ability of the first kind of optimization.

The results are shown in the table and the figures below.

Results of the reduction of candidate sets In Table 1, we use opt 1 to denote the method 1, opt 2 to denote the method 2 and opt 2x to denote the improved version of method 2.

	Initial Average Size	Opt 1	Opt 2	Opt 2x
Q_4	28.145100	19.721800	0.940200	1.050500
Q_8	28.003000	17.418600	0.115500	0.136500
Q_{12}	28.430400	17.546600	0.055300	0.063800
Q_{16}	28.204400	17.154400	0.038500	0.040800
Q_{20}	27.788900	16.453000	0.026200	0.027900
Q_{24}	27.470700	15.550400	0.018400	0.019300

Table 1: Reduction of candidate sets.

Clearly, the average size of $\Phi(u)$ of opt 2x is just a slightly larger than opt 2, but we obtain a significant reduction of average size by all the three optimizations, indicating that our optimizations indeed take effects.

Results of Running Time In Figure 2, we compare the effects of the 3 optimizations on GraphQL. Note that optimizations 2 and 2x are not suitable in small graphs (left, AIDS dataset), but vital in bigger graphs (right, DBLP dataset). Since the time saved by optimizations may not compensate the time comsumed by themselves. We can also conclude from the figures that optimization 2x's effect is similar to optimization 2's.

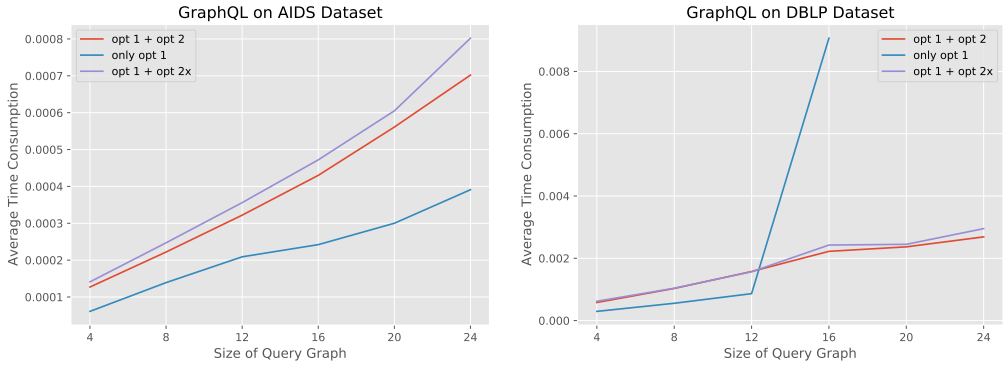

Figure 2: Results for GraphQL.

In Figure 3, we compare the effects of the 2 optimizations on QuickSI. We shall first note that in the following test, we ignore one query instance that makes the testing too slow. Clearly encountering such graph instances is highly unlikely, supporting our ignorance of this instance.

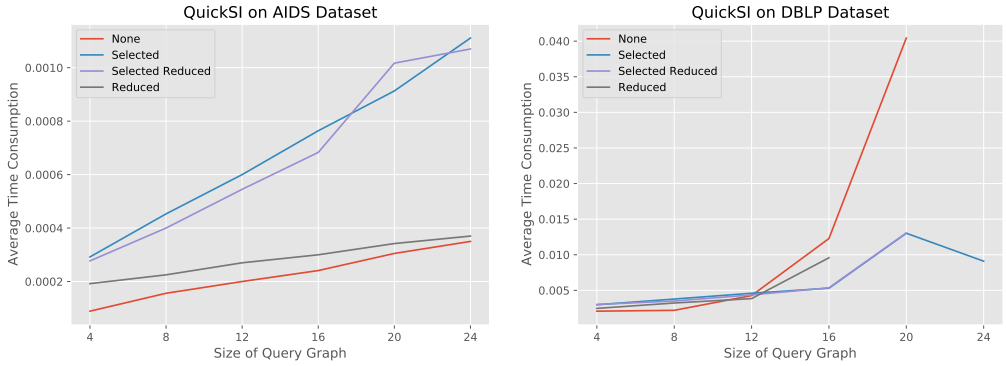

Figure 3: Results for QuickSI.

Similar to the results of GraphQL, we see that complex optimizations greatly improve the performances on large graphs while they are unnecessary in small graphs.

Finally, we compare the performances of different algorithms and optimizations. Different algorithms show similar efficiency on small graphs, while GraphQL shows apparent superiority in large graphs. As what the experiments suggest, GraphQL with opt 1 and 2 performs best.

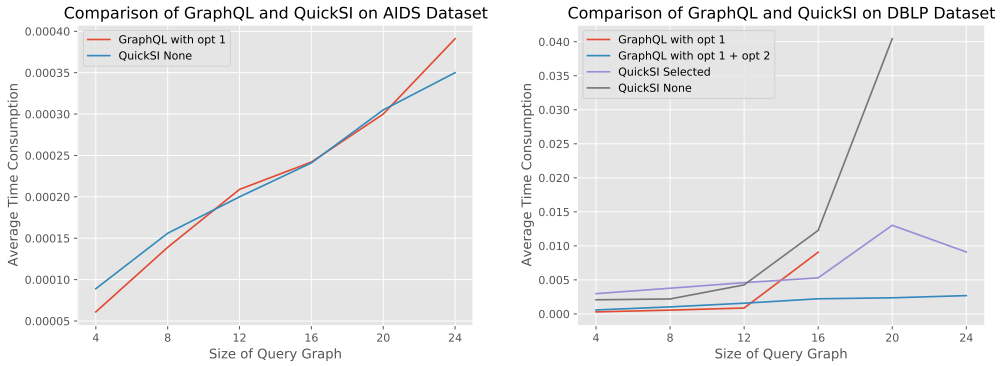

Figure 4: Comparisons of the two methods.

6 Conclusions

In this project, we present, analyse and test various (sub)graph isomorphism algorithms and our own optimizations. We successfully find out some useful optimization routines for this problem. We also conclude that different approaches should be adopted considering the sizes of the graphs. Our group also gains a lot of experiences on coding (about 4500 slocs of code written) and algorithm analysis.

There is also some meaningful future work to do. We may try Graph Isomorphism Network that has not been implemented due to limited time. Moreover, we could combine the algorithms and optimizations mentioned above in an adaptive manner to further improve the performances.

References

László Babai. Graph Isomorphism in Quasipolynomial Time. *arXiv e-prints*, art. arXiv:1512.03547, December 2015.

L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10): 1367–1372, 2004.

Huahai He and Ambuj K Singh. Closure-tree: An index structure for graph queries. pages 38–38, 2006.

Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. pages 405–418, 2008.

Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133144, December 2012. ISSN 2150-8097. doi: 10.14778/2535568.2448946.

Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617628, January 2015. ISSN 2150-8097. doi: 10.14778/2735479.2735493.

Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364375, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453899.

J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):3142, January 1976. ISSN 0004-5411. doi: 10.1145/321921.321925.

Boris Weisfeiler and Andrei A Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks. 2019.

Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(12):340351, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920887.