Runtime Performance of Systems and Algorithms for Subgraph Isomorphism

Independent Study

Fall 2018

Suhas Keshavamurthy

University of Massachusetts Amherst, MA

Abstract

This project implements the Generic Join algorithm, specifically in the context of subgraph isomorphism in the Arabesque system [1], and compares its runtime performance with Qfrag [2]. Additionally, the output of the CFLMatch application that is elaborated in [3] is evaluated. It is observed that the performance of the implementations for Qfrag and Generic Join yield comparable runtimes across the various datagraphs and queries, while CFL-Match performs much better atleast for smaller datagraphs.

1. Introduction

While the subgraph-isomorphism problem is considered to be NP-complete in complexity, a lot of effort has been invested in developing various computational paradigms including the use of heuristics in order to solve the problem as efficiently as possible.

$Turbo_{ISO}$ [1] is considered to be one of the state-of-the-art solutions to this problem. A version of this algorithm is implemented in the Qfrag system in the Arabesque framework. Postponing Cartesian Products [3], and data-parallel dataflow computation of worst-case optimal join algorithms [4] are some of the recent approaches at solving this problem.

The following subsections contain a brief background of various topics that are relevant to the implementation and benchmarking of the algorithms.

1.1. Subgraph Matching

Given a query **q** and a large data graph **G**, the problem is to extract all subgraph isomorphic embeddings of q in G. An instance of this problem is depicted in Fig 1. Here we are attempting to analyze if the Target Graph G_t contains a subgraph that is isomorphic to the Pattern Graph G_p.

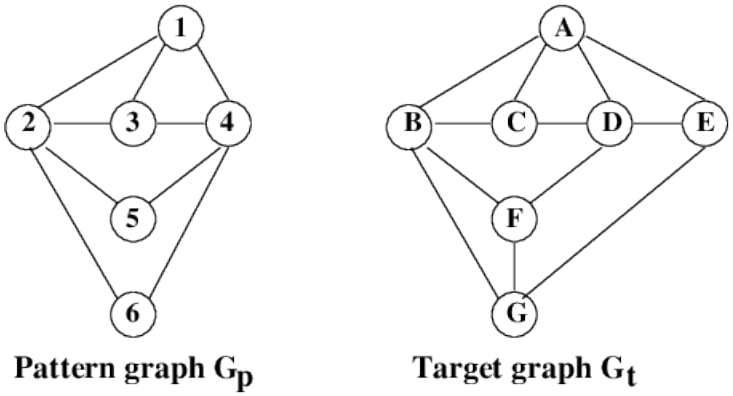


Figure 1: Instance of a Subgraph Isomorphism Problem [5]

1.2. Generic Join

Ngo et.al [6] presented a description of Join algorithms with provably worst case optimality runtime guarantees. While the paper discussed the algorithm in terms of database relations, an equivalent view can be extended to nodes (attributes) of a graph. The treatment and implementation of Generic Join in this project is based on the introductory explanation of the algorithm provided in [4]. This implementation does not contain many of the optimizations including optimal ordering of attributes, addressing skew in input datagraphs etc.

1.3. Arabesque and Qfrag

Arabesque is a distributed framework for implementing graph mining algorithms [1]. Qfrag [2] is a distributed system for graph search built on top of Bulk Synchronous Protocol systems. In this framework, the input graph is distributed to all the worker nodes in order to avoid expensive distributed joins and minimize communication. A unique task fragmentation strategy is implemented to avoid the problem of skew in the graphs while ensuring minimum communication among the nodes. The claim is that for certain type of problems where the input graph can fit in main memory, Qfrag is better able to handle computationally expensive analytical queries.

1.4. CFL-Match

CFL-match is an algorithm described in [3] for efficient and scalable subgraph matching. It addresses two main issues in the state-of-the-art algorithms like $Turbo_{ISO}$.

1. *Redundant Cartesian Products by Dissimilar Vertices* : Taken care of by postponing Cartesian Products in a new framework which processes the query graph into a *Core*, *Forest* and *Leaf* decomposition and helps avoid redundant potential mappings
2. *Exponential size of the Path-based Data Structure in* $Turbo_{ISO}$: A new Compact Path-Index (CPI) data structure with polynomial size is proposed

1.5. Amazon EMR

Amazon EMR is a PaaS offering in Amazon AWS services. As stated on the website 'Amazon EMR provides a managed Hadoop framework that makes it easy, fast, and cost-effective to process vast amounts of data across dynamically scalable Amazon EC2 instances.' For the purposes of the project, Amazon EMR aided in setting up a 'Hadoop + Spark' environment on which the compiled program Jar could be executed for a scalable number of instances.

2. Project Goals

The independent study had the following main objectives

1. Implementation of the Generic Join algorithm for subgraph isomorphism search in the Arabesque System
2. Measure and Compare the performance of Qfrag, Generic Join implementation and CFL-match
3. Explore possible extensions to the Qfrag system

3. Implementation

3.1. Data

The experiments are performed on 4 datagraphs. The datagraph characteristics are presented in Table 3.1

DataGraph	No. of Vertices	No. of Edges	No. of Labels
Citeseer	3312	9072	6
Mico	100000	2160312	29
Patent	2745761	27930818	37
Youtube	4589876	87937596	108

The query set consists of 8 queries for each datagraph. Each of the 8 queries is similar for the 4 datagraphs. The queries look for a similar pattern of edges between the nodes but for nodes with different labels in the different datagraphs.

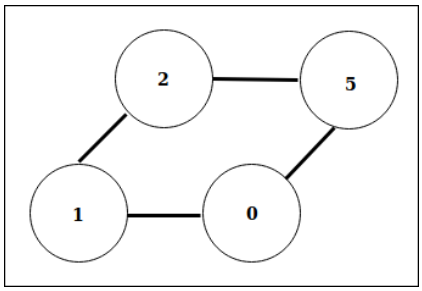


Figure 2: Example Query Graph. Query-6 for Citeseer

Code is also implemented for the case of unlabeled graphs for the same set of datagraphs. All the labels in the input datagraph are set to the same value, and the query graphs are modified accordingly.

3.2. Generic Join Algorithm

The code is implemented within the Arabesque framework. The driver code follows the template of the Qfrag implementation. The pseudocode of the algorithm is shown in 3.

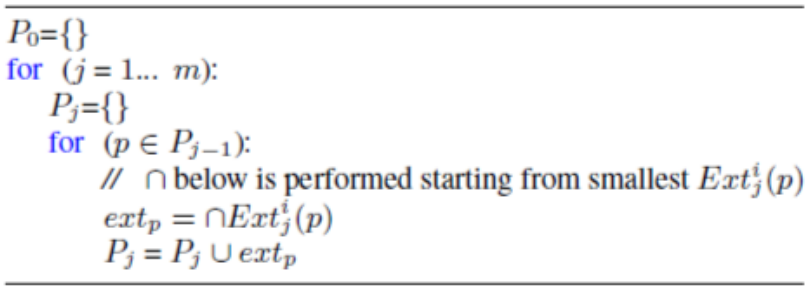


Figure 3: Pseudocode of the Generic Join Algorithm. Taken from [4]

Since each of the workers receives the entire datagraph, the initial list of candidate vertices for each partition is determined by splitting the entire set of vertices into the number of partitions in the configuration (*Num Workers* x *Num Compute Threads per Worker*). A map of labels and the set of vertices in the datagraph with that label is also maintained.

At each iteration, a list of prefix extensions is maintained in a Queue. For each node in the query graph, we iterate through all the existing prefix extensions in order to evaluate if it can be extended with a node with the new label. A priority queue (based on the size of the set of vertices) is used to maintain set of vertices that correspond to the neighbours of the particular node in the query graph under consideration. The intersection of these set of vertices provide the new list of potential addition to the existing prefix extensions. Since the intersection of the Sets are carried out in increasing order of the size of the Sets, this leads to optimal computation of this step. At any point of time when we encounter a Set with zero elements, the evaluation of the particular iteration can be abandoned.

The result of the algorithm is the prefix extensions that are remaining after iterating through all the nodes in the query graph.

3.3. CFL-Match

The CFL-Match binary is available from the authors of [3]. Python scripts are written in order to convert the input datagraphs of 3.1 and the queries into the format expected by the CFL-Match application. The application is then executed and the output parsed in order to obtain the runtime query performance. The CFLMatch-Enumeration binary is used to retrieve the embeddings that were matched by the application. The results are de-duplicated

and the result sizes obtained are matched with the ground-truth values, as well as the output of the Qfrag/Generic-Join implementation.

3.4. Scripts

Various python scripts are written in order to test the code, benchmark the algorithms and retrieve the results. This includes comparing and running Qfrag and Generic Join Implementation in various configurations, sorting the outputs in order, verifying the results of the two approaches, convenience scripts written to insert data into HDFS, retrieve data and code from AWS S3 into the Master node and run the experiments on the EMR cluster.

3.5. Measurements

The experiments to measure the runtime performance was carried out on an Amazon EMR cluster. All the master and worker instances were of similar configuration. The configuration of the nodes were m4.xlarge instances with 8 vCore, 16 GiB memory, EBS only storage, EBS Storage:32 GiB. The cluster had the following frameworks installed. Spark 2.3.2 on Hadoop 2.8.5, YARN with Ganglia 3.7.2 and Zeppelin 0.8.0.

4. Analysis

While the implementation supported both labelled and unlabelled graphs, all the experiments were performed for the case of labelled graphs only. For each of the configuration/variation, multiple runs (5-10) are carried out. The shaded regions in the plots that follow denote the variations in runtime for that particular configuration/variation. The line/marker denotes the average runtime recorded.

The total computation time is calculated as the difference between the minimum (start time) and maximum (end time) of all the time values reported by each of the workers.

4.1. Verification of Implementation

The ground-truth values for the labelled and unlabelled graphs are available. The output of the Generic Join implementation is verified against this. Further, it is verified that the Generic Join output is equivalent to the Qfrag output. Once this has been carried out, during the benchmarking period, only the result size returned by the Generic Join implementation is verified with hard-coded ground-truth values in the script.

4.2. Single Thread

The plots 4a, 4b, 4c and 4d present the runtime performance of both Qfrag and the Generic Join implementation running on a single thread. While the Generic Join implementation performs much better (5 times faster) for the *Citeseer* datagraph, Qfrag performs consistently better for the rest of the datagraphs across all queries (atleast 1.5-3 times faster). The Generic Join implementation also shows greater skew in runtime performance for different queries against these datagraphs.

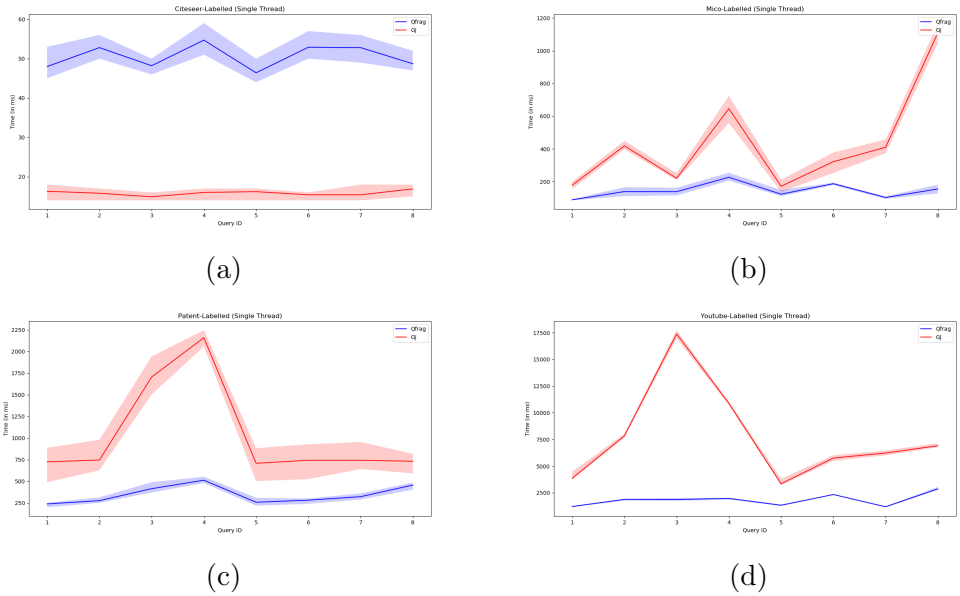


Figure 4: Runtime performance - Single Thread (a) **Citeseer** (b) **Mico** (c) **Patent** (d) **Youtube**

4.3. Single Worker, Multiple Threads

The plots 5a, 5b, 5c and 5d present the runtime performance of both Qfrag and the Generic Join implementation running on multiple threads. The experiments are run for 2, 4 and 8 number of compute threads per worker respectively. It can be observed that the performance is similar to that obtained from a single thread. It is in fact observed that the performance deteriorates slightly as the number of threads increase. This can be attributed to the fact that the increase in threads does not yield sufficient parallelism benefits.

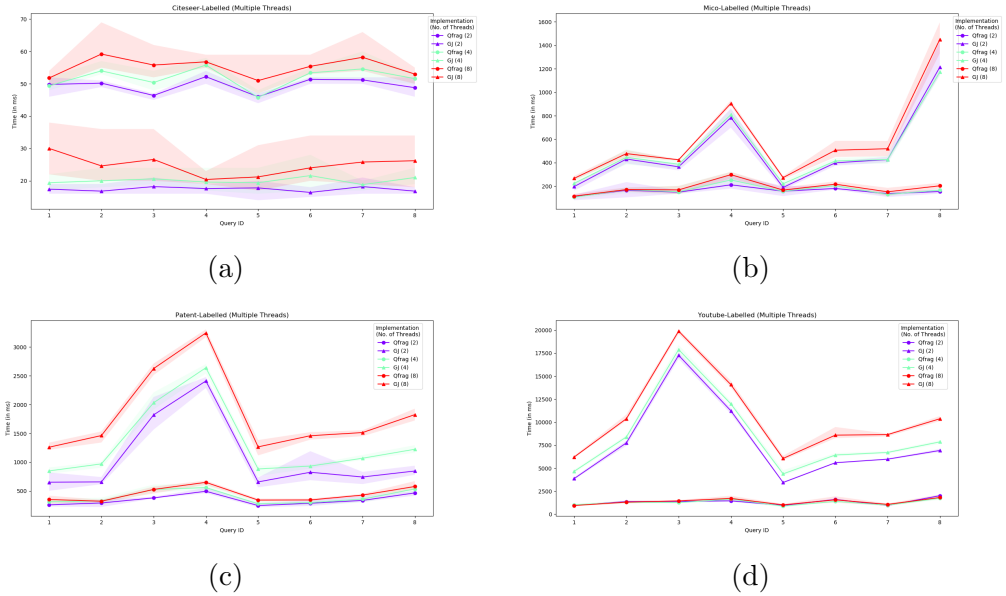
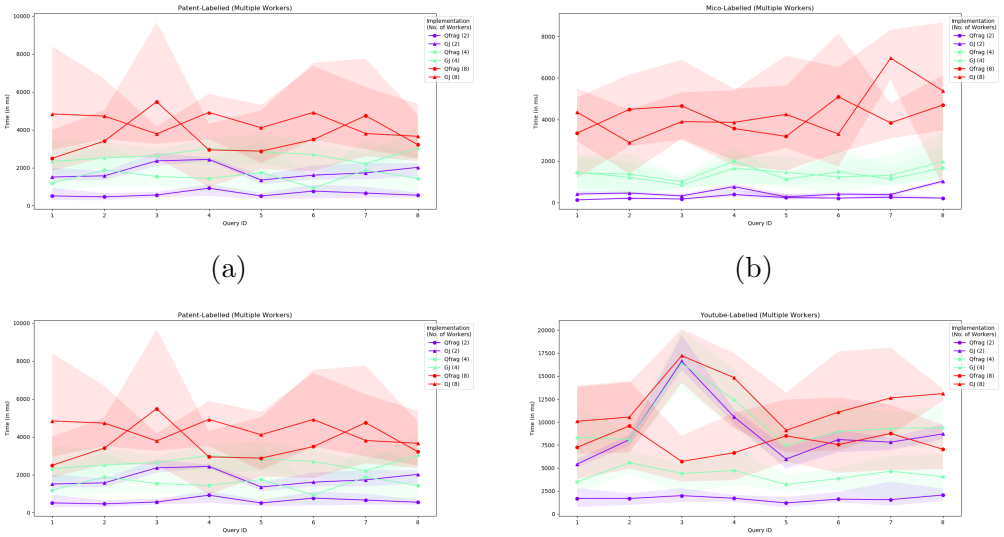


Figure 5: Runtime performance - Multiple Threads (a) **Citeseer** (b) **Mico** (c) **Patent** (d) **Youtube**

4.4. Multiple Workers, Multiple Threads

The plots 6a, 6b, 6c and 6d present the runtime performance of both Qfrag and the Generic Join implementation running on multiple workers. The experiments are run for 2, 4 and 8 number of workers. Each of the workers are running with 8 compute threads. It is observed that the performance is considerably worse as the number of workers is increased. This can be attributed to the way the computation runtime is calculated. The runtime is the difference between the max-time and the min-time at which the computation ended and started respectively amongst all the workers. Since there is variations at which the tasks are executed at each of the workers, this variation is comparable to the actual computation time and in cases of smaller graphs like *Citeseer*, even tends to dominate it.



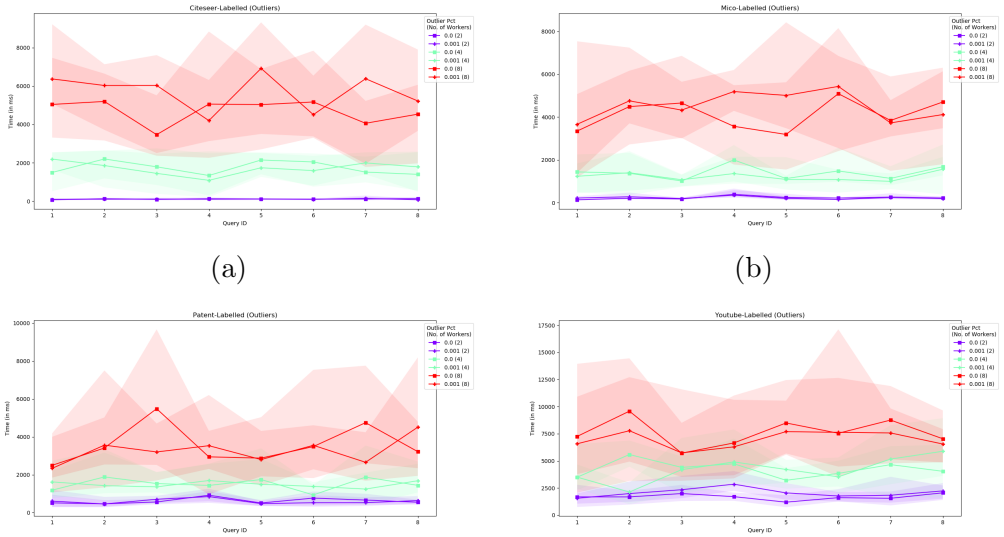
(a) (b)

(c) (d)

Figure 6: Runtime performance - Multiple Workers (a) **Citeseer** (b) **Mico** (c) **Patent** (d) **Youtube**

4.5. Load Balancing

The *outlier_pct* option is varied between 0.0 and 0.001 in order to control the redistribution of load in the embedding enumeration stage of Qfrag. The results for varying this parameter for different number of workers are then plotted as shown in 7a, 7b, 7c and 7d. There is no distinct effect of load balancing and the average runtime values for both the scenarios are in the same range.



(a) (b)

(c) (d)

Figure 7: Runtime performance of Qfrag - Load Balancing (a) **Citeseer** (b) **Mico** (c) **Patent** (d) **Youtube**

4.6. CFL-Match

The CFLMatch application is run locally on the development machine. While it was not possible to verify the output of the application (the output did not match with the ground-truth for 1 query in *Citeseer* and all queries for *Mico*), the average runtime for each query is still noted. For both the datasets (*Citeseer* and *Mico*), the CFL-Match runtime is atleast 2 orders of magnitude faster than the Qfrag system. Figure 8a and 8b contains outputs of the CFLMatch application for one of the queries for each of the datagraphs.

```
****************************************************************
Data File :./graphs/citeseer.unsafe.graph_cfl
Query file:./queries_cfl/Q1-citeseer_cfl
****************************************************************
Reading data graph ...OK!
Building index...OK!
Average Search Time Per Query => 0.0070 ms!
Average Total Time Per Query => 0.0580 ms!
Total Number of Found Matches=> 76
```

(a)

```
****************************************************************
Data File :./graphs/mico.unsafe.graph_cfl
Query file:./queries_cfl/Q1-mico_cfl
****************************************************************
Reading data graph ...OK!
Building index...OK!
Average Search Time Per Query => 1.2140 ms!
Average Total Time Per Query => 2.0510 ms!
Total Number of Found Matches=> 207,920
```

(b)

Figure 8: Runtime performance of CFL-Match (a) **Citeseer** (b) **Mico**

5. Conclusions

In this study, the Generic Join algorithm for the task of subgraph isomorphism is implemented. The runtime performance of this algorithm is then compared with the Qfrag distributed system. It is observed that Qfrag is consistently faster than the Generic Join implementation for the 3 datagraphs that were tested (*Mico*, *Patent* and *Youtube*). Only for the case of *Citeseer*, it is observed that the Generic Join performance is much better than Qfrag. The Generic Join implementation also demonstrates higher skewed performance across different queries, whereas the performance of Qfrag is more consistent.

There exists potential improvements to the Generic Join implementation to address skew as well as improve runtime performance in general. Similarly potential optimizations for Qfrag can be implemented in order to compare the runtime performance of the different approaches more accurately.

The performance of CFLMatch is atleast two orders of magnitude better than the Qfrag system for the *Citeseer* and *Mico* datagraphs. Since the source code of the application is not available, it is difficult to ascertain the cause for the extent of the improved performance. It can be fundamental algorithm differences or even due to implementation differences (C++ vs Java; distributed vs sequential).

6. Future Work

The performance of the Generic Join implementation can be improved by addressing the following issues

1. The hashmap of labels and set of vertices that is created initially adds considerable overhead to the runtime, especially as the size of the datagraph increases. This can be alleviated by either using a more optimal data structure rather than the generic Java implementation or by performing this action at the time of building the graph. This helps avoid the overhead that is created for each worker in the process.
2. Presently, the initial set of attributes for each partition are chosen by dividing all the vertices into equal chunks in lexicographical order. There exists a stronger preference for global attribute order.
3. The issue of skew is not addressed in the implementation. Hence, wide variations in performance are observed for different types of queries.

7. References

[1] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, A. Aboulnaga, Arabesque: A system for distributed graph mining, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 425–440.

[2] M. Serafini, G. De Francisci Morales, G. Siganos, Qfrag: Distributed graph search via subgraph isomorphism, in: Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, ACM, New York, NY, USA, 2017, pp. 214–228.

[3] F. Bi, L. Chang, X. Lin, L. Qin, W. Zhang, Efficient subgraph matching by postponing cartesian products, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, ACM, New York, NY, USA, 2016, pp. 1199–1214.

[4] K. Ammar, F. McSherry, S. Salihoglu, M. Joglekar, Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows, Proc. VLDB Endow. 11 (2018) 691–704.

[5] S. Zampelli, Y. Deville, C. Solnon, Solving subgraph isomorphism problems with constraint programming, Constraints 15 (2010) 327–353.

[6] H. Q. Ngo, C. Ré, A. Rudra, Skew strikes back: New developments in the theory of join algorithms, SIGMOD Rec. 42 (2014) 5–16.