l2Match: Optimization Techniques on Subgraph Matching Algorithm using Label Pair, Neighboring Label Index, and Jump-Redo method

Chi Qin Cheng
School of Information Technology
Monash University
Selangor, Malaysia
0000-0003-0718-3981

Kok Sheik Wong
School of Information Technology
Monash University
Selangor, Malaysia
0000-0002-4893-2291

Lay Ki Soon
School of Information Technology
Monash University
Selangor, Malaysia
0000-0002-8072-242X

***Abstract*—Graph database is designed to store bidirectional relationships between objects and facilitate the traversal process to extract a subgraph. However, the subgraph matching process is an NP-Complete problem. Existing solutions to this problem usually employ a filter-and-verification framework and a divide-and-conquer method. The filter-and-verification framework minimizes the number of inputs to the verification stage by filtering and pruning invalid candidates as much as possible. Meanwhile, subgraph matching is performed on the substructure decomposed from the larger graph to yield partial embedding. Subsequently, the recursive traversal or set intersection technique combines the partial embedding into a complete subgraph. In this paper, we first present a comprehensive literature review of the state-of-the-art solutions. l2Match, a subgraph isomorphism algorithm for small queries utilizing a Label-Pair Index and filtering method, is then proposed and presented as a proof of concept. Empirical experimentation shows that l2Match outperforms related state-of-the-art solutions, and the proposed methods optimize the existing algorithms.**

***Index Terms*—subgraph isomorphism, subgraph matching, information retrieval, communication and information theories**

I. Introduction

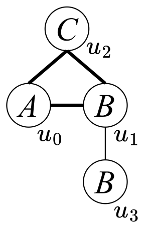
Graph algorithm is an intensively researched subject. Its significance has grown in recent years. For instance, the shortest path algorithm is frequently used in the logistics and transportation sectors to cut waste and boost productivity [1]. Additionally, Natural Language Processing groups similar objects using the graph clustering technique to facilitate information retrieval [2]. These are just a few, not all, instances of how the graph algorithm is commonly used. In a graph, edges depict the connections between vertices or objects. Protein-to-protein and social interactions are two examples of the kinds of information that can be represented with graphs. These data cannot be stored in a relational database due to the high level of randomness and uncertainty, as the schema and structure of the data must be identified and defined beforehand.

The Subgraph Matching problem, also known as Subgraph Isomorphism (**SI**), aims to match isomorphic subgraphs (embeddings) in a larger graph for a specific query graph. This procedure is somewhat comparable to matching the occurrences of a string pattern p in a text file X. String matching is however a simpler problem than SI, where SI is NP-Complete and more challenging [3].

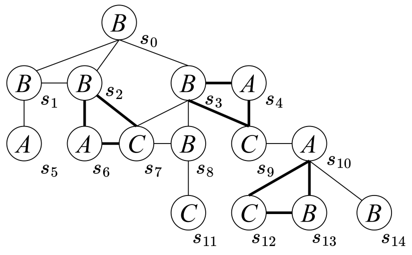
Recent researches [4]–[8] approach the SI problem with the filter-and-verification framework, which essentially trims the solution space to decrease the time consumption in the verification stage [7]. In particular, CFL-Match[4] and CECI[5] apply Forward Candidate Generation and Backward Candidate Pruning (**FCGBCP**) to prune invalid candidates by leveraging the connection properties of a query vertex. Additionally, CECI proposes a Compact Embedding Cluster Index (**CECI**) auxiliary data structure to speed up index lookup during the filtering step. When moving forward through the query graph, Forward Candidate Generation (**FCG**) generates and filters the candidates. Backward Candidate Pruning (**BCP**), in contrast, filters and removes invalid candidates in the other direction. Nevertheless, the removal of an invalid candidate for a query vertex is not propagated to its neighboring candidates that are mapped to other query vertices. Therefore, the removal of invalid candidates is incomplete, necessitating another expensive refinement step while traversing backwards as observed in the CECI algorithm. Furthermore, various studies [4]–[6] use the Neighboring Label Frequency (NLF) filtering method to compare the frequency of occurrence of each unique label among a vertex's neighbors between a query vertex and a candidate. It prunes a candidate if its NLF is lesser than the NLF of a query vertex for any neighboring unique label. Although the NLF is already known, FCGBCP has to scrutinize every edge of a candidate to find its neighbors with a particular label. The above shortcomings motivate the proposal of l2Match algorithm in this research to optimise the filtering step of the CECI algorithm and reduce the total time consumption of query by: (1) traversing query graph efficiently during filtering step, and (2) avoiding unnecessary edge scanning in NLF filtering, and (3) reducing exploration of the redundant search branches in the enumeration step.

Our research makes following contributions:

1) We develop the Label Pair Filtering (LPF) method that generates the set of candidate vertices for a given query



(a) Query graph Q.



(b) Data graph D.

Figure 1: Query Graph, Data Graph, and Embeddings.

vertex. LPF utilizes LPI to obtain a subset of data edges with a specific label pair. As a result, LPF skips the filtering process on the data vertices that are not connected to any neighbors of a specific label.

2) We propose a Neighboring Label Index that extends the NLF index to omit the label and degree filtering on data edges with a mismatching label pair.
3) We simplify FCGBCP filtering and traversal by combining the Backward Candidate Pruning (BCP) with the backward refinement step into the BCPRefine method to remove additional traversal on the query graph and avoid expensive invalid candidate removal operations and NLF filtering in the BCP step.
4) We present the Jump-Redo (JR) technique (RO4) to reduce the number of search nodes explored and the time taken in the enumeration step.
5) We demonstrate the effectiveness of the I2Match algorithm that combines the LPI, LPF, Neighboring Label Index, CECI auxiliary indexing data structure, BCPRefine, and JR methods against the state-of-the-art algorithms through empirical evidence.

A. Preliminaries

Given a query graph Q, a data graph D, and a label function L, non-induced SI is defined as an injective function f that maps the query vertices $V(Q)$ to data vertices $V(D)$ complying to certain constraints, as shown in Table V attached to the Appendix. For instance, the set of vertices of an isomorphic subgraph that can be mapped to $\{u_3, u_1, u_0, u_2\}$ in the query graph Q are shown in Figure 1 as $\{\{s_0, s_2, s_6, s_7\}, \{s_1, s_2, s_6, s_7\}, \{s_0, s_3, s_4, s_9\}, \{s_8, s_3, s_4, s_9\}\}$. Each vertex's label is represented by an alphabet within it.

II. Related Work

There are varying methods for approaching the SI problem, such as query graph decomposition, search space reduction, and query parallelization.

A. Indexing Method

SubGlw decomposes the data graph into small candidate graphs using a ranking function based on candidate count and maximum distance between data vertices [9]. Nonetheless, the decomposition of the data graph takes an additional $O(|V'|)$ time. VC [10] implements a bipartite graph index, but the cost (size) of each query vertex is calculated in advance before filtering and pruning procedures. Meanwhile, SMS2 [11] employs lattice-based index and hashing algorithm as an alternative SI indexing approach. QuickSI [7] sorts the ordering of edges in the query graph based on the frequency of the edges presented in the data graph. TurboISO [8] and CFL-Match [4] implement a tree-based index to reduce pruning cost and iteration. Furthermore, the matching is performed at the index level instead of the data level. However, both TurboISO [8] and CFL-Match [4] have the common drawback that the ordering of edges is not considered when traversing the index.

B. Reducing Search Space

In terms of reducing search space, VEQ_M employs dynamic equivalence to avoid enumerating candidates that would not yield any embedding if a candidate that shares similar neighbors rooted at a subtree in Candidate Space (CS) fails to yield any embedding [12]. Here, embedding refers to a complete subgraph that is isomorphic to the query graph Q. Besides, HyGraph utilized branching in a potential isomorphism boundary to select candidates [13]. The branching approach in HyGraph is prone to automorphism, i.e., the permutation of partial embedding. VF2 suggests the employment of ordered relationships and State Space Representation (SSR), which resembles an automaton machine where certain conditions must be fulfilled to proceed to the next state, also known as the feasibility rules. Candidate graphs are collected and refined progressively while transiting through states in SSR. Furthermore, GraphQL [14] proposes three steps to reduce search space, such as neighbourhood subgraph string profile, joint reduction, and search ordering using a cost model. Nonetheless, GraphQL [14] does not support recursive pattern search.

C. Parallel Processing

Parallel processing technique exploits the advantages of processing data simultaneously to increase the efficiency of an algorithm. Glasgow is a constraint programming SI algorithm adopting the parallel processing method [15]. It employs a bit-parallel data structure and threads to explore search spaces in parallel. Similarly, G-Morph algorithm makes use of the advantage of parallel processing on a GPU to solve induced SI for labeled graphs [16]. Last but not least, GR1 is a parallel processing SI algorithm that presents a transition from parallelism to a quantum computing approach to solve the SI problem [17]. It facilitates a producer-consumer pattern that allows interchanges of varying existing pruning strategies to adapt to different use cases, though it has only experimented on the protein-to-protein interaction graph.

III. L2MATCH

A. Overview

Algorithm 1: l2Match.

```
Input  : Query graph $Q$, Data graph $D$
Output: All matched subgraph isomorphic to $Q$
1 $Q \leftarrow$ convertToCSR($Q$) ;                    // Convert $Q$ to CSR
2 $D, \Phi, \pi \leftarrow$ constructLPI($D$) ;        // Construct Label Pair
    Index
3 $C, I, o \leftarrow$ Filter($Q, D, \Phi, \pi$) ;   // Ordering, filtering and
    index construction
4 Enumerate($Q, D, C, I, o, \{\}, 0$) ;       // Recursive enumeration
```

l2Match takes the form of filter-and-verification framework described in [4], [18]. It firstly constructs a Label Pair Index (**LPI**) on the query and data graph in Section III-C. Subsequently, the filtering step prunes the invalid candidates and stores the partial candidate-to-query vertex mappings in an auxiliary indexing data structure, Compact-Embedding Cluster Index (**CECI**). The filtering step comprises of:

1) a Label Pair Filtering (**LPF**) technique defined in Section III-D to filter and map the candidates to each query vertex with label pair constraint.
2) Breadth-First Search Ordering method to sort the traversal order of the query vertices in the enumeration step.
3) Forward Candidate Generation (**FCG**) to generates and filters the candidates for each query vertex. The mapping is stored in an auxiliary indexing data structure.
4) Backward Candidate Pruning and Refinement (**BCPRefine**) defined in Section III-E to prune the invalid candidates and update the auxiliary indexing data structure.

Finally, the enumeration step recursively verifies and extends the partial mapping in the auxiliary indexing data structure into an embedding. l2Match uses Jump-Redo (**JR**) method defined in Section III-F to reduce the exploration of redundant search branches in the enumeration step. The novelty of this research is observed in LPI, LPF, BCPRefine, and JR method. Algorithm 1 shows the highly-abstracted pseudocode of l2Match.

B. Neighboring Label Index

Neighboring vertices of any vertex u may have identical or dissimilar labels. The size of each group with a *distinct* label $\forall l \in L(N(u))$ is defined as Neighboring Label Frequency (NLF) in [4], [10], [19], or formally $|N_l(u)|$. NLF is computed and generated during preprocessing operations on the input graphs. Neighboring Label Offset (NLO) extends the concept of NLF by enabling direct access to a vertex's neighbors with a particular label. FCG and BCPRefine filtering technique employ NLO to skip validation on the neighbors of any candidate with an undesired label. Given a label l,

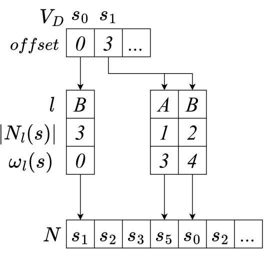


Figure 2: Neighboring Label Index π

the NLO $\omega_l(u)$ of a vertex u stores the offset position to the neighbor vertex set $N_l(u)$ in the sorted adjacent array N of a Compressed Sparse Row (**CSR**). The combination of NLO and NLF of query vertex u with label l is defined as Neighboring Label Index $\pi_l(u) = \big(\omega_l(u), |N_l(u)|\big)$ and computed during the computation of NLF.

C. Label Pair Index

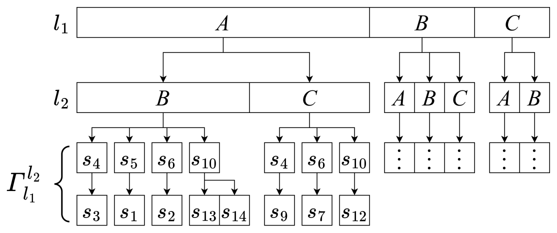


Figure 3: Label Pair Index Φ

Each edge in the context of the data graph D connects two vertices, for example, $e_D(s,t) \in E(D)$. As a result, the labels of the two ends of an edge form a label pair $\big(L(s), L(t)\big)$. Label Pair Index (**LPI**), denoted by Φ, groups data edges $E(D)$ by the label pair of each edge into an index data structure. It allows direct access to data edges with the desired label pair $\big(L(s), L(t)\big) = \big(L(u), L(v)\big)$. This approach reduces the number of iterations down to $\Sigma_{i=0}^{|u|}|\Phi_{L(u)}^{L(v_i)}| \quad \forall v_i \in N(u)$.

D. Label Pair Filtering

LPF employs LPI to yield a minimal candidate set $C(u)$ for any query vertex u. It computes the smallest subset of data edges with the label $L(u)$ and neighboring labels $L\big(N(u)\big)$. The neighboring labels of u that generate the minimum subset of data edges are referred to as l_{min}. Following this, it performs degree and NLF filtering. As a result, LPF skips the degree and NLF filtering on the data vertices that have a different label than u, or are not connected to any neighbors with the label l_{min}. LPF uses the edge filtering technique for greater pruning power compared to conventional methods, which use the vertex filtering technique to obtain a preliminary set of data vertices before degree and NLF filtering. In other words, the smaller the preliminary set of data vertices to process, the less filtration needs to be performed.

E. Backward Candidate Pruning and Refinement

Lemma III.1. *Removing just an invalid candidate s from parent index $I_u^{u.p}(t)$ and $C(u)$ is sufficient to prevent s from being returned as false positive candidate of $LC(u)$ in the enumeration step.*

Proof. Assume that removing just an invalid candidate s from $I_u^{u.p}(t)$ and $C(u)$ where $s \in C(u)$ and $e_D(s,t) \in E(D) \quad \forall t \in C(u.p)$ will cause s to be perceived as valid candidate in local candidate set $LC(u) = \bigcap_{u.b \in N_-^o(u)} I_u^{u.b}(\mu[u.b])$ of u during the enumeration step. Since $s \notin I_u^{u.p}(t) \quad \forall t \in C(u.p)$ and $u.p \in N_-^o(u)$, thus $s \notin LC(u)$. The lemma holds by contradiction. □

Lemma III.2. *Retaining an invalid candidate s from index of forward neighbor $I_{u.f}^u(s) \quad \forall u.f \in N_+^o(u)$ will not yield s as a false positive candidate mapped to u in the enumeration step.*

Proof. Assume that retaining an invalid candidate s in $I_{u.f}^u(s) \quad \forall u.f \in N_+^o(u)$ where $e_D(s,t) \in E(D) \quad \forall t \in I_{u.f}^u(s)$ will cause s to be taken as valid candidate for computing local candidate set $LC(u.f) = \bigcap_{u.f \in N_+^o(u)} I_{u.f}^u(\mu[u])$. Since $s \notin I_u^{u.p}(t) \quad \forall t \in C(u.p)$ and $s \notin LC(u)$ according to Lemma III.1, s is never mapped to u as $\mu[u]$. This contradicts the assumption. Hence, we proof by contradiction that the lemma holds. □

Definition III.3. *Backward Candidate Pruning Refinement (BCPRefine) Rule: A candidate $s \in C(u)$ of a query vertex u that is not connected to any candidate $t \in C(u.b)$ of one or more backward neighbors $u.b \in N_-^o(u)$ can be safely pruned, such that $\exists u.b \in N_-^o(u), N(s) \cap C(u.b) = \varnothing$. Since $e_D(s,t) \notin E(D) \quad \forall t \in C(u.b)$ for some $u.b \in N_-^o(u)$ but $e_Q(u,v) \in E(Q)$, thus s violates the injective function f.*

BCPRefine integrates the backward refinement step into the Backward Candidate Pruning (**BCP**) method through the utilization of two observations, namely Lemma III.1 and III.2, thus reducing the expensive removal of invalid candidates from the candidate set C and auxiliary index I despite consuming more memory. The combined method, BCPRefine, employs a filtering constraint derived from Definition **??** as shown in Definition III.3. It performs the filtering constraint on any query vertex and its backward neighbors, excluding the root vertex in the BFS tree T. As a result, BCPRefine is less complex than the ordinary BCP method but has weaker pruning power.

F. Jump and Redo

Jump and Redo (**JR**) is an optimization method that reduces search branches in the enumeration step given a static enumeration order. The result is a reduction in the quantity of data vertices to be explored. If a search branch on the current query vertex fails, the enumeration step jumps to the nearest backward neighbor in the enumeration order. Meanwhile, each query vertex between the jump (the source and the target query vertices) is mapped to the first candidate in its respective local candidate set. Recalculating the local candidates set for each intermediate query vertex will yield the same local candidates, inclusive of candidates that has been mapped to other query vertices, unless the mapping of the vertex's backward neighbors has been changed. This is because the computation of the local candidates of any query vertex is dependent on the mapping of its backward neighbors. As a result, the enumeration step will only recompute the local candidate set for any query vertex if the mapping of its backward neighbors has altered.

IV. Environment Setup

All code is compiled in C++20 programming language with cmake (version 3.20.1, linux-x86_64 distribution) g++ (GCC) v10.2.0 using O3 optimization and additional flags ("-march=native", "-pthread") on a machine with Intel Xeon W-2145 3.70GHz base clock speed, eight cores processor and 64GB RAM running Ubuntu 18.04.5 LTS OS.

V. Implemented Algorithm

Four state-of-the-art SI indexing algorithms, DP-iso (DPiso)[6], GraphQL (GQL)[14], CFL-Match (CFL)[4] and CECI[5] are implemented to empirically compare against the proposed method, I2Match. We repurposed the source code of the state-of-the-art algorithms implemented by the researchers from a survey [20]. Their codes can be found in the public repository https://github.com/RapidsAtHKUST/SubgraphMatching/. We also integrate the LPF filtering method and JR optimization technique into the CECI, CFL, GQL, and DPiso algorithms, namely CECI-LPF, CFL-LPF, GQL-LPF, DPiso-LPF, CECI-JR, CFL-JR, and GQL-JR algorithms, to measure the effectiveness of the optimization method. DP-iso is not compatible with the JR optimization method, as JR only works on static enumeration order but not dynamic enumeration order.

VI. Inputs

This research considers five real-world datasets (DBLP, Human, Patents, Yeast, and Youtube) and a publicly available synthetic dataset (ERP), undirected in this context and widely used in previous work [7], [21]–[24], to evaluate the performance of the SI algorithms. The specifications of the datasets are described in Table I and Table II. Human and Yeast use edges to represent the interaction between proteins labeled using the Gene Ontology Term. Labels are generated at random and assigned to unlabeled datasets such as Patents, Youtube and DBLP. ERP consists of randomly generated Erdős–Rényi graphs with query graphs that are close to the phase transition between the satisfiable-unsatisfiable regions defined in [25]. Satisfiability denotes whether a query graph is isomorphic to at least one subgraph in a given data graph, such that the query graph is satisfiable if true and otherwise unsatisfiable. On the other hand, the query graphs for the real-world datasets are induced subgraphs extracted from each dataset using the random walk method. These query instances are guaranteed to be satisfiable. The density of a graph is calculated using the formula $density = \frac{2|E|}{(|V|(|V|-1))}$.

Table I: Properties of Query Datasets.

Category	Dataset	#inst	Query Graph					
			$\lvert V(Q)\rvert$		$\lvert E(Q)\rvert$		density	
			min	max	min	max	min	max
Biology	**Human**	900	4	20	3	190	.15	1
	Yeast	900	4	32	3	88	.09	1
Citation	**Patents**	900	4	32	3	210	.09	1
Social	**Youtube**	900	4	32	3	89	.09	.83
	DBLP	900	4	32	3	335	.09	1
Synthetic	**ERP**	200	30	30	128	387	.29	.89

[1] This table reports the specifications of each dataset with number of query graphs (#inst), minimum and maximum number of query vertices ($|V(Q)|$), query edges ($|E(Q)|$) and density.

Table II: Properties of Data Graph.

Category	Dataset	Data Graph							
		$\lvert V(D)\rvert$		$\lvert E(D)\rvert$		$\lvert\Sigma\rvert$	average degree	density	
Biology	**Human**	4,674		86,282		44	36.9200	0.007901	
	Yeast	3,112		12,519		71	8.0456	0.002586	
Citation	**Patents**	3,774,768		16,518,947		20	8.7523	0.000002	
Social	**Youtube**	1,134,890		2,987,624		25	5.2650	0.000005	
	DBLP	317,080		1,049,866		15	6.6221	0.000021	
		min	max	min	max			min	max
Synthetic	**ERP**	150	150	4132	8740	0	91.2718	.37	.78

[1] This table reports the specifications of each dataset with the number of data vertices ($|V(D)|$), data edges ($|E(D)|$), number of labels ($|\Sigma|$), average degree and density.

VII. Metrics

The time taken for the filtering step (filtering time), the enumeration step (enumeration time), and the total time taken (query time) are measured in nanoseconds, however reported in seconds (s). This research excludes memory consumption as a metric, as all competing algorithms never exceed the maximum memory capacity. Meanwhile, the total candidate count is calculated as $\Sigma_{u\in V(Q)}|C(u)|$. The enumeration step of each algorithm is halted at 10^6 embeddings to allow sufficient coverage of search space within a reasonable amount of time in accordance to [4], [20]. In addition, we set the time limit to 0.3×10^3 seconds (5 minutes) for the enumeration step. The performance comparisons of the competing algorithms are divided into two groups. Any query that cannot be solved by one or more algorithms within the time limit is excluded from the average query time and reported as the number of solved queries within the time limit. On the contrary, we report the average query time of any query that can be solved by all algorithms. The number of query edges are mostly greater than the number of query vertices except for some query graphs with $|V(Q)| = 4$ and $|E(Q)| = 3$. Hence, we report the performance evaluation over the number of query edges.

VIII. Results and Analysis

A. *Hardness Analysis*

Figure 4 shows that the query time does not increase with the number of query edges. Most of the queries that took 5 seconds or more to solve are concentrated in the similar area for all algorithms. We set the color scale from 0 to 10 seconds, as any longer query time is considered slow in performance. In fact, most of the queries are solvable in less than 1 second. The l2Match algorithm demonstrates better performance in terms of query time as it solves the majority of the queries within 0 to 5 seconds. Conversely, GQL has the highest number of query instances solved in 10 seconds and above. l2Match and GQL outperform other algorithms in terms of the number of search nodes explored in the enumeration step as evidenced in Figure 5. Furthermore, this proves that the JR optimization method is valid and effective in reducing search branches (the number of search nodes) and query time, as the l2Match algorithm employs an enumeration method that is identical to the CECI algorithm. Further details are discussed in VIII-E.

Table III: Satisfiability and Hardness of the Query Graphs.

Dataset	satisfiability		hardness			
	satisfiable	unsatisfiable	Easy, E	Easy or Hard, EH	Hard, H	Unsolved, U
Human	900	0	855	42	1	2
Yeast	900	0	892	8	0	0
Patents	900	0	757	133	4	6
Youtube	900	0	773	119	7	1
DBLP	900	0	843	44	9	4
ERP	164	36	0	0	1	199

Table IV: Number of Solved Queries.

Dataset	Number of solved queries									
	EH					H				
	l2Match	CECI	CFL	GQL	DPiso	l2Match	CECI	CFL	GQL	DPiso
Human	27	18	35	35	27	0	0	0	1	0
Yeast	8	4	6	6	8	0	0	0	0	0
Patents	131	130	132	6	123	0	0	2	1	1
Youtube	86	51	70	106	93	0	0	0	4	3
DBLP	34	33	38	23	32	0	0	1	7	1
ERP	0	0	0	0	0	0	0	0	1	0

[1] This table presents the number of solved queries in terms of hardness of the query: Easy or Hard (EH), and Hard (H). The best results among the competing algorithms in each dataset are highlighted in blue.

Thus, we classify the hardness of any query graph into four categories:

1) Easy query (E), if all algorithm are able to solve it;
2) Easy or Hard (EH), if more than one algorithms but not all are able to solve the query;
3) Hard (H), if only exactly one algorithm is able to solve it;
4) Unsolved (U), if none of the algorithms is able to solve the query within the time limit.

The hardness of the query graphs is not to be confused with the satisfiability, as an unsolved query (U) may be satisfiable or unsatisfiable, but any other classes, such as E, EH, and H, are surely satisfiable.

The satifiability and hardness of the query graphs are shown in Table III. We further report the number of solved queries in Table IV. CFL solves more EH queries in comparison to the other algorithms. Since the enumeration step is the only part of the algorithm that is responsible for verifying the isomorphism of the subgraphs, an effective ordering method allows the enumeration function to prune invalid search branches as early as possible and reduce enumeration time. Clearly, this indicates that the ordering technique of CFL that utilizes the divide-and-conquer method is more effective than the ordering method of the others. Although GQL solves the majority of H queries by employing the DFS ordering method, it is evidenced that the DFS enumeration order slows down the enumeration process in comparison to CFL, which uses a similar enumeration method. Both l2Match and CECI employ the BFS ordering method to sort the query vertices. However, l2Match ranks second in the EH class and fourth in the H class despite being the fastest algorithm. In summary, the BFS ordering method is impractical and non-versatile to optimize the enumeration step since it cannot reduce unpromising intermediate results by utilizing the effect of Postponing Cartesian Product [4].

Research proves that the Constraint Programming (CP) SI algorithm is much more adept at solving computationally challenging queries and estimating the satisfiability of any query [25]. Nevertheless, results from [24] show that the filter-and-verification algorithms are faster than the CP algorithms in terms of easy queries. Thus, the essential task is to prove that l2Match is more efficient at solving easy queries. Undoubtedly, the experimentation outcomes align with the research objectives and expectations.

B. Average Query Time

l2Match significantly outperforms other algorithms, especially on the lower end of the number of query edges, as reported in Figure 6. It achieves performance improvement[1] over CECI by 19.22%, CFL by 13.11%, GQL by 45.21%, and DPiso by 37.79%. In contrast, the performance of GQL and DPiso are inferior as the number of query edges increases. Furthermore, the average and standard deviation of the results suggest that the l2Match and CFL algorithms scale almost linearly to the number of query edges until they reach a maximum value. The average degree and density of a query graph increase as the number of edges increases. Nonetheless, the number of candidates to verify and the enumeration time decrease as the search is focused on the denser region of the data graph. This observation supports the findings that the hardness of a query is peaked in the phase transition between the satisfiable and unsatisfiable areas [25].

C. Effectiveness of LPI, LPF, and BCPRefine Method in Filtering Step

Figure 7 shows the average performance difference between l2Match and CECI in terms of filtering time and candidate count. A positive difference signifies better performance of l2Match algorithm, and vice versa. It is apparent that l2Match outperforms CECI with the implementation of LPI, LPF, and BCPRefine methods (corresponding to RO1, RO2, and RO3) that reduce the redundant traversals on the query graph and the expensive removal of invalid candidates. On average, l2Match has 14.39% lesser candidate count and 39.85% shorter filtering time than CECI[2].

D. Improvement using LPI and LPF Method in Filtering Step

The effectiveness of the LPI and LPF methods are evaluated and portrayed in Figure 8. One positive trend (DPiso) is seen in the candidate count, and two strong positive trends (GQL and DPiso) are seen in the filtering time. The candidate count of the DPiso algorithm is decreased by 16.19%, on averagely as a result of the optimization in filtering using the LPI and LPF methods[3]; however, it shows no changes in CECI, CFL, and GQL algorithms. It indicates that the pruning power of the LPF and NLF method are equivalent. Besides, LDF's pruning power is weaker than that of the LPF and NLF methods, as observed in the positive difference between DPiso-LPF and DP-iso.

LPI and LPF methods shorten the filtering time by 9.60% (CFL), 20.75% (GQL), and 16.94% (DPiso), respectively, but increase the filtering time of CECI by 0.05%. We speculate that the overhead of accessing a query vertex's neighbors stored in the LPI outweighs the time taken to scan every neighbor of a query vertex. This is true when the query vertex has a minimal number of neighbors. In spite of the shortcoming, LPI and LPF improve the filtering step by cutting down on the number of neighbors to access and verify, ultimately decreasing the filtering time.

E. Improvement using JR Method in Enumeration Step (RO4)

The improvements in the enumeration step of CECI, CFL, and GQL algorithms achieved using the JR method are visualized in Figure 9. JR demonstrates positive improvements when paired with CECI algorithm. CFL-JR occasionally performs worse than the original in terms of query time when the number of query edges range between 80 and 90. Similarly, the query time of the original algorithm peaked in the same range in Figure 6 of Section VIII-B. Nonetheless, the performance of CFL-JR is mostly better than CFL. The difference in the number of search nodes is insignificant for GQL algorithm even though the query time of GQL-JR is, on average, shorter than GQL. The ordering method of GQL prioritizes neighboring vertices with minimum number of candidates. Consequently, the closest backward neighbor of each query vertex in the enumeration order is placed right before itself (distance of 1). Further inspections reveal that the JR method achieves better optimization in reducing exploration of redundant search branches when the spread and distance between the source and target query vertices of a jump is further.

The average improvement achieved with JR method in the enumeration step are 12.35% (CECI), 5.88% (CFL) and 41.62% (GQL) in terms of the query time, and 46.47% (CECI), 55.26% (CFL) and 52.94% (GQL) in terms of the number of search nodes[4].

Acknowledgment

This research was supported by Monash University Malaysia Merit Scholarship Scholarship. I cannot express enough thanks to my supervisors Assoc Prof Wong Kok

[1]Performance improvement is calculated as the percentage of decrease $p = 100 \cdot (t^{algo} - t^{l2Match}) \div t^{algo}$ where t is the average query time and $algo$ represents each competing algorithm.

[2]Average performance different is calculated as the percentage decrease $\overline{p} = 100(\Sigma_{i=1}^{N}(x_i^{CECI} - x_i^{l2Match}) \div x_i^{CECI}) \div N$ where x is the value of candidate count or filtering time, and N is the number of easy (E) queries.

[3]Average performance is calculated directly as the percentage decrease $\overline{p} = 100(\Sigma_{i=1}^{N}(x_i^{ORI} - x_i^{LPF}) \div x_i^{ORI}) \div N$ where x is the value of the query time or the number of search nodes, ORI and LPF represents the original and optimized algorithm respectively, and N is the number of easy (E) queries.

[4]Average performance different calculated as the percentage decrease $\overline{p} = 100(\Sigma_{i=1}^{N}(x_i^{ORI} - x_i^{JR}) \div x_i^{ORI}) \div N$ where x is the value of the query time or the number of search nodes, ORI and JR represents the original and optimized algorithm respectively, and N is the number of easy (E) queries.

Sheik and Dr Soon Lay Ki. Their tremendous guidance and advice are the keys to accomplish this research. A special thank to Julie Holden, who continuously assist me to structure and ameliorate my thesis. I would also like to acknowledge with much appreciation to the advice given by the Milestone Review Panel of Monash University. They inspire me to explore further in Subgraph Matching field and ensure that I am not swayed from my scope.

References

[1] W. Shu-Xi, "The improved dijkstra's shortest path algorithm and its application," *Procedia Engineering*, vol. 29, pp. 1186–1190, 2012.

[2] C. Biemann, "Chinese whispers-an efficient graph clustering algorithm and its application to natural language processing problems," in *Proceedings of TextGraphs: the first workshop on graph based methods for natural language processing*, 2006, pp. 73–80.

[3] M. R. Garey and D. S. Johnson, *Computers and intractability*. freeman San Francisco, 1979, vol. 174.

[4] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1199–1214, ISBN: 9781450335317. DOI: 10.1145/2882903.2915236. [Online]. Available: https://doi.org/10.1145/2882903.2915236.

[5] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1447–1462, ISBN: 9781450356435. DOI: 10.1145/3299869.3300086. [Online]. Available: https://doi.org/10.1145/3299869.3300086.

[6] M. Han, H. Kim, G. Gu, K. Park, and W.-S. 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*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1429–1446, ISBN: 9781450356435. DOI: 10.1145/3299869.3319880. [Online]. Available: https://doi.org/10.1145/3299869.3319880.

[7] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 364–375, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1453856.1453899. [Online]. Available: https://doi.org/10.14778/1453856.1453899.

[8] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 337–348, ISBN: 9781450320375. DOI: 10.1145/2463676.2465300. [Online]. Available: https://doi.org/10.1145/2463676.2465300.

[9] Z. A. Ansari, Jahiruddin, and M. Abulaish, "An efficient subgraph isomorphism solver for large graphs," *IEEE Access*, vol. 9, pp. 61 697–61 709, 2021. DOI: 10.1109/ACCESS.2021.3073494.

[10] S. Sun and Q. Luo, "Subgraph matching with effective matching order and indexing," *IEEE Transactions on Knowledge and Data Engineering*, 2020. DOI: 10.1109/TKDE.2020.2980257.

[11] L. Hong, L. Zou, X. Lian, and P. S. Yu, "Subgraph matching with set similarity in a large graph database," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2507–2521, 2015. DOI: https://doi.org/10.1109/TKDE.2015.2391125.

[12] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Fast subgraph query processing and subgraph matching via static and dynamic equivalences," *The VLDB Journal*, Jun. 2022, ISSN: 0949-877X. DOI: 10.1007/s00778-022-00749-x. [Online]. Available: https://doi.org/10.1007/s00778-022-00749-x.

[13] M. Asiler, A. Yazıcı, and R. George, "Hygraph: A subgraph isomorphism algorithm for efficiently querying big graph databases," *Journal of Big Data*, vol. 9, no. 1, p. 40, Apr. 2022, ISSN: 2196-1115. DOI: 10.1186/s40537-022-00589-0. [Online]. Available: https://doi.org/10.1186/s40537-022-00589-0.

[14] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, (Vancouver, Canada), L. V. S. Lakshmanan, R. T. Ng, and D. Shasha, Eds., ACM, Jun. 2008, pp. 405–418, ISBN: 978-1-60558-102-6. DOI: https://doi.org/10.1145/1376616.1376660.

[15] C. McCreesh, P. Prosser, and J. Trimble, "The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants," in *Graph Transformation*, F. Gadducci and T. Kehrer, Eds., Cham: Springer International Publishing, 2020, pp. 316–324, ISBN: 978-3-030-51372-6.

[16] B. Rowe and R. Gupta, "G-morph: Induced subgraph isomorphism search of labeled graphs on a gpu," in *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings*, Lisbon, Portugal: Springer-Verlag, 2021, pp. 402–417, ISBN: 978-3-030-85664-9. DOI: 10.1007/978-3-030-

85665-6_25. [Online]. Available: https://doi.org/10.1007/978-3-030-85665-6_25.

[17] G. Radu-Iulian, "The gr1 algorithm for subgraph isomorphism. a study from parallelism to quantum computing," in *High Performance Computing and Networking*, C. Satyanarayana, D. Samanta, X.-Z. Gao, and R. K. Kapoor, Eds., Singapore: Springer Singapore, 2022, pp. 83–93, ISBN: 978-981-16-9885-9.

[18] L. Zhu, Y. Yao, Y. Wang, *et al.*, "A novel subgraph querying method based on paths and spectra," *Neural Computing and Applications*, vol. 31, pp. 5671–5678, 2019.

[19] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu, "Treespan: Efficiently computing similarity all-matching," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 529–540, ISBN: 9781450312479. DOI: 10.1145/2213836.2213896. [Online]. Available: https://doi.org/10.1145/2213836.2213896.

[20] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1083–1098, ISBN: 9781450367356. DOI: 10.1145/3318464.3380581. [Online]. Available: https://doi.org/10.1145/3318464.3380581.

[21] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[22] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *arXiv preprint arXiv:1205.6691*, 2012.

[23] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[24] C. Solnon, "Experimental evaluation of subgraph isomorphism solvers," in *Graph-Based Representations in Pattern Recognition: 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19–21, 2019, Proceedings 12*, Springer, 2019, pp. 1–13.

[25] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble, "When subgraph isomorphism is really hard, and why this matters for graph databases," *Journal of Artificial Intelligence Research*, vol. 61, pp. 723–759, 2018.

APPENDIX

Table V: Notations.

Table V – continued from previous column

Notation	**Description**
D, Q	Data Graph and Query Graph
$V(D)$, $E(D)$	Data vertex set and data edge set
$V(Q)$, $E(Q)$	Query vertex set and query edge set
$e_D(s,t)$	data edge connecting data vertex s and t
$e_Q(u,v)$	query edge connecting query vertex u and v
Σ	label set
$L : V \to \Sigma$	Label function that maps each vertex in vertex set V to a label in label set Σ
$L(u)$	Label of query vertex u
$N(u)$	Neighbor vertex set of query vertex u
$N_l(u)$	Neighbor vertex set of query vertex u with label l
$\|N_l(u)\|$	Neighboring Label Frequency (**NLF**) of query vertex u with label l
$\omega_l(u)$	Neighboring Label Offset (**NLO**) of query vertex u with label l
$\pi_l(u) = (\omega_l(u), \|N_l(u)\|)$	Pair of NLO and NLF of query vertex u with label l
$\|u\|$	Degree of query vertex u
$C(u) = \{s \in V(D) \,\|\, \|s\| \geqslant \|u\|, L(s) = L(u)\}$	Candidate set of data vertices that have similar label and greater or equal degree to the query vertex u
I	Index data structure
$I_v^u(s)$	Neighbor vertex set $N(s)$ of data vertex s in $C(v)$ where $s \in C(u)$ and $e_Q(u,v) \in E(Q)$
Φ	Label Pair Index (**LPI**)
Φ_{l_1}	Data vertex set with label l_1
$\Phi_{l_2}^{l_1}$	Data edges that have one data vertex with label l_1 and a neighbor data vertex with label l_2
T	Breadth First Search (**BFS**) tree decomposition of query graph Q

Table V – continued from previous column

Notation	Description
$u.p, u.c$	Parent and child query vertex of query vertex u in BFS tree T
o	Enumeration order is a traversal sequence of query vertex in query graph Q during enumeration step
$N_-^o(u)$	Neighbor query vertex set of query vertex u that are positioned ahead of u in the enumeration order o
$N_+^o(u)$	Neighbor query vertex set of query vertex u that are positioned after u in the enumeration order o
μ	Partial mapping of incomplete subgraph of data graph D to query graph Q
$LC(u) = \begin{cases} \{s \in C(u) \mid \exists e_D(s, \mu[u.b]) \in E(D) \forall u.b \in N_-^o(u)\} & if\ \|N_-^o(u)\| > 0 \\ C(u) & if\ \|N_-^o(u)\| = 0 \end{cases}$	If query vertex u is not root vertex in T, the local candidate set is a subset of $C(u)$ for the query vertex u such that each candidate $s \in LC(u)$ is connected to all candidate of backward neighbors of u in μ, otherwise $LC(u) = C(u)$
$\mu[u]$	A data vertex $s \in LC(u)$ that is mapped to u in μ
$f : V(Q) \rightarrow V(D)$	An injective function that maps query vertices to data vertices such that for all query edge $e_Q(u, v) \in E(Q)$, $L(u) = L(f(u))$, $L(v) = L(f(v))$, $\|f(u)\| \geqslant \|u\|$, $\|f(v)\| \geqslant \|v\|$, and $e_D(f(u), f(v)) \in E(D)$.

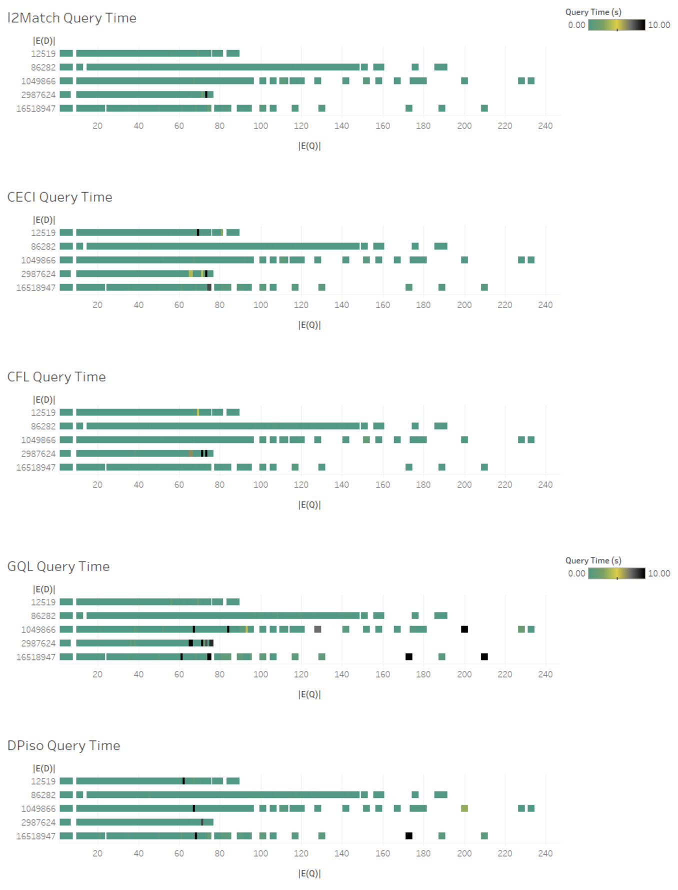


Figure 4: Number of query edge (x-axis), number of data edges (y-axis), and query time (color).

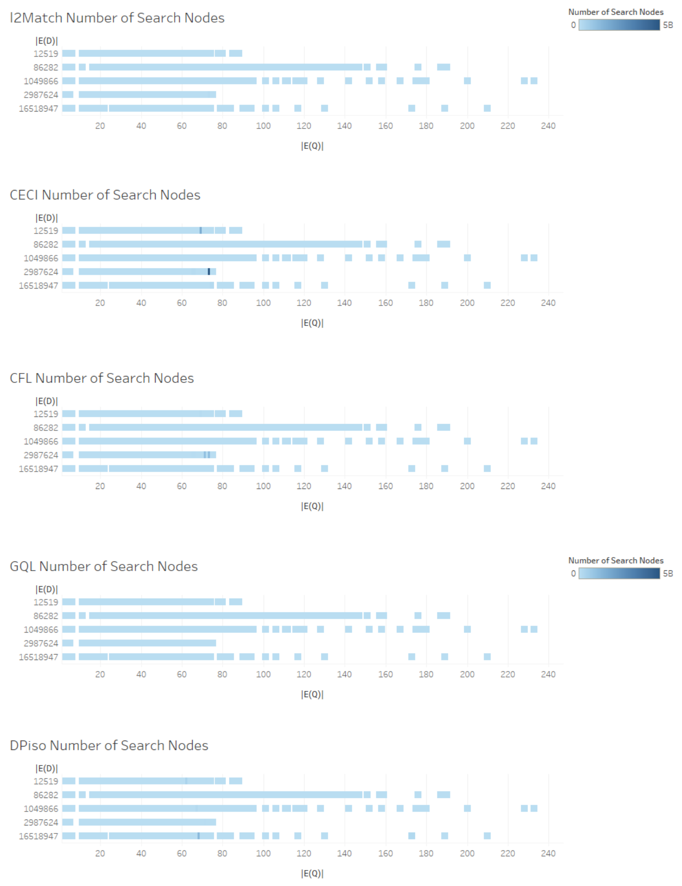
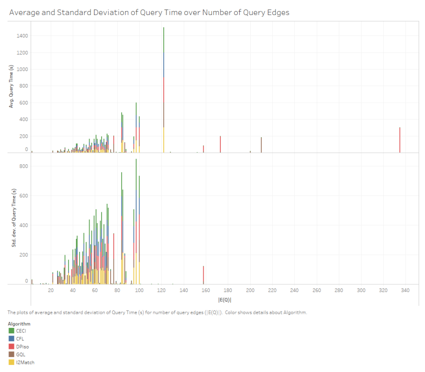
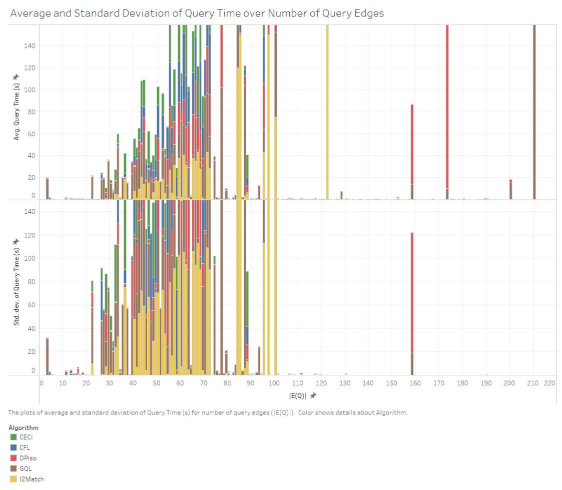


Figure 5: Number of query edge (x-axis), number of data edges (y-axis), and number of search nodes (color).

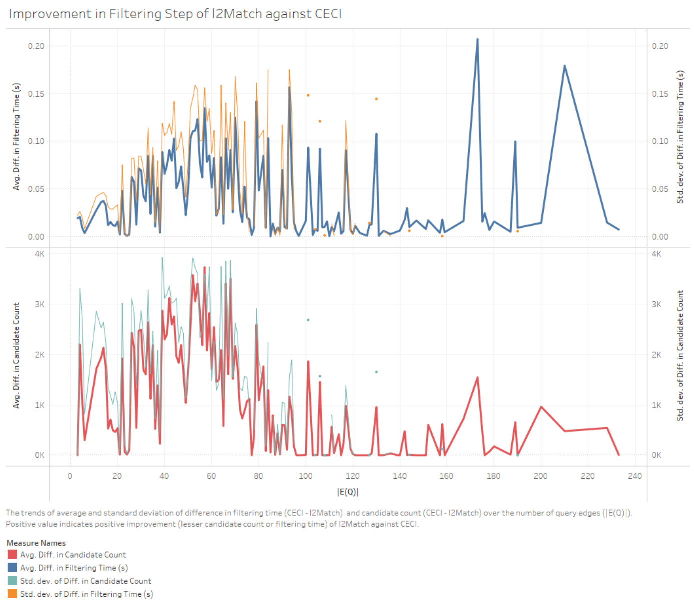


(a) Original size.



(b) Zoomed size.

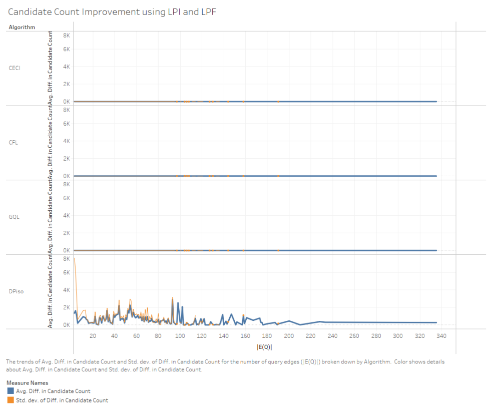
Figure 6: Evaluation on the query time over the number of query edges.



The trends of average and standard deviation of difference in filtering time (CECI - l2Match) and candidate count (CECI - l2Match) over the number of query edges (|E(Q)|). Positive value indicates positive improvement (lesser candidate count or filtering time) of l2Match against CECI.

Measure Names
🟥 Avg. Diff. in Candidate Count
🟦 Avg. Diff. in Filtering Time (s)
🟩 Std. dev. of Diff. in Candidate Count
🟧 Std. dev. of Diff. in Filtering Time (s)

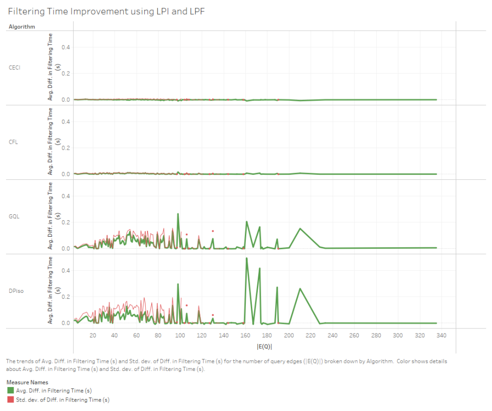
Figure 7: Improvement in filtering step of l2Match against CECI. l2Match employs CECI filtering method and indexing data structure but optimizes it with LPF and BCPRefine techniques.



The trends of Avg. Diff. in Candidate Count and Std. dev. of Diff. in Candidate Count for the number of query edges (|E(Q)|) broken down by Algorithm. Color shows details about Avg. Diff. in Candidate Count and Std. dev. of Diff. in Candidate Count.

Measure Names
🟦 Avg. Diff. in Candidate Count
🟧 Std. dev. of Diff. in Candidate Count

(a) Candidate count.

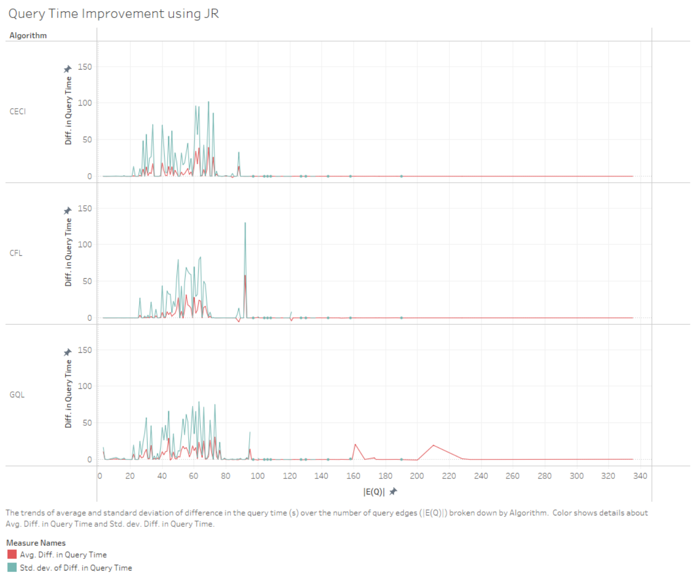


The trends of Avg. Diff. in Filtering Time (s) and Std. dev. of Diff. in Filtering Time (s) for the number of query edges (|E(Q)|) broken down by Algorithm. Color shows details about Avg. Diff. in Filtering Time (s) and Std. dev. of Diff. in Filtering Time (s).

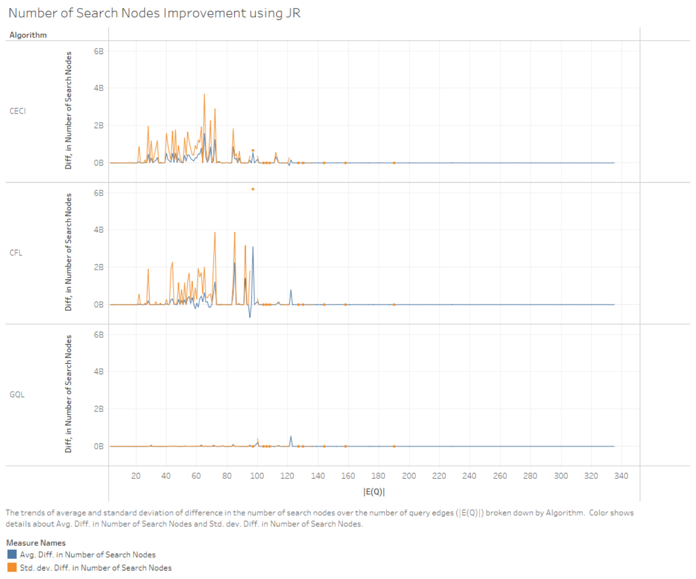
Measure Names
🟩 Avg. Diff. in Filtering Time (s)
🟥 Std. dev. of Diff. in Filtering Time (s)

(b) Filtering time.

Figure 8: Evaluation on the average and standard deviation of difference in the candidate count and the filtering time in the filtering step over the number of query edges.



(a) Query time.



(b) Number of search nodes.

Figure 9: Evaluation on the average and standard deviation of difference in the query time and the number of search nodes explored in the enumeration step over the number of query edges.