Performance and Scalability of Indexed Subgraph Query Processing Methods

Foteini Katsarou
School of Computing Science
University of Glasgow, UK
f.katsarou.1@research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Graph data management systems have become very popular as graphs are the natural data model for many applications. One of the main problems addressed by these systems is subgraph query processing; i.e., given a query graph, return all graphs that contain the query. The naive method for processing such queries is to perform a subgraph isomorphism test against each graph in the dataset. This obviously does not scale, as subgraph isomorphism is NP-Complete. Thus, many indexing methods have been proposed to reduce the number of candidate graphs that have to underpass the subgraph isomorphism test. In this paper, we identify a set of key factors-parameters, that influence the performance of related methods: namely, the number of nodes per graph, the graph density, the number of distinct labels, the number of graphs in the dataset, and the query graph size. We then conduct comprehensive and systematic experiments that analyze the sensitivity of the various methods on the values of the key parameters. Our aims are twofold: first to derive conclusions about the algorithms' relative performance, and, second, to stress-test all algorithms, deriving insights as to their scalability, and highlight how both performance and scalability depend on the above factors. We choose six well-established indexing methods, namely Grapes, CT-Index, GraphGrepSX, gIndex, Tree+Δ, and gCode, as representative approaches of the overall design space, including the most recent and best performing methods. We report on their index construction time and index size, and on query processing performance in terms of time and false positive ratio. We employ both real and synthetic datasets. Specifically, four real datasets of different characteristics are used: AIDS, PDBS, PCM, and PPI. In addition, we generate a large number of synthetic graph datasets, empowering us to systematically study the algorithms' performance and scalability versus the aforementioned key parameters.

1. INTRODUCTION

Graphs show great representative power for complex relationships, such as social networks [8], chemical compounds and proteins [1, 18, 14, 12], etc. Relevant datasets vary wildly on several key characteristics, such as the number of graphs in the dataset, the number of nodes per graph, the average density of the graphs in the dataset, and the size of the set of distinct node labels. One of the most frequently arising query types in such systems is *subgraph queries*, where a graph q is given as a query and the system must return those graphs in the dataset that contain the query graph. A naive way to solve this problem, is to perform subgraph isomorphism testing on all the graphs in the dataset. However, the subgraph isomorphism problem is NP-complete and by taking into consideration the number of the graphs in the dataset, this often proves to be too computationally expensive. To this end, many, so-called *filter-and-verification methods* have been proposed to alleviate the problem. These solutions utilize an index based on features (i.e., substructures) of the graphs to filter out some of those that definitely do not contain q; however, the graphs remaining after the filtering – called the *candidate set* – may not actually contain q (i.e., the filtering process can produce false positives). Due to this, a verification stage is required, during which q is tested for subgraph isomorphism against all of these remaining graphs. The main premise of these algorithms is that the candidate set is usually much smaller in size than the complete dataset.

Given the importance of the problem and the large attention it has received in the research community, with this paper we provide a systematic and comprehensive evaluation of the performance and scalability of a representative set of related methods, which includes the most recent and most competitive approaches and approaches representing different key decisions in the design space. More specifically, the contributions of this paper are:

1. *Algorithm Performance.* A comprehensive study of the performance of related methods. A set of methods is selected to represent different key design decisions with respect to the type of graph features indexed (i.e., paths, trees, cycles, subgraphs) and the method for generating graph features (i.e., based on frequency mining or exhaustive enumeration of graph features). Further, this set of methods includes the most recent and higher-performing methods (such as GraphGrepSX, Grapes, and CT-Index). Our performance metrics include query indexing time and space, as well as query processing time and false positives.
2. *Algorithm Scalability.* Our experiments also aim to

Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

stress-test the above methods, deriving conclusions on the methods' scalability. The existing work focuses only on performance issues – i.e., time and space comparison of the algorithms – and fails to look at scalability issues – i.e., what is the performance of the algorithm when both the dataset and the graphs grow large and/or more complex.

3. *A systematic evaluation of performance and scalability.* We employ both real and synthetic graph datasets. Specifically, four real datasets of different characteristics are used: AIDS, PDBS, PCM, and PPI (see [9]). Furthermore, a very large number of synthetic datasets are generated that facilitate a systematic study on the dependence of the algorithms' performance and scalability on the key problem parameters (e.g., number of nodes, graph density, number of distinct labels, number of graphs, and query graph size).

Such a scalability and performance showdown is currently very much lacking. Most related works are tested against the AIDS antiviral dataset[14] and synthetic datasets, formed of many small graphs. These sets are not adequate to provide definitive conclusions on how an algorithm is influenced by the characteristics of the graphs. Of these works, Grapes [9] alone used several real datasets; however, the authors did not evaluate scalability. Also, their performance evaluation did not include a systematic exploration of the effect of the key problem parameters. The iGraph comparison framework [10], which implements several such techniques, compared the performance of older algorithms (up to 2010). Since then, several, more efficient algorithms have been proposed (e.g., GraphGrepSX[2], Grapes[9], CT-Index[13]). Finally, virtually all of these works report on different metrics; thus, no concrete conclusion can be reached regarding their relative scalability. In order to address all above problems, we conducted a systematic and comprehensive evaluation on existing implementations and we report the results.

The remainder of the paper is organized as follows. Section 2 provides useful definitions about the subgraph query problem and a brief summarization of the related work. Section 3 provides a detailed presentation of the competing algorithms chosen for this paper. Section 4 describes the setup of the experiments, the configuration of the algorithms, and the characteristics of each dataset and query workload. Section 5 discusses the results of our performance evaluation. Finally, Section 6 summarizes key insights gained through our evaluation and concludes the paper.

2. BACKGROUND

2.1 Definitions

All of the related indexing algorithms can in theory support arbitrary graphs with labels on both vertices and edges; however, the available implementations of several of them can only handle undirected graphs with labels on vertices. We thus focus on graph datasets with such graphs.

Definition 1 (Graph). *A graph $G = (V, E, L)$ is defined as the triplet consisting of the set $V = \{v_i\}, i = 1, ..., n$ of vertices of the graph, the set $E = \{(v_j, v_k)\}, j, k = 1, ..., n$ of edges between vertices in the graph, and a function $L : V \to \mathcal{L}$ assigning a label $l \in \mathcal{L}$ ($\mathcal{L}$ being the set of all possible labels) to each vertex $v \in V$.*

In this work we consider undirected graphs. We assume that, in each graph, each vertex has a unique identifier. Note that, by the above definition, each node in a graph can have only one label, but any given label can be assigned to multiple nodes in a graph.

Definition 2 (Graph Isomorphism). *Two graphs $G = (V, E, L)$ and $G' = (V', E', L')$ are isomorphic iff there exists a bijection $I : V \to V'$ that maps each vertex of G to a vertex of G', such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$, $L(u) = L'(I(u))$, $L(v) = L'(I(v))$, and vice versa.*

Definition 3 (Subgraph Isomorphism). *A graph $G = (V, E, L)$ is subgraph isomorphic to a graph $G' = (V', E', L')$, denoted by $G \subseteq G'$, iff there exists an injective function $I : V \to V'$ such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$ and $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$. Graph G is then called a subgraph of G' and G' is called a supergraph of G; equivalently, we say that G is contained in G'. Note that subgraph isomorphism is injective and thus there may exist edges in E' for which there are no corresponding edges in E.*

Definition 4 (Graph Density). *The density d of a graph $G = (V, E, L)$ is defined as the quotient of the division of the number $|E|$ of edges in the graph over the number of edges in a complete graph with the same number of vertices. In an undirected graph with $|V|$ vertices, the latter is equal to $\frac{|V| \times (|V|-1)}{2}$ edges, and thus:*

$$d = \frac{2 \times |E|}{|V| \times (|V| - 1)}, d \in [0, 1] \tag{1}$$

Definition 5 (Average Degree). *The degree of a node v in a graph $G = (V, E, L)$ is defined as the number of edges in the graph having v as an endpoint. The average degree avg_{deg} of graph G is then defined as the average of the degrees of all vertices in the graph. For undirected graphs:*

$$avg_{deg} = 2 \times \frac{|E|}{|V|} \tag{2}$$

2.2 Related Work

As mentioned previously, a naive way to process a subgraph query is to check the query graph for subgraph isomorphism against each graph in the dataset. As subgraph isomorphism is an NP-complete problem, and graph datasets may contain a large number of graphs, this procedure often gets too time consuming. To this end, many indexing methods have been proposed over the years, attempting to reduce the set of graphs against which to test for containment. [23] provides an extensive discussion of related methods (although some of the methods considered in our work were not mentioned in [23]).

The design space is characterised through a classification of related works in 4 major categories: (i) type of indexed features[1]: paths, trees, simple cycles, or graphs; (ii) approach for extracting said features from indexed graphs: i.e., exhaustive enumeration or frequent mining techniques; (iii) index data structure: hash table, tree, trie; and (iv) whether the index stores location information or not. Moreover, all

[1] In this work we use the term "feature" to refer to substructures of indexed graphs used to produce the index, independently of whether these are then stored in the index.

such algorithms operate in three stages: (a) index construction, (b) filtering and creation of a set of candidate matching graphs, and (c) verification of containment of the query graph in the latter.

Index Construction. In this stage, features are extracted from the dataset graphs and indexed in an appropriate data structure. Depending on the algorithm, these features can be (a) simple paths[2, 7, 9, 28], (b) trees[15, 11, 25], (c) graphs[5, 19, 20, 21, 23, 27], or (d) a combination of trees and graphs/cycles[13, 27]. Additionally, the features can be extracted from the graphs by either (i) exhaustively enumerating all such features across all graphs[2, 13, 19, 28], or (ii) mining the dataset graphs for frequent patterns[5, 11, 20, 21, 23, 25, 27]. [23] reuses the frequent feature extraction primitives of previous algorithms (e.g., [5, 15, 21, 27]), and is thus able to function with several feature types.

In the case of frequent feature mining algorithms, the support ratio of a feature is defined as the percentage of graphs in the dataset containing it, and a feature is considered *frequent* if its support ratio is above some algorithm-specific threshold. Moreover, the discriminative ratio of a feature is a metric characterizing the pruning power of a feature compared to its sub-features. All frequent mining-based works mentioned above provide differing formulae for this metric; we thus do not provide a formula here but rather refer interested readers to the cited papers for more details.

In all cases, a limit is imposed on the size of the indexed features, where the size of a feature is defined as the number of edges comprising it. Features are identified by their canonical label; i.e., a unique representation of each feature, computed on the labels of the vertices of the feature using an algorithm appropriate for the feature's structure (path, tree, etc.). A list of the IDs of graphs containing a feature is also associated with that feature and stored in the index. Additionally, some of the algorithms choose to maintain *location information*, such as the id of the first node in each path feature [9], or the id of the node at the center of a tree feature [25]. Last, all this is organized in algorithm-specific structures, such as hash tables, prefix trees, tries, or lattices.

Filtering. In this stage, the query graph is first looked up in the index. If an exact match is found, the related graph IDs are returned. Otherwise, the query graph is broken up into features of the same form as those used to create the index. These are then matched against the information in the index structure, resulting in a set of graphs possibly containing the query graph, called the *candidate set* for the given query. In most algorithms the candidate set is computed as the intersection of the sets of graphs containing each feature of the query graph. Additionally, the algorithms that also store location information take advantage of this added knowledge at this stage for further filtering.

Verification. The above filtering stage may well produce false positives as graphs in the dataset may contain all (size-limited) features of a query graph but not the graph itself. To this end, a final verification step is necessary, consisting of testing the query graph for subgraph isomorphism against the graphs in its candidate set. Most algorithms perform this test using the VF2 algorithm[6], with the exception of [13] and [25] which employ algorithm-specific tests.

All aforementioned algorithms reported performance results against datasets consisting of a larger number of small graphs (e.g., the AIDS antiviral dataset containing 40,000 graphs each consisting of 45 nodes and 47 edges on average, the PubChem dataset containing 1 million graphs each consisting of 24 nodes and 26 edges on average, etc.), often providing no proper insight on the performance and scaling of the algorithms against considerably larger and more complex datasets and query workloads. Of these works, Grapes[9] was the only one to provide results against datasets with larger graphs (PDBS, PCM, PPI, to be discussed shortly), but the number of graphs in these datasets was quite small (PDBS contains 600 graphs, PCM contains 200 graphs, and PPI only 20 graphs). Moreover, virtually all such works reported a different set of metrics, making comparisons between the various algorithms a very hard task.

iGraph[10] provided a comprehensive comparison of indexing methods for subgraph query processing. Our work replicates some of the experiments in [10]. However, our work complements and surpasses iGraph in several ways. First, we consider several indexing algorithms that were published after [10] and were proven to be significantly superior to those tested by the latter (often by orders of magnitude). Second, in iGraph, scalability was not addressed; all the previous papers, and specifically iGraph, focus on performance evaluation against small graphs (especially in terms of number of nodes), while their "large datasets" are typically only large in terms of the number of graphs in the dataset and not with respect to the size/complexity of the graphs. Third, we performed a systematic study on performance and scalability based on 5 characteristics of a graph dataset/workload: the number of nodes, the density of the graphs, the number of distinct labels, the number of graphs in the dataset, and the query size.

[16] has looked at subgraph query processing for datasets consisting of a single very large (billion-node) graph. This work takes a totally different approach, by not building an index at all and instead utilizing a memory cloud and massively parallel computing primitives. Its authors motivate their approach by claiming that index-based solutions do not scale, and by providing theoretical arguments based on the asymptotic complexity of the latter (but no experimental evaluation of their approach against index-based techniques). Our work complements and substantiates this claim, by providing hard numbers and a systematic examination of the breaking points of each index algorithm type, across a large number of datasets of varying characteristics.

Last, there has been considerable work on the subjects of approximate graph pattern matching and of supergraph query processing. In the first case, related techniques (e.g, [11, 13, 17, 19, 22, 26], etc.) do perform subgraph matching, but with support for wildcards and/or approximate matches. In the second case, the related algorithms (e.g., [24, 3]) return those graphs in the dataset which are contained in the query graph (as opposed to containing the query graph; see [24] for an overview of related approaches). All these algorithms are not directly related to our work, as we focus on exact-match index-based subgraph query processing, and have thus been omitted from our evaluation.

3. COMPETING ALGORITHMS

In order to cover as much of the design space as possible, we opted to perform an extensive comparison of six of the above algorithms. These algorithms were purposefully selected to represent different points in the design space, characterised by their use of: different feature types, different feature extraction approaches, different index data structures, and different filtering and verifications processes. We chose to evaluate a set of representatives of the various regions in the design space, as opposed to providing an exhaustive (and unwieldy) examination of all subgraph query processing algorithms to date (amounting to dozens). More specifically, we compare CT-index[13], GCode[28], gIndex [21], Grapes[9], GraphGrepSX[2], and Tree+Δ[27]. We justify our selection of algorithms and discuss their characteristics below.

gIndex[21] uses a frequent mining approach, with graph-structured features. It uses both the features' support ratio and discriminative ratio to decide whether a feature is frequent or not, and indexes these features in a prefix tree. It uses no location information, and thus it only stores a graph ID list per feature. During query processing, [21] enumerates all graph-structured fragments of the query graph up to a maximum fragment size, in a way that ensures that (a) smaller fragments are enumerated before larger ones, by starting with fragments of size one and expanding each fragment with one additional edge at a time, and (b) if a fragment does not appear in the index, no supergraphs of that fragment will be produced. Then, the candidate set of the query is computed as the intersection of the graph ID lists of the largest fragments along each expansion path. Finally, the verification is performed by comparing the query graph against all candidate graphs using the VF2 algorithm. gIndex was chosen to represent the frequent subgraph mining algorithms, as its performance exhibited the same trends as other algorithms of its type ([5, 23]) while its implementation was relatively stabler, e.g., compared to that of [5], which proved quite problematic, for larger datasets.

Tree+Δ[27] also uses a frequent mining approach, but initially only indexes tree-structured features of up to a predefined size. The feature information is stored in a hash table. Like [21], no location information is maintained. In the query processing phase, all tree-structured fragments of the query graphs are enumerated and looked up in the index; the candidate set is then computed as the intersection of the graph ID lists corresponding to these fragments, and a final verification step is performed using the VF2 algorithm. However, [27] takes an extra step: in addition to trees, the algorithm also enumerates simple cycles found in query graphs, which it then extends by adjacent edges. Those cycle-based structures that are found to be discriminative enough (based on a predefined threshold on their discriminative ratio) are added to the index structure and used just like tree-structured features for subsequent queries. [27] was chosen over related frequent tree mining algorithms ([11, 25]) due to its being the best overall performer.

gCode[28] takes a different route and chooses an exhaustive enumeration approach. First, it enumerates all paths of up to a predefined size. Given these paths, it then produces vertex *signatures*, consisting of three components. The first two components are a counter-string encoding of the labels of vertices in each path, and a counter-string encoding the neighbors of each vertex in each path. The third component is computed as follows: (i) first, for each node in the dataset, the algorithm creates a "*level-N path tree*" rooted at said node and consisting of all length-N paths starting at that node; (ii) this tree is encoded in an adjacency matrix form; (iii) the eigenvalues of the matrix are computed and sorted by value; and (iv) the top-m such values (for some user-configurable m) are used as the third component of the vertex signature. For every graph in the dataset, all these vertex signatures are then combined to form the graph's *code*. All graph codes are finally stored in a balanced search tree. When a query comes in, first [28] follows the same process as above to construct a graph code for the query graph. This code is then compared against the codes of graphs in the dataset. This results in a first set of candidate graphs, which is then further pruned by comparing the individual vertex signatures of the query graph and candidate graphs. Finally, verification is performed by comparing the query graph against all graphs in the final candidate set using the VF2 algorithm. gCode was chosen to represent algorithms encoding (but not storing) exhaustively enumerated path-based features, as it is rather unique in this sense; it is also a prime example of an algorithm which, albeit considerably slower than its competitors, manages to outscale them for certain very large input cases.

CT-Index[13] also uses exhaustive enumeration to build its index. It uses path-, tree- and cycle-structured features (of up to a user-configurable size). The canonical labels of all such features are then combined and hashed, producing a fixed-size bit array *fingerprint* for each graph in the dataset. Neither this algorithm maintains any location information. During query processing, a similar fingerprint is created for the query graph and is then compared against the fingerprints of all graphs in the dataset via a bitwise-AND operation. This produces a candidate set which is then fed to a verification stage utilizing a modified VF2 algorithm with additional heuristics. [13] was chosen to represent algorithms encoding (but not storing) exhaustively enumerated features whose structure is more complex than that of [28] above; [13] is also rather interesting in that it showcases the impact of trading off filtering power for a very fast matching approach and a low memory overhead, and offsetting the former with a very fast subgraph isomorphism algorithm.

GraphGrepSX[2] (GGSX for short) enumerates all paths up to a maximum length using depth first search and organizes them in a suffix tree. Each node of the suffix tree also stores the graph-id list and the number of occurrences of the corresponding path feature in each graph it appears in. During query processing, maximal paths (of the same maximum length as above) of the query graph are extracted and organized in a query suffix tree, which is then compared against the index structure. Unmatched branches of the index are pruned away, and a further filtering is performed based on the frequencies of features in each graph. This process produces the candidate set, which then undergoes a verification stage using VF2. GraphGrepSX was chosen over related algorithms due to its superior performance.

Finally, Grapes[9] also uses an exhaustive enumeration approach, indexing paths of up to a maximum length. However, in addition to the paths' canonical labels, Grapes also maintains location information in the form of the ID of the starting node of each path for each graph it appears in, as well as a counter denoting how many times each feature appears in each such graph. This information is then indexed using a trie. An added benefit of Grapes is that it is the only of the above algorithms which was designed specifically to support parallel execution of both its indexing and query processing chores. In the former case, this is accomplished via a smart assignment of graph nodes to threads so that

		AIDS	PDBS	PCM	PPI
Dataset	# graphs	40000	600	200	20
	#disconnected graphs	3157	360	200	20
	#labels	62	10	21	46
Per Graph	Avg #nodes	45	2939	377	4942
	StdDev #nodes	21.7	3215	186.7	2648
	Avg #edges	46.95	3064	4340	26667
	Avg density	0.0475	0.0007	0.0612	0.0022
	Avg degree	2.09	2.06	23.01	10.87
	Avg #labels	4.4	6.4	18.9	28.5

Table 1: Characteristics of real datasets

each thread can produce a complete and disjoint part of the final trie, without needing to synchronize with the rest. During query processing, the query graph also undergoes the same (parallel) process of path enumeration and trie construction. The query trie is then compared against the index trie and non-matching parts are pruned away. The "surviving" parts of the trie are further reduced by taking into account the aforementioned location information, and then translated into a set of connected components per graph. This set then consists the candidate set of this algorithm. In the verification stage, the query graph is tested for subgraph isomorphism against each of these connected components in parallel, with each such component assigned to a different thread. Although [9] is in the same (conceptual) region of the design space as [2] above, it was chosen to showcase the effects of parallelized computation and of trading off space to maintain location information for a higher filtering capability. Moreover, [9] is a prime example of a high performer – being one of the fastest algorithms regarding query processing times across most of our scenarios – which is however occasionally outscaled by "inferior" algorithms when some problem dimensions become large.

4. THE EXPERIMENTAL FRAMEWORK

4.1 Setup

All experiments were conducted on a Windows 7 SP1 host, featuring 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB Cache, 8 cores/16 threads per CPU) and 128GB of RAM. For each experiment, a time limit of 8 hours[2] was imposed, after which the experiment was terminated.

For Tree+Δ, gIndex and gCode we used the implementations provided by [10]. For all remaining algorithms, we used the implementations provided by their respective authors. In the case of [9], we had to alter the source code so that the VF2 verification step returns after the first match of the query graph against a connected component for an indexed graph, as opposed to the original implementation which was returning all possible matches; this was necessary as all other algorithms return the first match by default.

We used the default values for the input parameters of compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. More specifically:

- For gIndex, the maximum feature size was set to 10, the support ratio to 0.1, and the discriminative ratio to 2.0. The same parameters were used for both indexing and query processing.
- For Tree+Δ the maximum feature size was set to 10, the support ratio to 0.1, and the discriminative ratio[3] to 0.1. For query processing, the support ratio threshold to add new features to the index is set to 0.8.
- For gCode, paths of up to size 2 were used to construct the vertex signatures, and the top 2 eigenvalues are maintained. Additionally, vertex label and neighbor label bit-strings were both 32 bits long.
- For CT-Index, we created 4096-bit fingerprints by exhaustively enumerating trees and cycles of up to length 4. Note that [13] uses trees of size 6 and cycles of size 8, but [9] showed that a size of 4 for the features results in a somewhat worse filtering power but a significantly lower indexing and query processing time.
- For GGSX, we enumerated paths of up to a size of 4.
- For Grapes, we used 6 threads and enumerated paths of up to a size of 4.

4.2 Real and Synthetic Datasets

We tested the performance and scalability of these algorithms against (a) a set of real datasets provided by [9] (i.e., AIDS, PDBS, PCM, PPI), and (b) synthetic datasets created using the widely used GraphGen[4] generator.

Table 1 summarizes the characteristics of the real datasets. All four datasets differ across all of the characteristics which we identified as significant; specifically, AIDS consists of a large number of small and low average degree graphs, PDBS contains a moderate number of large but low average degree graphs, PCM is comprised of a moderate number of medium-sized but high average degree graphs, and PPI includes few large but medium average degree graphs. Thus, they provide individual data points across the evaluation space, but are not adequate to examine the algorithms' scaling across datasets of varying sizes and complexities.

Concerning the synthetic datasets' generation, we took a rather systematic approach. First, we examined the values of the core input parameters for the four real datasets (AIDS, PDBS, PCM, PPI) and established an initial set of relevant values:

- Mean number of nodes per graph: {50, 200, 400, 4000},
- Mean graph density: {0.005, 0.025, 0.05, 0.075},
- Number of labels in the dataset: {10, 20, 40, 60},
- Number of graphs in the dataset: {1000, 10000}.
- Query size: {4, 8, 16, 32}.

We then tested the algorithms using all possible combinations of these parameter values ($4 * 4 * 4 * 2 = 128$ cases in the indexing phase and $4 * 128 = 512$ in the query processing phase). Alas, the results revealed that many of the algorithms could not produce an index or process queries for most of these combinations. We then computed a set of "sane defaults", so that they represent a challenging case but for which all algorithms could produce results; namely, 200 nodes per graph, average density 0.025, 20 distinct labels and 1000 graphs in the dataset. We then executed several experiments to study the scalability of the various algorithms, varying one parameter at a time to examine its effect on the various metrics and algorithms.

[2] As a matter of fact, we waited for more than 24 hours before terminating those experiments exceeding the 8-hour limit, but to no avail.

[3] Tree+Δ uses a different formula than gIndex to compute the discriminative ratio, hence the different parameter value.

GraphGen allows the parametrization of all above mentioned key parameters. Specifically, it creates graphs through the following steps:

1. The user specifies the number of distinct labels, of distinct edges, and of graphs in the dataset, as well as the average graph density and graph size;
2. GraphGen produces an *alphabet* of distinct edges, consisting of all possible pairs of distinct node labels;
3. Then, for every new graph, GraphGen computes a random size (number of edges) and density, following a normal distribution around the aforementioned averages and a standard deviation of 5 and 0.01 respectively, and iteratively selects a (uniformly distributed) random edge from the *alphabet*, adding it to the current graph, until the requested size/density is reached or the system runs out of edges to use.

It is also worth mentioning that several of the graphs in the real datasets are disconnected, whereas all graphs in the synthetic datasets are connected. Moreover, for the vast majority of the input parameter values considered in our setting, the datasets generated by GraphGen consisted almost exclusively – more than 95% of the graphs in the dataset – of graphs with cycles (i.e., not trees or paths). There were only two exceptions to this rule: datasets with only 50 nodes per graph, where almost half of the graphs were tree-shaped, and datasets with average graph density of 0.005, where 8.5% of the graphs contained no cycles.

4.3 Query Workloads

Given the number of query graphs and their desired size (in number of edges), queries are constructed as follows:

1. Select a graph uniformly at random from the dataset;
2. Select a node uniformly at random from said graph;
3. Starting from that node, perform a random walk;
4. Maintain the graph created by the union of visited nodes and travelled edges;
5. When the desired query graph size is reached, terminate and return the above graph as the new query.

We created query graphs with 4, 8, 16, and 32 edges, to match the query graph sizes used by related work. As these query graphs are actually subgraphs of the various datasets, they have the same characteristics (on average) as the latter with regard to density and distribution of labels.

In order to evaluate the filtering power of the algorithms, we use the false positive ratio, defined as:

$$FP = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \frac{|C\{q\}| - |A\{q\}|}{|C\{q\}|} \qquad (3)$$

where $|\cdot|$ denotes set cardinality, $\mathcal{Q}$ is the set of all queries in each query workload, and $C\{q\}$ and $A\{q\}$ are the candidate set and answer set respectively for query q.

5. EVALUATION RESULTS

5.1 Real Datasets

Figures 1(a) and 1(b) present the time and size requirements to perform the index creation with all algorithms. Grapes and GGSX are the only algorithms which managed to complete indexing for all datasets in the 8-hour time limit. Grapes consistently outperformed the other methods in terms of indexing time, often by at least one order of magnitude; conversely, its index size grows quite large compared to all but Tree+Δ's. Figures 1(c) and 1(d) present the query processing time and false positive ratio respectively. Again, Grapes outperforms all contenders in processing time, with the sole exception of GGSX on the PPI dataset. The fact that there is no result for gCode for the PDBS dataset, is due to its implementation not being able to handle signatures of the size required for this dataset and thus crashing.

5.2 Synthetic datasets

We will now focus on stress-testing the various algorithms using synthetic datasets, with the intent of both covering the space of possible parameter value combinations, and exploring the breaking points of the various indexing and query processing algorithms. Unless otherwise noted, we are using the sane defaults mentioned above to generate the graph datasets and query workloads.

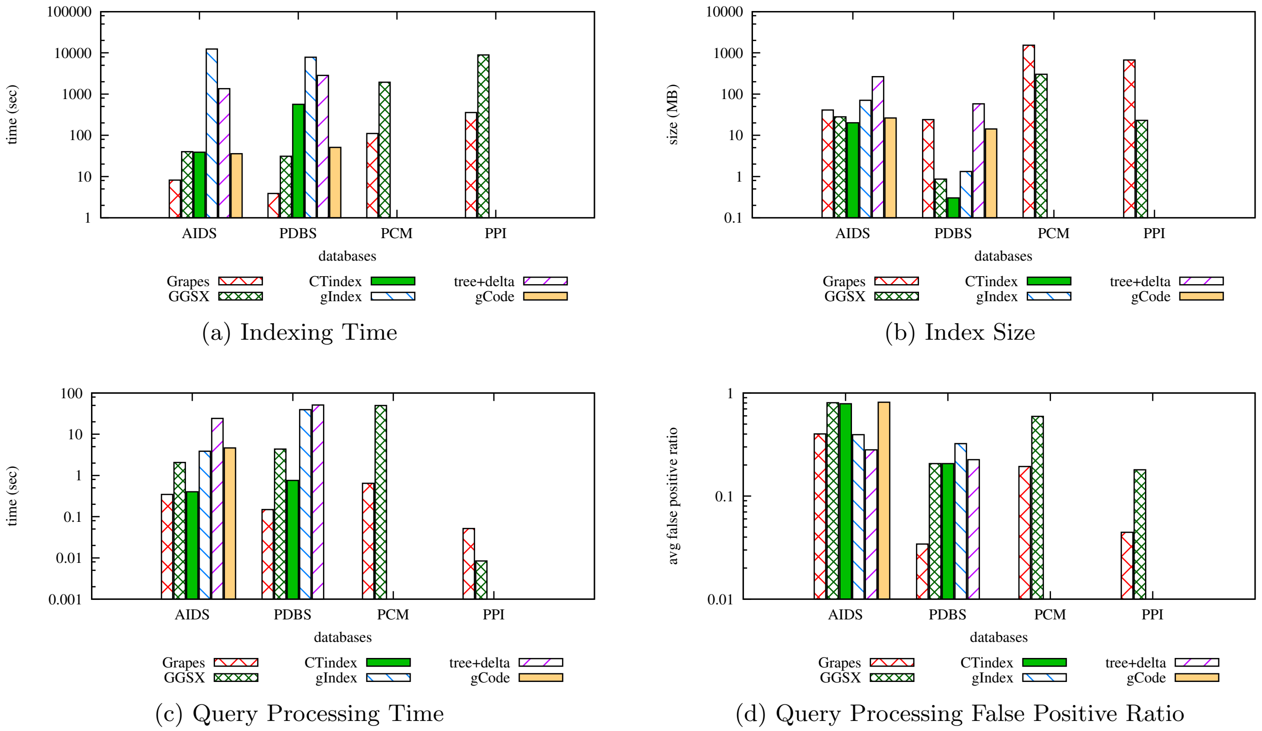
5.2.1 *Number of nodes*

First, we vary the number of nodes per graph in the dataset; more specifically, we have created datasets consisting of graphs with 50, 75, 100, 125, 150, 175, 200, 250, 300, 400, 500, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 nodes. Please note that, given a fixed value for the average graph density, a linear increase in the number of nodes translates to a quadratic increase in the number of edges in the graph (see equation (1)).

Figures 2(a) and 2(b) present the time and size results for the index construction for each algorithm. For small graphs (less than 175 nodes), Tree+Δ is marginally better than Grapes in index construction time. For larger graphs, Grapes takes the lead, being faster than the rest by at least one order of magnitude. GGSX comes second, with gCode and CT-Index being third and fourth; gIndex and Tree+Δ fail to produce an index even for as few as 250-300 nodes per graph. This result is a direct artefact of the complexity of the indexed features and the methods of feature extraction. Frequent feature mining is known to be a very computationally costly process[10] and thus gIndex and Tree+Δ have the worst running times; moreover, as graphs are more complex (and more numerous) than trees, gIndex fairs worse than Tree+Δ. CT-Index and gCode exhaustively enumerate their features and are thus faster than the frequent mining approaches; however, the computation of fingerprints/signatures is non-trivial and this shows in the results. Moreover, tree features are more complex (and numerous) than paths, and thus CT-Index fairs worse than gCode. Last, Grapes and GGSX both exhaustive enumerate paths (leading to the lowest running times), with the former having an edge due to its multi-threaded implementation.

On the index size front, CT-Index and gCode have the smallest indices since these algorithms only store fixed-size fingerprints/signatures per graph (gCode's index is larger as it also stores node signatures). The index size for GGSX and Grapes levels out after some point; as these algorithms use a prefix tree/trie to store indexed paths, as soon as all possible paths up to the size limit have been produced, the index structure doesn't grow any further (other than location/frequency information being recorded). Last, the frequent mining algorithms start off with a small index, but the larger the graphs the more the frequent features, and this exponential increase is evident in both of these figures.

Figures 2(c) and 2(d) depict the query processing time and false positive ratio. The x-axis extends only up to 800



(a) Indexing Time

(b) Index Size

(c) Query Processing Time

(d) Query Processing False Positive Ratio

Figure 1: Indexing (a, b) and query processing (c, d) results over the real datasets

nodes due to the fact that no algorithm could handle queries on datasets with larger graphs within the 8-hour limit. Although Grapes managed to complete the indexing stage for larger cases (up to 1800-node graphs), in the query processing phase the increase in the number of candidates and in the average size of each candidate graph made the subgraph isomorphism time take more than the 8-hour limit. Moreover, gCode was failing for graphs larger than 200 nodes with error messages indicating the implementation was unable to handle signatures of the produced sizes. Trend-wise, the "simple" algorithms (exhaustive enumeration of paths) were the clear winners, with the other algorithms following in a similar order as in the indexing time case. The exception here is gCode, whose convoluted signature generation algorithm resulted in a larger execution time than even the frequent mining algorithms. In conclusion, the order of the algorithms from the fastest to the slowest is: (Grapes, GGSX) $<$ CT-Index $<$ (Tree+Δ, gIndex) $<$ gCode.

Last, as far as the algorithms' filtering power is concerned, Figure 2(d) shows an interesting trend: for all algorithms, the false positive ratio increases initially with the number of nodes, but then decreases again after some point: This "knee" appears around the 100-node mark for CT-Index and Tree+Δ, around the 200-node mark for gIndex and gCode, on the 500-node mark for GGSX, while Grapes doesn't reach its turning point in the x-axis range depicted in the figure.

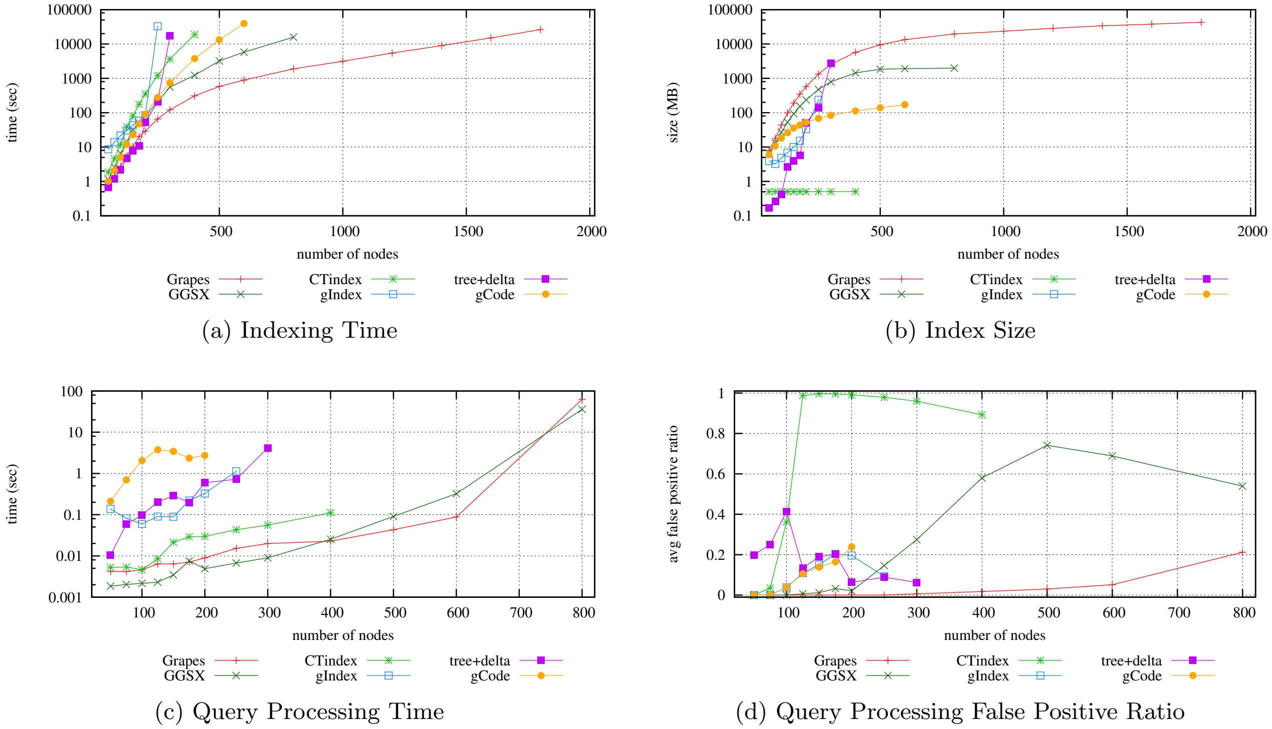
5.2.2 *Density*

Next, we vary the density of the graphs in the dataset. Specifically, we used the following density values: 0.005, 0.006, 0.007, 0.008, 0.009, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3.

Figures 3(a) and 3(b) depict the indexing time and size results, while Figures 3(c) and 3(d) show the query processing results. Apart from Grapes and GGSX, no other algorithm could produce an index for density values above 0.1 within the 8-hour limit, and Grapes was the only one capable of dealing with densities above 0.2. Please note that, as aforementioned, with a fixed number of graph nodes, increasing density results in a proportional increase in the number of graph edges (see equation (1)). We would thus expect to see similar behavior for both indexing and query processing as in the previous case, only with a less dramatic effect as the dependency is now proportional, not quadratic, and this is exactly what can be seen in these figures.

The astute reader will note that although GGSX and Grapes did produce an index for density values up to 0.3, the x-axis in Figures 3(c) and 3(d) does not extend beyond the 0.1 point. This was due to the fact that, when increasing densities, these algorithms didn't manage to produce results for densities above 0.1 within the 8-hour limit. To this end, we also show per-query-size query processing time results in Figure 4. An interesting observation stemming from this figure is that, for density values up to 0.1, the exhaustive enumeration approaches are rather insensitive to the size of the queries, whereas the frequent mining approaches show a small but noticeable increase in their query processing times. Grapes was the only algorithm capable of producing some query results for densities above 0.1. Moreover, we can see how the increase in query processing time becomes more abrupt the larger the query size, with Grapes producing results within the 8-hour limit for density values up to only 0.2 for 16-edge queries, and only up to 0.1 for 32-edge queries.

5.2.3 *Number of distinct node labels*



(a) Indexing Time

(b) Index Size

(c) Query Processing Time

(d) Query Processing False Positive Ratio

Figure 2: Indexing (a, b) and query processing (c, d) performance results for varying number of nodes

Varying the number of nodes per graph or the average graph density, resulted in larger graphs with more nodes and more edges. However, other than the graph size, the number of distinct labels in the dataset can also affect the indexing and query processing performance of the algorithms. More specifically, the *alphabet* of edges produced by GraphGen grows quadratically to the number of distinct labels. With the rest of the parameters constant, a larger alphabet translates to less overlap in the edges across graphs of any given dataset, inadvertently affecting approaches based on frequent fragment mining. Thus, we performed an evaluation ranging the number of distinct labels from 10 to 80.

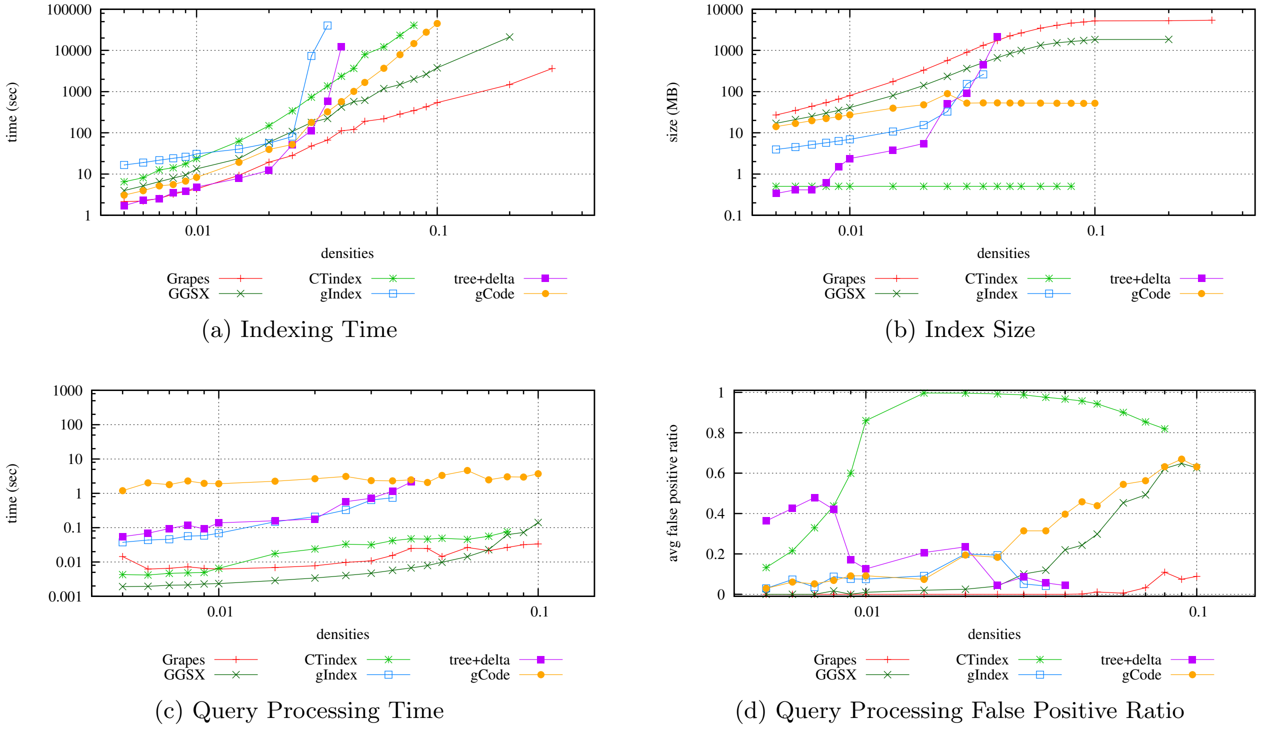
Figures 5(a) and 5(b) present the indexing time and index size for all algorithms. True to our above intuition, the indexing time of approaches using exhaustive enumeration is relatively unaffected by the increase in the number of distinct labels. On the other hand, the two frequent-mining algorithms are definitely affected, albeit curiously in completely opposite ways; whereas gIndex's indexing time increases with more distinct labels, that of Tree+Δ's decreases. We attribute this discrepancy in the different heuristics implemented by each algorithm, the fact that the former mines graphs while the latter mines trees and the discriminative ratio set by the algorithms. Also note that tree canonical labels are less computationally expensive to be produced than graph canonical labels and require less space for storage. Also notably, the frequent mining techniques could not construct an index within the 8-hour limit for the case of 10 distinct labels. We speculate the following: these mining techniques start from small features of size 1 (1 edge), that are expanded by one edge at every stage. If all features are found to be frequent because of the small number of labels, then they all need to be expanded in the next step, leading to an exponential increase of combinations to be considered.

Figures 5(c) and 5(d) present the query processing time and false positive ratio results, averaged over all different query sizes. We can see that the processing time of all algorithms seems to improve with more distinct labels, with the sole exception of gIndex. Similary, the filtering power of the algorithms also improves with more distinct labels, with the exception of Tree+Δ and CT-Index. Intuitively, the more the distinct labels the less repetition there is of distinct edges (i.e., pairs of labels) from the *alphabet*, so the number of false positives is expected to decrease. CT-Index again has the worst filtering power of all contenders, since its fixed-size hash-based fingerprints seem to suffer from "collisions" (considerably different graphs producing very similar fingerprints); however, what it loses in filtering power it gains in processing time, with the simplicity of its hash-based approach and its tweaked verification algorithm. Again, the general pattern regarding query processing time is: (Grapes, GGSX) < CT-Index < (Tree+Δ, gIndex) < gCode.

5.2.4 Number of graphs in the dataset

Last, we vary the number of graphs in the dataset. This parameter takes on values of 1000, 2500, 5000, 7500, 10000, 25000, 50000, 100000 and 500000 graphs. We would expect all performance metrics (indexing time, index size, processing time) to scale linearly to the number of graphs, as the latter does not affect in any way the complexity or size of individual graphs in the dataset. Along the same lines, we would expect the query processing false positive ratio to be relatively unaffected, for the exact same reason. These intuitions are indeed verified by the results depicted in figure 6, where these tendencies are rather prominent and clear.

Although the increase in performance metrics is linear



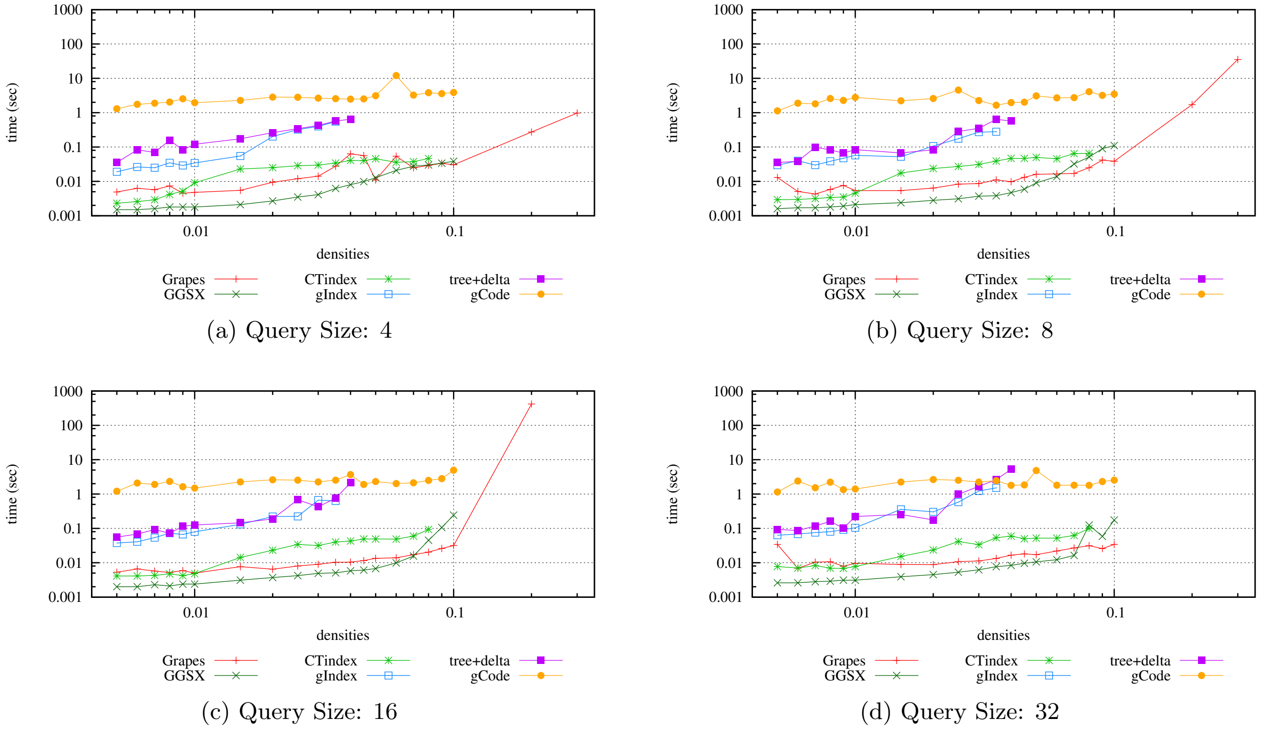
(a) Indexing Time

(b) Index Size

(c) Query Processing Time

(d) Query Processing False Positive Ratio

Figure 3: Indexing (a, b) and query processing (c, d) performance results for varying density values



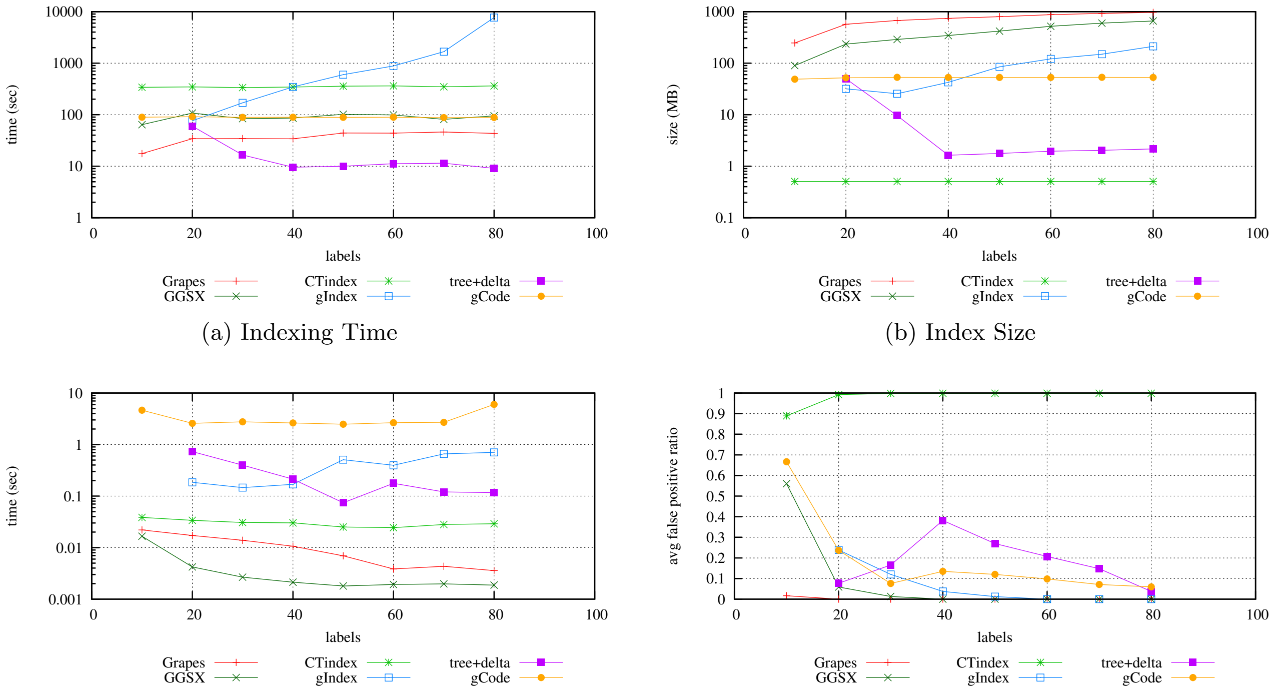
(a) Query Size: 4

(b) Query Size: 8

(c) Query Size: 16

(d) Query Size: 32

Figure 4: Query processing times for individual query graph sizes and varying density values



(a) Indexing Time

(b) Index Size

(c) Query Processing Time

(d) Query Processing False Positive Ratio

Figure 5: Indexing and query processing performance results for varying number of distinct labels

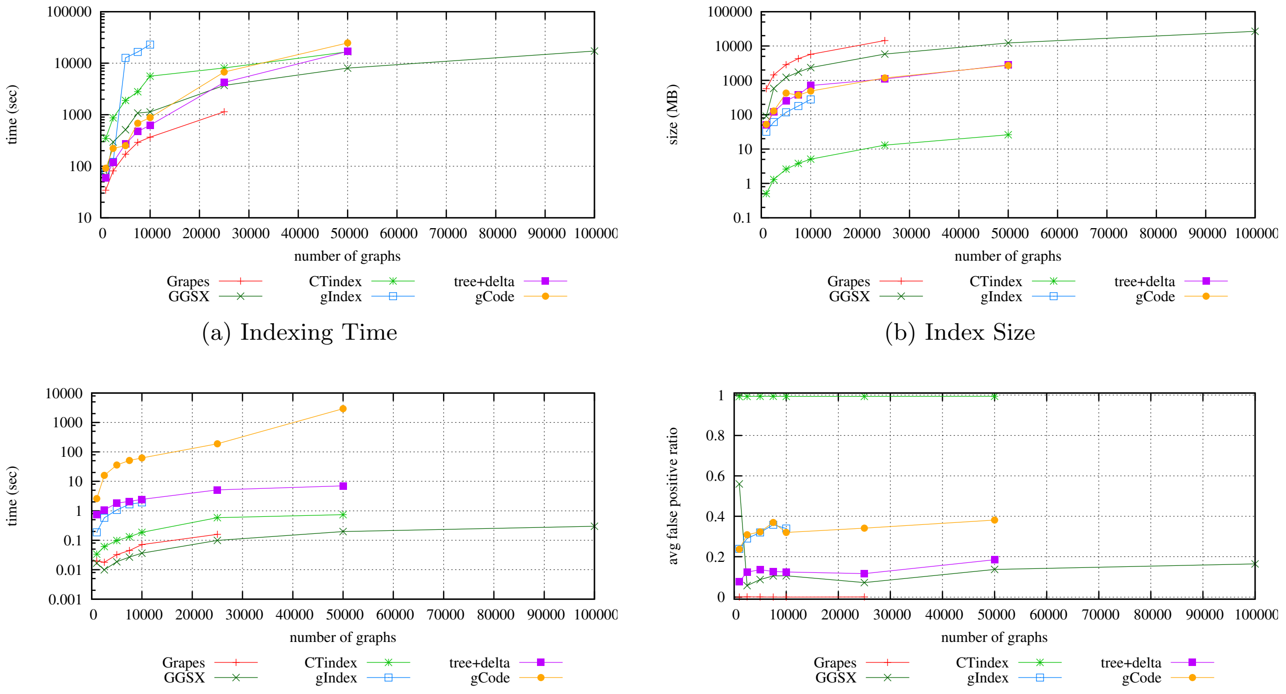
to the number of graphs, we can see that all algorithms other than GGSX didn't manage to produce an index for very large numbers of graphs. For gCode, CT-Index, and Tree+Δ, this was a case of the indexing process taking more than 8 hours for datasets with more than 50,000 graphs (actually the breaking points of individual algorithms were spread in the 50,000-100,000 node range, but we decided to forego adding individual data points for each algorithm for the sake of presentation consistency). gIndex also failed to produce indices due to excessive indexing time; moreover, its average time was much higher than that of other algorithms (an artefact of frequent mining and graph features) and thus exceeded the 8-hour limit much earlier than the rest (around the 10,000-node mark). Furthermore, this is the only case in our experiments where Grapes didn't manage to complete all indexing chores; the reason for this was excessive memory usage (its index size curve alludes to this), leading to thrashing even in our 128GB RAM host. However, for the cases where it managed to produce an index, its query processing performance was on par with that of GGSX and considerably better than the rest. Last, we can see again the same paradox as before for CT-Index; although its filtering power was the worst by a large margin, its query processing time was second only to Grapes and GGSX, due to the speed of using hash-based fingerprints and implementing a smarter subgraph isomorphism algorithm. Once again, the general pattern regarding query processing time is: (Grapes, GGSX) < CT-Index < (Tree+Δ, gIndex) < gCode.

6. LESSONS LEARNED & CONCLUSIONS

Effect of key dataset/workload characteristics. Our findings are summarized below:

- As indicated by equation (1), a linear increase in the *number of nodes* results in a quadratic increase in the number of edges; along the same lines, given a constant number of nodes, the number of edges increases linearly to the *graph density*. As the number of features is superlinear to the size of a graph, the increase of the above two factors leads to a detrimental increase in the indexing time, with the frequent mining techniques being more severely affected.
- The *number of graphs* increases the overall complexity only linearly (albeit the frequent mining techniques are more sensitive because more features have to be located across more graphs).
- The increase in the *number of distinct labels* leads to an easier dataset to index and an easier query workload to process, as it results in fewer occurrences of any given feature and thus a decrease in the false positive ratio of the various algorithms. Even relatively small changes in this characteristic affected drastically the performance of some of the algorithms.
- The *size of query graphs* affects all methods, even more so when the datasets consist of dense graphs. This effect is more pronounced for frequent mining techniques, even for moderately dense or even sparse graphs.

Sancta Simplicitas. Our findings give rise to the following adage: *"Keep It Simple and Smart"*. The general tendency is that, the simpler the feature structure and extraction process, the faster the indexing and query processing algorithm. Although seemingly counterintuitive, this conclusion is easily justifiable. Graphs are indeed more expressive than trees, which are in turn more expressive than paths, and thus a graph-based index would have a higher filtering power and lower processing time than a tree-based index etc.



(a) Indexing Time

(b) Index Size

(c) Query Processing Time

(d) Query Processing False Positive Ratio

Figure 6: Indexing and query processing performance results for varying number of graphs in the dataset

However, *the number of subgraphs of size n in a graph is significantly larger than the number of trees of size n, and the trees of size n is significantly larger than the number of paths of size n.* Due to this, indexes utilizing more features with complex structures are forced to maintain only a subset of them (frequent mining techniques) or apply some compression upon them (CT-index), and thus the expressive power gained by the more complex features is offset by the decrease in coverage and/or the introduction of yet more false positives in the filtering stage. Moreover, the more complex features also translate to higher indexing times and similarly higher filtering times. On the other hand, algorithms with simpler feature structures, resorting to exhaustive enumeration during their index construction, enjoy low indexing and filtering times, at the expense of considerably larger indexes.

Choosing the right index method for user needs. The various solutions tend to optimize different aspects of their operation. Answering which algorithm is best fit for any given case requires choosing an optimization criterion.

If *index size* is of importance, algorithms utilizing fixed-width encodings (CT-Index, gCode) should be chosen first; this is especially true as the size and complexity of the input grows. Frequent mining algorithms (gIndex, Tree+Δ) may be competitive for small/sparse datasets, but they quickly lose their edge as the datasets grow. Last, techniques using exhaustive enumeration and no encoding of features (Grapes, GGSX) have by far the largest indexes in size. This is particularly important if the index is to reside in main memory, as is usually desirable in most realistic use cases.

For the lowest *indexing time*, one should first look at techniques exhaustively enumerating their features (Grapes, GGSX, gCode, CT-Index), with approaches utilizing simpler features (paths; i.e., Grapes, GGSX) being considerably faster compared to those using more complex features (trees, cycles; i.e., CT-Index) and/or encoding (i.e., CT-Index, gCode); again, frequent mining approaches (gIndex, Tree+Δ) are competitive only for small/sparse datasets, but their indexing times grow very high very fast.

For *query processing time*, again the approaches using exhaustive enumeration (Grapes, GGSX, CT-Index) are the clear winners, with those indexing simple features (paths; i.e., Grapes, GGSX) having the edge over those with more complex features (trees, cycles; i.e., CT-Index). Frequent mining approaches (gIndex, Tree+Δ) are usually an order of magnitude slower than that. GCode here is the odd one out, as its encoding scheme seems to dominate the query processing time, hence it appears to be the slowest.

From a *scalability* point of view interesting trends evolve as the input grows larger and/or denser. For example, gCode, usually by far the slowest of the lot in query processing time, wins over the initially much faster frequent mining approaches as the dataset and graphs grow in size and density, as it exhibits a much better scaling. Conversely, Grapes, usually a very fast algorithm, fails to produce an index for certain very large datasets, and is routinely outscaled by GGSX. Notably, when the number of graphs goes beyond a few thousand, Grapes is also outscaled by Tree+Δ, gCode and CT-Index, as the additional location information in Grapes' index causes it not to fit in main memory.

Scalability limits. When dataset size (in number of nodes or density per graph and/or in number of dataset graphs) grows very large, *none* of the above methods can cope. Specifically, our results show that no method can scale beyond graph datasets with 1000 graphs, with each graph having 800 nodes, of medium (0.025) density. Reducing the average number of nodes per graph to 200 allows

GGSX to scale up to 100,000 graphs. At larger scales, one should either (i) rethink anew indexing methods, (ii) adopt an index-less approach (e.g., [16]), or (iii) turn to algorithms providing approximate answers.

Finally, let us compare our conclusions against those reached by the iGraph study. For the smaller datasets that iGraph studied and for algorithms common in iGraph and our paper, our results are in agreement. Specifically, the results for the index construction of gCode, gIndex and Tree+Δ show the same relative order and trends. Furthermore, our query processing results also coincide.

However, our work systematically studied the algorithms as they depend on key workload and dataset characteristics and further stress-tested them using 4 real and many synthetic datasets. This revealed the all new insights outlined above. Furthermore, with respect to the common algorithms with iGraph, gCode, although can be up to orders of magnitude worse than gIndex and Tree+Δ at smaller scales, it can outscale both gIndex and Tree+Δ, as density increases and can outscale gIndex and match the scalability of Tree+Δ as the number of dataset graphs increases. More importantly, in contrast to iGraph's conclusion, our study reveals 2 methods, GGSX and Grapes, one of which is always the clear winner for query processing time and scalability!

Acknowledgements. This research has been partially supported by the European Social Fund (ESF) and Greek national funds through the Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through ESF.

7. REFERENCES

[1] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic acids research*, 28(1):235–242, 2000.

[2] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, pages 195–203. 2010.

[3] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. VLDB*, pages 926–937, 2007.

[4] J. Cheng, Y. Ke, and W. Ng. GraphGen. `http://www.cse.ust.hk/graphgen/`.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proc. ACM SIGMOD*, 857-872, 2007.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[7] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha. Sing: Subgraph search in non-homogeneous graphs. *BMC Bioinformatics*, 11(1):96, 2010.

[8] Facebook Graph API. `https://developers.facebook.com/docs/graph-api`.

[9] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures. *PloS One*, 8(10):e76911, 2013.

[10] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: a framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[11] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Proc. IEEE ICDE*, pages 38–38, 2006.

[12] Y. He, F. Lin, P. R. Chipman, C. M. Bator, T. S. Baker, M. Shoham, R. J. Kuhn, M. E. Medof, and M. G. Rossmann. Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. *Proc. National Academy of Sciences of the United States of America*, 99:10325–10329, 2002.

[13] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. IEEE ICDE*, pages 1115–1126, 2011.

[14] National Cancer Institute - DTP AIDS antiviral screen dataset. `http://dtp.nci.nih.gov/docs/aids/aids_data.html`.

[15] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[17] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proc. IEEE ICDE*, pages 963–972, 2008.

[18] C. Vehlow, H. Stehr, M. Winkelmann, J. M. Duarte, L. Petzold, J. Dinse, and M. Lappe. CMView: Interactive contact map visualization and analysis. *Bioinformatics*, 27:1573–1577, 2011.

[19] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. IEEE ICDE*, pages 976–985, 2007.

[20] Y. Xie and P. Yu. CP-Index: on the efficient indexing of large graphs. In *Proc. ACM CIKM*, pages 1795–1804, 2011.

[21] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. ACM SIGMOD*, pages 335–346, 2004.

[22] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM TODS*, 31(4):1418–1453, 2006.

[23] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDBJ*, 22(2):229–252, 2013.

[24] D. Yuan, P. Mitra, and C. L. Giles. Mining and Indexing Graphs for Supergraph Search. *PVLDB*, 6(10):829–840, 2013.

[25] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *Proc. IEEE ICDE*, pages 966–975, 2007.

[26] S. Zhang, J. Yang, and W. Jin. SAPPER: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.

[27] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *Proc. VLDB*, pages 938–949, 2007.

[28] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. ACM EDBT*, pages 181–192, 2008.