Enhancing Graph Database Indexing by Suffix Tree Structure

Vincenzo Bonnici[1], Alfredo Ferro[1], Rosalba Giugno[1], Alfredo Pulvirenti[1], and Dennis Shasha[2]

[1] Dipartimento di Matematica ed Informatica, Università di Catania, Catania, Italy
vincenzo.bonnici@hotmail.it, ferro@dmi.unict.it, giugno@dmi.unict.it, apulvirenti@dmi.unict.it

[2] Courant Institute of Mathematical Sciences, New York University, New York, USA
shasha@cs.nyu.edu

Abstract. Biomedical and chemical databases are large and rapidly growing in size. Graphs naturally model such kinds of data. To fully exploit the wealth of information in these graph databases, scientists require systems that search for all occurrences of a query graph. To deal efficiently with graph searching, advanced methods for indexing, representation and matching of graphs have been proposed.

This paper presents GraphGrepSX. The system implements efficient graph searching algorithms together with an advanced filtering technique.

GraphGrepSX is compared with SING, GraphFind, CTree and GCoding. Experiments show that GraphGrepSX outperforms the compared systems on a very large collection of molecular data. In particular, it reduces the size and the time for the construction of large database index and outperforms the most popular systems.

Keywords: subgraph isomorphism, graph database search, indexing, suffix tree, molecular database.

1 Introduction and Related Work

Application domains such as bioinformatics and cheminformatics represent data as graphs where nodes are basic elements (i.e. proteins, atoms, etc.) and edges model relations among them. In these domains, graph searching plays a key role. For example, in computational biology locating subgraphs matching a specific topology is useful to find motifs of networks that may have functional relevance. In drug discovery, the main task is to find novel bioactive molecules, i.e., chemical compounds that, for example, protect human cells against a virus. One way to support the solution of this task is to analyze a database of known and tested molecules with the aim of building a classifier which predicts whether a novel molecule will be active or not. Future chemical tests can focus on the most promising candidates (see Fig. 1).

The graph searching problem can be formalized as follows. Given a database of graphs $D = \{G_1, G_2, \ldots, G_n\}$ (e.g. collection of molecules, etc.) and a query

T.M.H. Dijkstra et al. (Eds.): PRIB 2010, LNBI 6282, pp. 195–203, 2010.

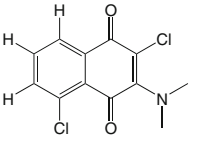
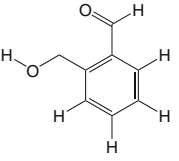
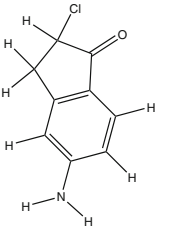
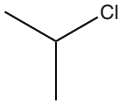
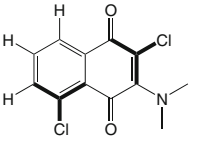
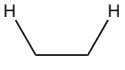
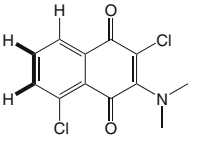
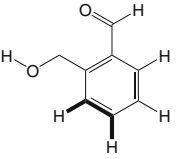
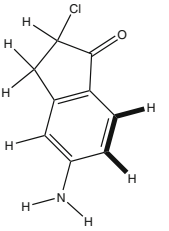
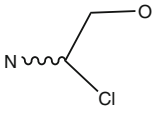
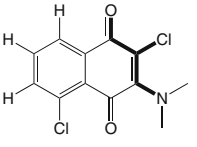
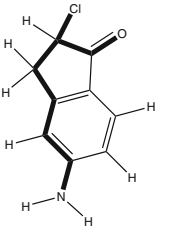
Graph Database	G_1	G_2	G_3
			
Exact Query	Number of Occurrences in Graphs		
			
Q_1	2		
			
Q_2	2	3	1
Approximate Query	Graphs Containing the Query		
			
Q_3	1		1

Fig. 1. Querying a database of graphs. Graphs represent molecules. During the match process, edge information is ignored. Query occurrences are shown in bold. For Q_2, since query matches overlap, only one occurrence in each molecule is depicted. The number of occurrences is also given. Molecular descriptions include hydrogen atoms for search accuracy. In a context where hydrogen atoms are not considered, query Q_2 is present 11 times in G_1, 6 in G_2 and 10 in G_3. The approximate query specifies any path of an unspecified length between atoms C and N. Approximate queries may also contain atoms with unknown label (they match any atom). In this paper we do not exploit approximate queries since the compared systems do not deal with such scenarios.

graph Q (e.g pattern), find all graphs in D containing Q as a subgraph. Ideally, all occurrences of Q in those graphs should be detected. Since most of these problems involve solutions of the graph isomorphism problem, an efficient exact solution cannot exist. In order to make searching time acceptable, research efforts have tried to reduce the search space by filtering out the graphs that do not contain the query. After candidate graphs have been selected, an exhaustive search on these graphs must be performed. This step is implemented either by traditional (sub)graph-to-graph matching techniques [7,3] or by an implementation that extends the SQL algebra [8].

For a database of graphs a filter limits the search to only possible candidate graphs. The idea is to extract structural features of graphs and store them in a global index. When a query graph is presented, its own structural features are extracted and compared with the features stored in the index to check compatibility [4,12,10]. Most existing systems use subgraphs (paths [4,12,5,6], trees [15,1], graphs [14]) of small size (typically not larger than 10 nodes).

In order to apply such systems on large graphs, SING [5] tool stores the starting node of each feature. This is done to capture the notion of features that are branches of trees. The matching algorithm is also modified to start the search on a selected node whose label is present in the query and not from a random one.

However, even though small subgraphs are used, the size of the index and its time construction may be high. Therefore, high-support/high-confidence mining rules are used to index only frequent and non-redundant subgraphs (i.e. a subgraph is redundant when its presence in a graph can be predicted by the presence of its subgraphs) [15,1,14]. More precisely, gIndex [14] stores, in a compact tree, all discriminat and frequent subgraphs. FGIndex [14] uses two indexes: the first one is stored in main memory, the second one is on disk. In order to assign a feature to an index, the query is performed on the main-memory-resident index. If it doesn't return any result, this index is used to identify the blocks of the secondary memory index to be loaded. GraphFind [6] uses the low-support data mining technique (Min-Hashing [2]) to reduce the index size. It is shown that such a mining technique can be successfully applied to enhance other systems such as gIndex. The above tools all require an effective but expensive data mining step.

Several indexes are based on capturing other discrimant characteristics of the graph. CTree [9] applies a graph closure to the database graphs, aligning vertices and edges using a fast approximate algorithm called neighbor biased mapping. It stores an ouput synthesized graph in a R-tree-like data structure. During the filtering phase an approximate match is executed on the closure graphs of the tree in a top down approach. Ctree spends much of its time in this matching phase. GCoding [16] uses graph signatures made by concatenating vertex signatures. A vertex signature is built from its label, neighbor labels and higher eigenvalues of the adjacency matrix of a tree representing all length n paths starting from a random node. The signature graph set is inserted into a B-tree-like structure index. In this way GCoding allows a compact representation of the indexes,

but the cost of the eigenvalue computation and the high number of produced candidates reduce the method's efficiency.

In this paper, we propose GraphGrepSX, a novel approach inspired by the GraphGrep ([12,6]) system. GraphGrepSX uses paths of bounded length as features stored in a Suffix tree [13] structure. By exploiting path prefix sharing, the algorithm reduces redundancies and achieves a more compact representation of the index. This approach is particularly effective on graphs with a small label space (e.g. chemical molecules). In such a case, the same partial combination of labels could be present several times in the features of different graphs. Although such a representation is very natural and simple, GraphGrepSX is able to speed up both the index construction and the filtering phases. Moreover since it has a low index loading time, it is suitable for searching on dynamic datasets. To evaluate the performance of GraphGrepSX, we compare it with the most prominent graph search systems.

2 GraphGrepSX

GraphGrepSX uses paths of bounded length as features stored in a Suffix tree [13] structure. In what follows we describe the phases of the method.

2.1 Preprocessing Phase

The preprocessing phase extracts the features from the graph database and inserts them into the global index. Every node v_j of a graph G_i of the database is visited by a depth-first search. During this phase, all the paths of length up to and equal to l_p are extracted. Each path is represented by the labels of its nodes. Each path $(v_1, v_2, ..., v_{l_p})$ is then mapped into its corresponding sequence of labels $(l_1, l_2, ..., l_{l_p})$. All the subpaths $\{(v_i, ..., v_j)$ for $1 \leq i \leq j \leq l_p\}$ of a path $(v_1, v_2, ..., v_{l_p})$ are features which will be included in the global index also.

For each extracted path, we keep track also of the number of time it appears in every single graph of the database. All these features are then stored in a Suffix tree. Each node of the tree represents a path obtained during the depth-first search traversal. The path can be reconstructed using its ancestors in the Suffix tree. Each node of the tree also stores the list of graphs containing it together with the number of times the path appears in each graph. The construction and update of the Suffix tree are done during the depth-first search of each graph. GraphGrepSX implements the Suffix tree as an N-ary tree in which the children of a node are represented by a linked list. The list of the occurrences of the features of the graphs is stored in a binary tree indexed by a unique graph id. The Suffix tree and the occurrences list are also stored in an archive file using a compact representation.

The worst case cost to search the child of a given node in the Suffix-Tree is $|l_s|$, where $|l_s|$ is the maximum number of distinct labels in the graph G_i, because the child list is represented as a linked list. Since the list of the feaures occurrences is stored in a binary tree, the cost to update a value is $\log |D|$. The cost of each

depth-first visit is $n_i m_i^{l_p}$, where n_i and m_i are respectively the number of the nodes and the maximum valence (degree) of the nodes in the graph G_i. The total cost to build the database index is $O(\sum_i^{|D|}(n_i m_i^{l_p}|l_s|\log|D|))$.

2.2 Filtering and Matching Phases

Given a query graph q, the filtering phase tries to filter out those graphs that cannot match the query graph. This phase is done in two steps. In the first step, the query graph q is processed and its features are extracted and stored in a Suffix Tree. In contrast to the preprocessing phase, here we consider only the maximal paths visited during the depth-first search of the query graph q. A path is considered maximal either if its length is equal to l_p or the path has length less than l_p but cannot be further extended, because the depth-first search can not continue. The nodes of the Suffix tree storing the end-point of a maximal path are marked. Only the occurrences of the maximal paths are stored in the marked nodes of the index.

In the second step the pruning of the candidate graphs of the database is performed by matching the query suffix tree against the suffix tree of the global index. Each marked node of the query tree representing a labeled path $(l_1, l_2, ..., l_n)$ is searched in the Suffix tree of the global index. Those graphs which either do not contain such a path or have such path with an occurrence number less than the occurrence number of the query are discarded. Those that remain represent the candidate set of possibly matching graphs.

The tesing of each candidate graph uses the VF2 [3] library for exhaustive subgraph isomorphism. VF2 is a combinatorial search algorithm which induces a search tree by branching states. It uses a set of topological feasibility rules and a semantic feasibility rule, based on label comparison, to prune the search space. At each state if any rule fails, the algorithm backtracks to the previous step of the match computation.

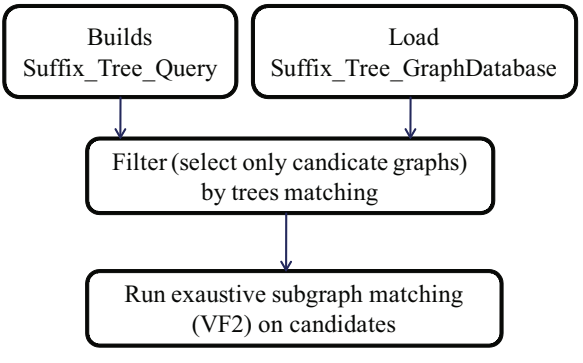


Fig. 2. Figure shows the filtering phase and the candidates verification phase made by GraphGrepSX

Let T_q to be the query Suffix-Tree and $|T_q|$ the number of nodes inside it.The building cost is $O(n_q m_q^{l_p} |l_s|)$ where n_q is the number of nodes in the tree, m_q is the maximum degree of a node and $|l_s|$ is the maximum number of distinct labels in the query graph. The cost of the pruning step is given by matching time of the query Suffix-Tree against the database index, i.e. $O(|T_q||l_s|)$, plus the average time of the occurrences verification, i.e. $O(|D| \log |D|)$. Therefore the total time is $O(|T_q||l_s||D| \log |D|)$ and, let C be the set of candidates graphs, the cost for each candidate C_i verification is $O(|V[C_i]|!|V[C_i]|)$.

2.3 Experimental Results and Biological Application

GraphGrepSX was implemented in C++ and compiled with the GNU compiler 3.3. In order to evaluate the performance of the proposed approach, it has been compared with the main graph search systems: GraphFind [6], CTree [9], GCoding [16], and SING [5]. Notice that, in what follows, we refer to GraphFind as GraphGrep since we do not use the mining step in the index construction phase. Moreover we do not report comparisons with gIndex [14] since GraphFind outperforms it without using mining. The system has been tested using the Antiviral Screen Dataset [11]. The AIDS database contains the topological structures of 42,000 chemical compounds that have been tested for evidence of anti-HIV activity. It contains sparse graphs having from 20 to 270 nodes. The entire set was divided into three subsets of sizes 8000, 24000 and 42000 respectively. Queries were randomly extracted from the AIDS database selecting a vertex v from a graph of the database and proceeding with a breath-first visit. This process generate groups of 100 queries from each database having a number of edges with 4, 8, 16, and 32 edges. Table 1 shows the index building time for each subset.

Table 1. Index building time (sec)

DB dim.	GraphGrepSX	GraphGrep	CTree	GCoding	SING
8000	16.51	550.4	8.21	632.21	22
24000	38	10399.39	25.34	1956.36	66
42000	66	45600.49	42.42	2944.8	108

GraphGrepSX and CTree yield comparable index construction time and outperform the other approaches. The sizes of the generated indexes are shown in table 2. Thanks to the compactness of its suffix tree structure, GraphGrephSX reduces the redundancy of the index. Therefore GraphGrepSX outperforms the latest graph matching tools. It outperforms SING if used with dynamically changing datasets.

In what follows we show the execution times of the filtering and verification phases. These results report tests made on the entire 42000 AIDS molecular dataset grouped by queries dimension. In table 3 we report the filtering time.

Table 2. Indexes size (Kb)

DB dim.	GraphGrepSX	GraphGrep	CTree	GCoding	SING
8000	3684	293992	13884	6687	8445
24000	11020	928912	41372	20088	22279
42000	18668	1577012	70208	30651	42830

Table 3. Filtering time (sec)

Query dim.	GraphGrepSX	GraphGrep	CTree	GCoding	SING
4	0.05	0.012	1.34	0.0042	0.51
8	0.05	0.006	1.57	0.01	0.76
16	0.041	0.005	1.51	0.026	0.17
32	0.04	0.014	1.01	0.059	0.071

Table 4. Query time (sec)

Query dim.	GraphGrepSX	GraphGrep	CTree	GCoding	SING
4	14.9	15.41	13.27	23.61	12.4
8	17.5	7.1	44.24	15.79	15.24
16	2.08	12.78	51.07	5.39	0.798
32	1.07	3.56	50.91	1.25	0.136

Table 4 shows the total time. The number of generated candidates after the filtering step is shown in table 5. CTree and GCoding generate smaller candidates sets. This is due to the fact that such indexes are able to capture the structure of the graphs. Unfortunately, they require more execution time because of the approximate match on the closure graphs and the mining operations during the filtering step.

Table 5. Number of generated candidates

Query dim.	GraphGrepSX	GraphGrep	CTree	GCoding	SING
4	26865	29196	16704	16188	23170
8	21337	13920	5840	8567	14012
16	1629	7053	289	1648	214
32	142	3193	3	142	4

Table 6. Total time time (sec)

Query dim.	GraphGrepSX	SING
4	15.89	23.39
8	18.56	26.42
16	3.07	1f1.45
32	2.04	10.68

GraphGrepSX and GraphGrep uses the same matching algorithm, but the first generates a smaller number of candidates by applying a redundant check deletion phase.

In Table 6 we report the total time needed by GraphGrepSX and SING to execute a single query. SING has an overhead of 10.5 seconds to load the index. Whereas GraphGrepSX needs less than one second (0.93 seconds) to load the index.

3 Conclusion

Indexing paths instead of subgraphs may result in more preprocessing time and indexing space. However, paths require less filtering and querying time. Results show that a further improvement on path-index base system is achieved by making use of Suffix Trees. GraphGrephSX reduces the size and time needed for the construction of large database index compared to the most prominent graph querying systems. Furthermore, GraphGrephSX outperforms all compared systems when the index structure needs to be rebuilt. It can be considered to be a good compromise between preprocessing time and querying time.

References

1. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 857–872 (2007)
2. Cohen, E., Datar, M., Fujiwara, S., Gionis, A., Indyk, P., Motwani, R., Ullman, J.D., Yang, C.: Finding interesting associations without support pruning. IEEE Transactions on Knowledge and Data Engineering 13(1), 64–78 (2001)
3. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10), 1367–1372 (2004)
4. Daylight chemical information systems, `http://www.daylight.com/`
5. Di Natale, R., Ferro, A., Giugno, R., Mongiovi, M., Pulvirenti, A., Shasha, D.: Sing: Subgraph search in non-homogeneous graphs. BMC bioinformatics 11(1), 96 (2010)
6. Ferro, A., Giugno, R., Mongiovì, M., Pulvirenti, A., Skripin, D., Shasha, D.: Graphfind: enhancing graph searching by low support data mining techniques. BMC bioinformatics 9(suppl. 4), S10 (2008)

7. Frowns, http://frowns.sourceforge.net/
8. Giugno, R., Shasha, D.: Graphgrep: A fast and universal method for querying graphs. In: Proceeding of the International Conference in Pattern Recognition (ICPR), pp. 112–115 (2002)
9. He, H., Singh, A.K.: Closure-tree: An index structure for graph queries. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, p. 38 (2006)
10. Messmer, B.T., Bunke, H.: Subgraph isomorphism detection in polynominal time on preprocessed model graphs. In: Proceedings of Asian Conference on Computer Vision, pp. 373–382 (1995)
11. National Cancer Institute. U.S. National Institute of Health, http://www.cancer.gov/
12. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: Proceeding of the ACM Symposium on Principles of Database Systems (PODS), pp. 39–52 (2002)
13. Ukkonen, E.: Approximate string-matching over suffix trees. In: Combinatorial Pattern Matching, pp. 228–242. Springer, Heidelberg (1993)
14. Yan, X., Yu, P.S., Han, J.: Graph indexing based on discriminative frequent structure analysis. ACM Transactions on Database Systems 30(4), 960–993 (2005)
15. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: Proceedings of IEEE International Conference on Data Engineering, pp. 966–975 (2007)
16. Zou, L., Chen, L., Yu, J.X., Lu, Y.: A novel spectral coding in a large graph database. In: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, pp. 181–192. ACM, New York (2008)