GraphGrep: A Fast and Universal Method for Querying Graphs

Rosalba Giugno
Department of Mathematics and Computer Science
University of Catania
viale A. Doria 6, 95125 Catania, Italy
giugno@dmi.unict.it

Dennis Shasha
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York, NY 10012
shasha@cs.nyu.edu

Abstract

GraphGrep is an application-independent method for querying graphs, finding all the occurrences of a subgraph in a database of graphs. The interface to GraphGrep is a regular expression graph query language Glide that combines features from XPath and Smart. Glide incorporates both single node and variable-length wildcards. Our algorithm uses hash-based fingerprinting to represent the graphs in an abstract form and to filter the database. GraphGrep has been tested on databases of size up to 16,000 molecules and performs well in this entire range.

1. Introduction

Many applications in industry, science, and engineering share the same problem: given a subgraph, find its occurrences in a database of graphs. The increasing size of application databases requires efficient structure searching algorithms. Examples of such database and substructure searching methods can be found in computational chemistry [6],[14], vision [3], and web exchange data (XML)[7][1].

Finding occurrences of a subgraph in a set of graphs is known to be NP complete [5]. Although graph-to-graph matching algorithms [2], [13] can be used, efficiency considerations suggest the use of special techniques to reduce the search space and the time complexity.

There is an extensive literature on graph (or substructure) searching. For a review see [14], [12]. Most of the existing methods however, are designed for specific applications. For example several querying methods for semistructured databases, and in particular for XML databases, have been proposed ([7],[1],[11], [4], [12], [10]). These methods use different data models, query languages and indexing strategies. The data objects used in XML databases are viewed as rooted labeled graphs. Regular path expressions are used to address substructures in the database. Cycles are searched by evaluating recursion functions or by formulating complex queries. To avoid unnecessary traversals of the database during the evaluation of a path expression, indexing methods are introduced in [7] and [9].

Daylight[6] proposes a searching system for a database of molecular graphs. It finds all the molecules that contain, as a subgraph, at least one occurrence of the query. Daylight uses fingerprints consisting of bit vectors, where each position represents a small path. It also provides a graph expression language based on the Smiles [15] molecule representation to formulate queries.

Messmer and Bunke[8] propose an application independent method. The method indexes the graphs in a database and computes a graph isomorphism. Both indexing and matching are based on all possible permutations of the adjacent matrices of the graphs. This algorithm works extremely well on small graphs, but doesn't scale well to larger graphs or large databases of graphs.

In this article we present an application-independent method to perform *exact* subgraph queries in a database of graphs. Our system `GraphGrep` finds *all* the occurrences of a graph in a database of graphs. To formulate queries we introduce a graph query language which we term Glide: Graph LInear DEscription language. Glide descends from two query languages Xpath [1] for XML documents and Smart [6] for molecules. In Xpath, queries are expressed using complex path expressions where the filter and the matching conditions are included in the notation of the nodes. Glide uses graph expressions instead of path expressions. Smiles is a language designed to code molecules and Smart is a query language to discover components in a Smiles databases. Glide borrows the cycle notation from Smiles and generalizes it to any graph application.

2. GraphGrep description

`GraphGrep` assumes that the nodes of the database graphs have an identification number (*id-node*) and a label (*label-node*). Edges are undirected and unlabeled (for pur-

poses of this paper). We define an *id-path* of length n to be a list of n id-nodes with an edge between any two consecutive nodes. Similarly a *label-path* of length n is defined as a list of n label-nodes. For example in Fig. 1, (C,A) is a label path in graph g_1, and (3,1) is an id-path corresponding to it.

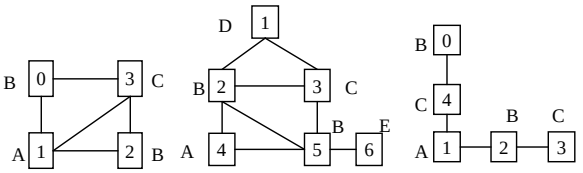


Figure 1. A database containing 3 graphs. The labels can be strings of arbitrary length.

The basic steps of `GraphGrep` are to: (1) build the database to represent the graphs as sets of paths (this step is done only once), (2) filter the database based on the submitted query to reduce the search space, and (3) perform exact matching. We discuss these steps in turn.

Database construction. For each graph and for each node, find all paths that start at this node and have length one (single node) up to a (small, e.g. 10) constant value l_p (l_p nodes). We use the same l_p for all graphs in the database. Because several paths may contain the same label sequence, we group the id-paths associated with the same label-path in a set. The "path-representation" of a graph is the set of label-paths in the graph, where each label-path has a set of id-paths (see Fig. 2a).

The keys of the hash table are the hash values of the label paths. Each row contains the number of id-paths associated with a key (hash value) in each graph. We will refer to the hash table as the fingerprint of the database (see Fig. 2b).

(a) **Path representation of graph g1**

A={(1)} AB={(1, 0), (1,2)} AC ={(1, 3)} ACBA={…}
ABCA={(1 ,0 ,3 ,1),(1, 2, 3, 1)} CB={(3,0),(3,2)} C={(3)}
CBAB={(3, 0, 1 ,2),(3 ,2 ,1 ,0)} B={(0),(2)} BA={(0,1),(2,1)}
BAB={(0,1,2), (2,1,0)} ABC ={(1, 3, 0), (1,3,2)} ACB={…}
ABCB={…} BC={…} BAC={…} BCB={…} CBA={…}
BABC={…}CBAC={…} CABC={…} CAB={(3,1,0), (3,1,2)}
BACB={…}BCBA={…}BCAB={…} BCA={…} CA={(3,1)}

Key	g_1	g_2	g_3
h(CA)	1	0	1
……			
h(ABCB)	2	2	0

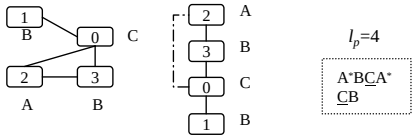
(b) **Fingerprint of database**

Figure 2. (a) The path representation of g_1 with $l_p = 4$. (b) The fingerprint of the database showing only part of rows.

Parsing a query graph. A query graph is given in the Glide language (see Sec. 3 and Fig. 3); it can be seen as a linear representation of a tree generated in a depth first search (DFS) traversal of the query graph (see Fig. 3). The query is parsed to build its fingerprint (hashed set of paths) and the branches in the depth-first tree are decomposed into sequences of overlapping label-paths, which we also call *patterns*, of length l_p or less (see Fig. 3).

These overlaps may appear in the following cases: (1) for consecutive label-paths, the last node of a pattern coincides with the first node of the next pattern (e.g. A/B/C/B/, with $l_p = 3$ is decomposed into two patterns: ABC and CB); (2) if a node has branches it is included in the first pattern of every branch (see node C in Fig. 3d); (3) The first node visited in a cycle appears twice: in the beginning of the first pattern of the cycle and at the end the last pattern of the cycle (the first and last pattern can be identical, as in Fig. 3d).

A%1/B/C%1/B/
(a) **Glide graph query q1**



(b) **Graph query q1** (c) **Depth first tree** (d) **Patterns**

Figure 3. (a) A query graph in Glide representation. (b) The graph query. (c) The depth first tree (to which Glide expression (a) corresponds to). (d) A set of patterns obtained with $l_p = 4$. In this example overlapping labels are marked with asterisks or underlining. Labels with the same mark represent the same node.

Filtering the database. To avoid visiting all the graphs in the database during a query the search space is reduced by discarding graphs that clearly do not contain any occurrences of the query. The remaining graphs *may* contain one or more subgraphs matching the query.

We filter the database by comparing the fingerprint of the query with the fingerprint of the database. A graph, for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query, is discarded. For example, for the graph query in Fig. 3 with $l_p = 4$, graphs g_2 and g_3 are filtered out because they do not contain the label-path ABCA.

Finding subgraphs matching with queries. After filtering, we look for all the matching subgraphs in the remaining graphs. We use the path representation of the graphs to look for occurrences of the query. Only the parts of each (candidate) graph whose id-path sets correspond to the patterns of the query are selected and compared with the query. Here is how. After the id-path sets are selected, we identify overlapping id-path lists and concatenate them (removing overlaps) to build a matching subgraph. For the overlapping case (1) and (2) a pair of lists is combined if the two lists contain the *same* id-node in the overlapping position. In the overlapping case (3), a list is removed if it does *not* contain the *same* id node in the overlapping positions; finally, lists are removed if the id-nodes which are *not* placed in overlapping positions are equal.

Example Let us consider the steps to match the query (Fig. 3) and the graph g_1 (Fig. 2).

1. Select the set of paths in g_1 matching the patterns of the query (with $l_p = 4$): ABCA= $\{(1,0,3,1), (1,2,3,1)\}$ CB= $\{(3,0),(3,2)\}$.
2. Combine any list l_1 from ABCA with any list l_2 of CB if the third id-node in l_1 is equal to the first id-node of l_2 and the first id-node in l_1 is equal to the fourth id-node of l_1: ABCACB= $\{((1,0,3,1),(3,0)), ((1,0,3,1),(3,2)), ((1,2,3,1),(3,0)), ((1,2,3,1),(3,2))\}$.
3. Remove lists from ABCACB if they contain equal id-nodes in non-overlapping positions (the positions in the each list not involved above). The two substructures in g_1 whose composition yields ABCACB are $((1,0,3,1),(3,2))$ and $((1,2,3,1),(3,0))$.

Graph queries with wildcards are treated by considering the parts of the query graph between wildcards as disconnected components. (For example, the disconnected components of the graph in Fig. 4 are the path AB and the single node D.) The matching algorithm described above is done for each component. A cartesian product between the sets that match each component constitutes the candidate matches. An entry in the cartesian product is a valid match if, in one graph that contains the entry, there is a path (of length equal to the wildcard's value) between nodes that are connected with wildcards. The paths in the candidate graph are checked with a DFS traversal of the graph. This step is optimized by maintaining the transitive closure matrices of the database graphs and searching in the candidate graph only if the wildcard's value is greater than or equal to the shortest path between the nodes.

Complexity. Here is a description of the worst case complexity for GraphGrep. Let $|D|$ be the number of graphs in a database D. Let n, e and m be the number of nodes, the number of edges and the maximum valence (degree) of the nodes in a database graph, respectively. The worst case complexity of building a path representation for the database is $\mathcal{O}(\sum_i^{|D|}(n_i m_i^{l_p}))$, whereas the memory cost is $\mathcal{O}(\sum_i^{|D|}(l_p n_i m_i^{l_p}))$. Given a query with n_q nodes, e_q edges and m_q maximum valence, finding its patterns takes $\mathcal{O}(n_q + e_q)$ time; building its fingerprint takes $\mathcal{O}(n_q m_q^{l_p})$. Filtering the database takes linear time in the size of the database. The matching algorithm depends on the number of query graph patterns p, that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the nodes in the query. The larger l_p, the smaller p, though this relationship is data-dependent. In general if $\tilde{n}$ is the maximum number of nodes having the same label, the worst case time complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p))$ with $|D_f|$ the size of the database after the filtering. For a query containing w pairs of nodes connected with wildcards the complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p + we_i))$.

3. Glide: a graph linear query language

The main idea of Glide (coming from Smiles) is to represent a graph and its branches in linear notation where each node is presented only once. Nodes are represented using their labels and they are separated using slashes; branches are grouped using nested parentheses '(' and ')' and cycles are broken by cutting an edge and labeling it with an integer. The vertices of the cut edge are represented by their labels followed by %, the integer and '/'. If the same node is a vertex of several cut edges the label of the node is followed by a list of % and integers.
For example the Glide representation of the graph in Fig. 1a is "A%1%2/B/C%2/B%1/", in Fig. 1b is "A%3/B%1%2%3/(E/)C%2/D%1/" and in Fig. 1c is "B/C/A/B/C/".

Unspecified components in a graph are described using wildcards '$*$','.','$+$' and '?'. The wildcards represent single nodes or paths. The wildcard '.' matches any single node; (2) '$*$' matches zero or more nodes; (3) '?'matches zero or one node; and (3) '$+$' matches one or more nodes (see Fig. 4).

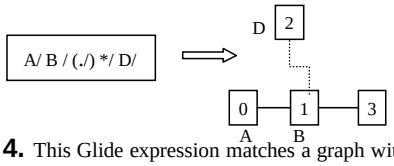
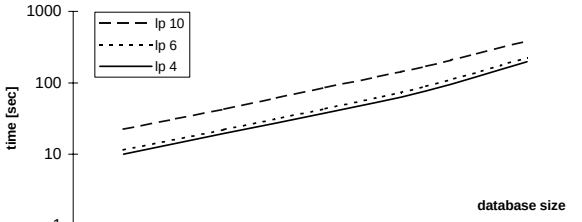


Figure 4. This Glide expression matches a graph with specific properties: (1) there exist a path between a node with label B and a node with label D; (2) there is an edge between the node with label B and a node with label A and (3) there is an edge between the node with label B and a node with any label.

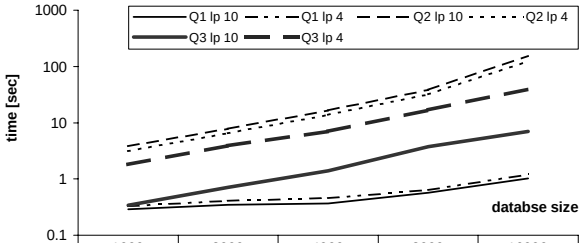
4 Results

To assess the practical efficiency of `GraphGrep` we performed a set of numerical experiments on NCI databases up to 16,000 molecules. The graphs in these databases have an average number of 20 nodes; several graphs have up to 270 nodes. We used an old Sun workstation equipped with the 650MHz Ultra80 processor.

In our experiments we varied query size (5, 13, to 66 nodes), database sizes (1,000 - 16,000 graphs), and l_p values (4, 6, and 10) (Fig. 5). The left diagram gives preprocessing times (on a logarithmic scale). The running time is exponential in l_p and is linear in the size of the database. The right diagram reports query times (also on a logarithmic scale) for the three different query lengths and four different l_p values (see Fig 5). Different values of l_p influence the running of the queries: for the $Q3$ query the matching



	1000	2000	4000	8000	16000
lp 10	22.38	42.81	86.01	170.4	386.06
lp 6	11.48	22.29	43.62	89.65	222.29
lp 4	10.04	19.53	38	76.98	196.47



	1000	2000	4000	8000	16000
Q1 lp 10	0.29	0.35	0.37	0.57	1.02
Q1 lp 4	0.33	0.41	0.46	0.64	1.2
Q2 lp 10	3.85	7.82	16.63	38.71	153.19
Q2 lp 4	3.15	6.54	13.71	31.74	124
Q3 lp 10	0.34	0.71	1.4	3.78	7.03
Q3 lp 4	1.8	3.9	7.02	16.98	40.03

Figure 5. The abscissa gives the size of the database and the ordinate the CPU time measured in seconds (both in logarithmic scale). The left diagram gives preprocessing times (database construction times) and the right diagram gives querying times. Query Q1 is "c%1/c/c/c/(c/c%1/)P/c%2/c/c/c/c/c%2/". Query Q2 is "c/(./c/)c/(*/S/)c/". Query Q3 is a molecule with 66 nodes, 72 undirected edges and its max valence per node is 6. For the Q1 and Q3 queries, 99% of the database was discarded, whereas for the Q2 query, 81% was discarded. In this experiment, changing the l_p value didn't change the effectiveness of filtering. For the 16,000 molecules database 640 subgraphs are found for Q1, 10,496 for Q2 and 564 for Q3.

algorithm performs better with $l_p = 10$ than with $l_p = 4$ (which is consistent with the time complexity analysis). In addition, in these examples we verify that the querying time is linear in the size of the database, and exponential in $p \times l_p$. Recall that p (the number of paths within size l_p that have to be tested) is proportional to the query size. As expected, p decreases substantially with larger l_p (e.g. for paths), but not always. Note that there is no exponential dependency on the data graph size.

5. Conclusion and Further Work

We have presented a search algorithm `GraphGrep` and a graph query language for database of graphs. We have shown that it performs well for small query graphs on large graph databases (in the thousands) and large size (270 nodes). We are extending `GraphGrep` to compute inexact subgraph matching. A software implementation is freely available at www.cs.nyu.edu/shasha/papers/graphgrep/.

References

[1] J. Clark and S. DeRose. *Xml Path Language (Xpath)*. http://www.w3.org/TR/xpath, 1999.

[2] L. Cordella, P. Foggia, C. Sansone, and M. Vento. An efficient algorithm for the inexact matching of arg graphs using a contextual transformational model. In *In proceedings of the 13th ICPR*, volume 3, pages 180–184. IEEE Computer Society Press, 1996.

[3] S. Dickinson, M. Pelillo, and R. Zabih. Introduction to the special section on graph algorithms in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10), October 2001.

[4] L. Galanis, E. Viglas, D. J. DeWitt, J. F. Naughton, and D. Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. Submit for pubblication, 2001.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[6] C. A. James, D. Weininger, and J. Delany. *Daylight theory manual-Daylight 4.71*. Daylight Chemical Information Systems, www.daylight.com, 2000.

[7] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. In *SIGMOD Record*, volume 26, pages 54–66, September 1997.

[8] B. T. Messmer and H. Bunke. Subgraph isomorphism detection in polynominal time on preprocessed model graphs. In *ACCV*, Lecture Notes in Computer Science, pages 383–392. Springer, 1996.

[9] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[10] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB Journal*, 1999.

[11] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.

[12] D. Suciu. An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29, 1998.

[13] J. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23:31–42, 1976.

[14] J. Wang, B. Shapiro, and D. Shasha. *Pattern Discovery in Biomolecular Data*. New York Oxford, oxford university press edition, 1999.

[15] D. Weininger. Smiles. introduction and encoding rules. *Journal Chemical Information in Computer Science*, 28(31), 1988.