Querying Subgraph Sets with G-Tries

Pedro Ribeiro Fernando Silva
CRACS & INESC-TEC, Faculdade de Ciencias, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{pribeiro, fds}@dcc.fc.up.pt

ABSTRACT

In this paper we present an universal methodology for finding all the occurrences of a given set of subgraphs in one single larger graph. Past approaches would either enumerate all possible subgraphs of a certain size or query a single subgraph. We use g-tries, a data structure specialized in dealing with subgraph sets. G-Tries store the topological information on a tree that exposes common substructure. Using a specialized canonical form and symmetry breaking conditions, a single non-redundant search of the entire set of subgraphs is possible. We give results of applying g-tries querying to different social networks, showing that we can efficiently find the occurrences of a set containing subgraphs of multiple sizes, outperforming previous methods.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; E.1 [**Data Structures**]: [Graphs and networks]

General Terms

Algorithms, Experimentation, Performance

Keywords

Graph Mining, Subgraphs, G-Tries, Network Motifs

1. INTRODUCTION

Graph mining is ubiquitous and has applicability on a multitude of fields [3]. One crucial task is to find relevant subgraphs, trying to discover patterns that can help us to better understand the structure and function of the underlying networks. The basic premise relies on being able to efficiently discover subgraphs, which is a computationally hard problem, closely related to the subgraph isomorphism problem, which is known to be NP-complete [5].

Past approaches for finding subgraph occurrences tend to rely on two almost opposite approaches. On one hand, we have algorithms that search for one specific individual subgraph, such as GraphGrep [6] (*subgraph-centric* approach). On the other hand we have algorithms that need to enumerate all subgraphs of a specific size, such as ESU [23] (*network-centric*). If we want to search for a specific set of subgraphs, in the first case we need to make several queries (not taking advantage of the previous computation), and in the second case we need to potentially do several queries and then filter the results to the subgraphs that interest us (spending time searching for unnecessary results).

Searching for subgraph sets is however a very useful task, with growing demand. For example, when finding network motifs [14] the bottleneck of the computation is finding the frequency of subgraphs that appear in the original network on a large set of similar randomized networks. Occurrences of subgraph sets are also being used to create characteristic graphlet fingerprints of networks [16], than can be used for comparison. Both these approaches have direct applicability in the social networks domain. For instance, motif profiles have been used to characterize the edition network of Wikipedia articles (showing predictive power on the quality of the articles) [25], and graphlets have been shown to be a powerful way of characterizing the social network structure (enabling the selection of an adequate model) [10]. Recent work has also shown that different subgraphs have different discriminative power [4], which increases the need for being able to search for pre-defined sets of subgraphs.

By acknowledging that we are indeed looking for a certain subgraph set, we may improve our methodology when comparing to previous methods. The more traditional approach to subgraph mining has its roots on association rule mining, and the associated problem has its core on finding frequent subgraphs supported by a set of graphs [12]. This leads to the creation of index structures for the set of graphs that are in the database [6, 9]. By contrast, our approach relies on the creation of an index like structure for the set of subgraphs being queried, by using the very recent g-tries data structure [19].

In this paper we present a methodology for performing an universal exact query of a set of subgraphs on a single larger network. We pre-process the subgraph set to create the correspondent g-trie, a tree that encapsulates information about common topologies. By adding symmetry breaking conditions to the g-trie, we are able to devise an algorithm that efficiently searches for the entire subgraph set using a single constrained pass through the entire network, in which each matched occurrence is already guaranteed to be isomorphic equivalent to one of the subgraphs in the searched

http://doi.acm.org/10.1145/2304536.2304541
.

set. The method is completely application-independent and can be applied to any base set of subgraphs, regardless of their size. It is also very flexible, meaning that it can be applied both to directed and undirected graphs, and also to colored and uncolored networks. We show experimental results when running our algorithm, showcasing its general applicability, and showing it can outperform previous algorithms for this specific task.

The remainder of the paper is organized as follows. Section 2 defines the problem we are tackling and overviews related work. Section 3 introduces the g-trie data structure and details how we use it for querying subgraph sets. Section 4 discusses the experimental results obtained, with comparisons to previous algorithms. Section 5 concludes the paper, commenting on the results and possible future work.

2. PRELIMINARIES

2.1 Subgraph Set Query

We formalize the problem we are trying to solve, which is to discover all occurrences of a certain set of subgraphs in a larger graph:

Definition 1. **General Subgraph Set Query:** Given a set of subgraphs S_G and a graph G, determine all exact induced occurrences of subgraphs of S_G in G. Two occurrences are considered different if they have at least one node or edge that they do not share.

Note that this definition if very flexible and can be applied to any type of graphs, directed and undirected, colored and uncolored. The last part establishes how we can distinguish different occurrences, stating the most widely used definitione. This has a direct impact on the number of subgraphs found, and thus, on the *hardness* of the problem. Other definitions may use different concepts [22].

Note also that for the purposes of this paper, we are concerned only with induced subgraphs, that is, the vertex set of the subgraph has exactly the same set of edges as the matched vertex set in the larger graph. G-tries could also be used to find non-induced matches if that was intended.

2.2 Related Work

The concept of g-tries was introduced by us in 2010 [19]. In previous work our main focus has been to compute the frequency of subgraphs of size k (k-subgraphs) for a given static k (implying the creation of a g-trie with all possible k-subgraphs). In this paper, besides a more mature underlying platform, the main innovation is the usage of sets of differently sized subgraphs on the same g-trie, allowing a single search for any general set of subgraphs. We have also done some preliminary work on parallelizing the g-tries associated algorithms [21], and on obtaining approximate results by using sampling [18].

Network-centric approaches look for all possible k-subgraphs for a certain k, by enumerating connected sets of k-vertices and in the end doing tests to discover the isomorphic class of each subgraph found. To our best knowledge, the most efficient algorithms for this task are ESU [23] and Kavosh[11]. Subgraph-centric approaches, in contrast, query individual subgraphs. Grochow and Kellis [7] show an efficient way of doing this. GraphGrepSX [2], based on its predecessor GraphGrep [6] is a state of the art subgraph-centric querying system for a database of graphs.

We should also note that the problem we are trying to solve is conceptually very different from *frequent subgraph discovery* [12], given that in the later we do not have a predefined set of querying subgraphs from which we need to know all possible occurrences, but instead we have a set of original graphs from which we need to extract frequent subgraphs that appear in all of them. In this way, algorithms such as gSpan [26] perform a different task and are not directly comparable.

3. QUERYING SUBGRAPH SETS

3.1 The G-Trie Data Structure

A g-trie is a multiway tree (with a variable number of children per node) able to store and describe a set of graphs. In order to avoid ambiguities, from now on we will use the term *node* for referring to g-trie tree nodes, and the term *vertex* to refer to the actual graph vertices stored in the g-trie. Each node contains information about a single vertex and its connections to ancestor vertices. Descendants of a node share a common subtopology. A path from the root to any given node defines one single subgraph.

Figure 1 gives an example of a g-trie with 9 undirected subgraphs: all possible subgraphs from sizes 2 to 4. Each g-trie node adds a new graph vertex (in black) to the already existing ones in the ancestor nodes (white vertices). The nodes indicated in gray are terminal nodes, meaning that they store the last vertex of a graph, that is, a path from the root to a gray node defines a graph in the stored set.

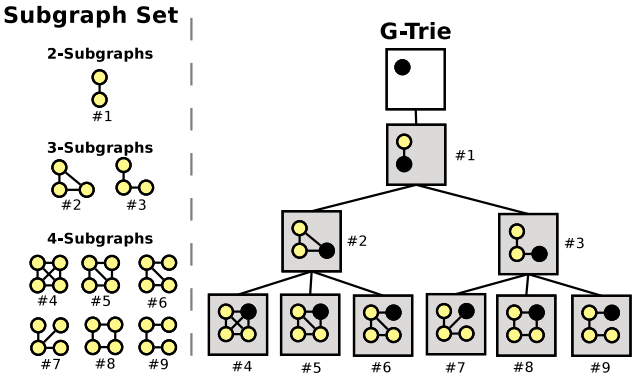

Figure 1: A g-trie storing 9 undirected subgraphs.

For the sake of clarity, and given the space constraints, the following sections will focus only on undirected graphs. The explained concepts are however easily extendable to directed and colored graphs.

3.2 Creating a G-Trie

A g-trie is created by an iterative algorithm that inserts one graph at a time, as exemplified in Figure 2. As before, gray nodes indicate the terminal node of a graph. The dashed nodes indicate the actual g-trie path that corresponds to the newest inserted graph. The numbers of the graph vertices indicate the order in which they are added.

Algorithm 1 describes the main steps used in the creation of a g-trie. Basically we iterate trough all graphs of the set (line 3) and execute a series of procedures for each of them. The first one is to obtain a canonical form for the graph (line 4). This determines the order in which the vertices are

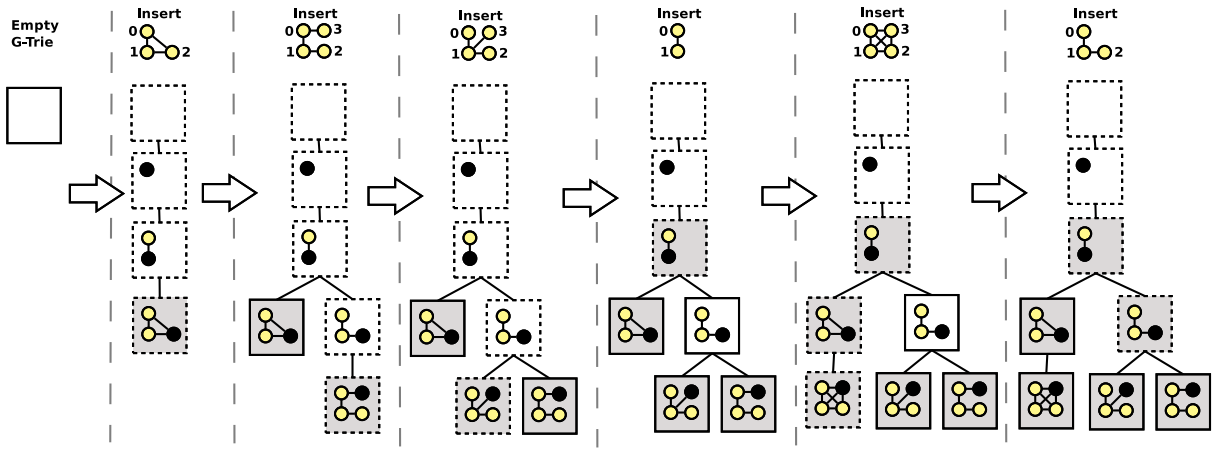

Figure 2: Example iterative insertion of 6 undirected subgraphs in a g-trie.

inserted and therefore the path through the g-trie. We need a canonical form in order to ensure that the same graph will always give origin to the same path, meaning that an unique graph set will give origin to an unique g-trie (regardless of the order in which we insert the graphs).

Algorithm 1 Creating a G-Trie from a set of subgraphs

Require: Graph Set S_G
Ensure: Inserts all Graphs of S_G in G-Trie T

```
 1: procedure CREATEGTRIE(S_G)
 2:     T := empty G-Trie
 3:     for all graph G of S_G do
 4:         Str := canonical form of G
 5:         Cond := symmetry breaking conditions of G
 6:         INSERT(T.root, Str, Cond, 0, |V(G)|)
 7:     Filter Conditions of T
 8:     return T
 9: procedure INSERT(Node, Str, Cond, k, size)
10:     Add relevant conditions of Cond to Node
11:     if k = size then
12:         Mark Node as terminal node of a Graph
13:     else
14:         for all children c of Node do
15:             if c.connections = k-vertex of Str then
16:                 INSERT(c, Str, Cond, depth + 1, size)
17:                 return
18:         c := new G-Trie node
19:         c.connections = k-vertex of Str
20:         INSERT(c, Str, Cond, depth + 1, size)
```

The choice of canonical form will have a direct impact on the topology of the g-trie. What we want is to produce the g-trie with as less nodes as possible, meaning that there are more common subpaths between different graphs, and therefore more common substructure. We will use this to improve the search procedure, as described in the next section. Note that computing a canonical from is a hard problem since it is related to graph isomorphism (two graphs are isomorphic if they have the same canonical form). Our current implementation uses *GTCanon*, a customized canonical form that tries to guarantee three properties: connectivity (a path in the g-trie will always define a connected graph), compressibility (as more common substructure as possible) and constraining (as more connections as possible in the first vertices of the path, in order to highly constrain the possible matching nodes in the larger graph). The actual algorithm uses `nauty` [13], a proven and very efficient algorithm, as the basis to obtain a fast and specialized canonization. More detail on this can be seen in [17]. As an example, when applied to the graph set of Figure 1, *GTCanon* would give origin to the pictured g-trie.

The symmetry breaking conditions role (lines 5, 7 and 10) will be explained in the next section, given that they are specially pertinent in the context using g-tries for querying, but what remains is that they are computed for each inserted graph and in the end they are filtered. The rest of the algorithm is an almost straightforward recursive procedure (`insert`) that follows the path that corresponds to the graph being inserted and creates new g-tries nodes as needed.

3.3 Using G-Tries to Query Subgraphs

With the g-trie already built, we are now ready to search for all its stored graphs as induced subgraphs of another larger graph. The main methodology is to search for vertices in the larger graph that match the subgraph represented in a g-trie node, and increasingly grow the partial match so that full occurrences are discovered. The efficiency gain is obtained by matching several subgraphs at the same time, in the sense that a partial match is a potential occurrence of all the subgraphs stored on descendant nodes. Furthermore, when on a subgraph terminal node, we already are assured that the match is isomorphic to the respective subgraph. This is different from what happens in subgraph-centric approaches, in which different individual subgraph queries would not take advantage of previous results. It is also different from what happens in network-centric approaches, since in these all subgraphs would have to be enumerated and the isomorphic test would be delayed to after the enumeration.

A naive approach for the described methodology would encounter a problem when dealing with the symmetry that inherently exists in subgraphs. This is because for each unique occurrence of a subgraph, we can find several possible matches, corresponding to the different number of possible automorphisms of that subgraph. For instance, each triangle would be found 6 times.

In order to counter this and to effectively find each subgraph instance just once, we introduce symmetry breaking conditions of the form $a < b$. These impose constraints on the labels of the actual matches found, in accordance to their respective subgraph vertex matching, guaranteeing only one possible matching that respects these ordering conditions. Figure 3 exemplifies this, showing conditions for a triangle and how they disallow 5 of the 6 possible automorphisms.

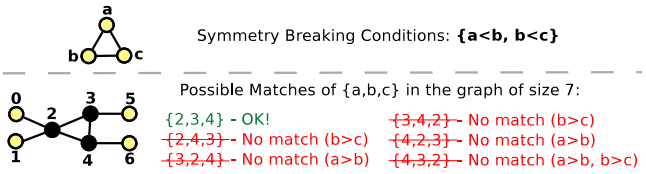

Figure 3: Symmetry breaking conditions example.

The conditions are defined similarly to what was done by Grochow and Kellis [7], but we use a different condition generation algorithm. For each graph, after having applied the canonization of *GTCanon*, we generate the set of possible automorphisms. We then proceed by finding the first vertex that is not fixed, that is, the one that still has equivalent vertices (for instance, in the case of the triangle, all vertices are equivalent). We then add a condition that fixes that vertex, guaranteeing that it must be smaller than all equivalent vertices. We proceed by finding that next non fixed vertice and applying the same process. More detail on the exact algorithm can be seen in [19].

After generating these conditions, we add them along the g-trie path that corresponds to that subgraph. When we are creating a possible match to a g-trie node, we ensure that the conditions of at least one possible descendant terminal node are respected. When all subgraphs are inserted with the respective conditions we apply a filtering process that reduces the total number of conditions, by removing redundancies. For instance, a simple case is to use the transitivity of the order relationship, reducing sets like $a < b, a < c, b < c$ to $a < b, b < c$. More specific details on all the steps taken by our filtering process can be seen in [17]. Figure 4 illustrates how a g-trie storing the six undirected 4-subgraphs would look like with the conditions added.

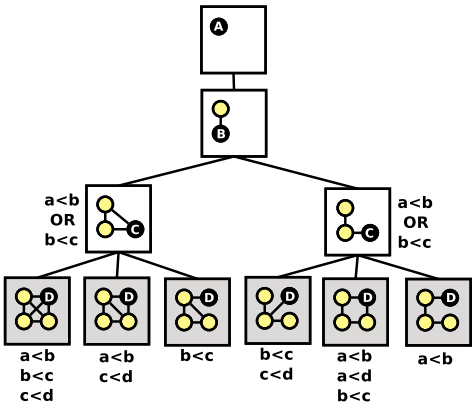

Figure 4: A g-trie with symmetry breaking conditions.

Given all of this we are ready to explain how we query the subgraphs. Algorithm 2 overviews the procedure for finding all occurrences of the graphs of a g-trie as induced subgraphs of another larger graph.

Algorithm 2 Querying a set of subgraphs

Require: G-Trie T and Graph G
Ensure: Occurrences of subgraphs of T in G

```
1:  procedure QUERYGTRIE(T, G)
2:      for all children c of T.root do
3:          QUERYNODE(c, ∅, G)
4:  procedure QUERYNODE(Node, V_used, G)
5:      V_cand := candidates of V(G) that match Node
6:      for all v ∈ V_cand do
7:          if isTerminal(Node) ∧ conditionsOk(Node) then
8:              foundOccurrence(V_used ∪ v)
9:          for all children c of Node do
10:             QUERYNODE(c, V_used ∪ v, G)
```

The searching methodology of the algorithm is to use a recursive backtracking procedure (`queryNode`) that basically tries to find an extension to existing partial match that complies with all the constraints imposed by the current g-trie node (in terms of connections and labels). A set of candidate vertices is generated (line 5) and then, for each of them, if the node is terminal and the symmetry conditions for that particular subgraph are met (line 7), we found an occurrence (line 8). If the node has children, then all possible g-trie continuation paths are tried (lines 9 and 10).

The core of the work that greatly influences the performance of the whole algorithm is the generation of the candidate vertices (line 5). For space reasons, we will not dwell in to the small details of this part (check [17] for more thorough information) but the main goal is to generate a list of vertices of the larger graph that simultaneously obey to the topological constraints (having all the needed connections) and to the ordering constraints (not breaking symmetry conditions). Much care is taken to do this as efficiently as possible, and the basic idea is that already matched nodes (V_{used}) can constraint as much as possible, and as early as possible, the candidates. We follow three main concepts: (1) the vertex from V_{used} with the smaller neighborhood in the graph and a connection to the current g-trie vertex gives the smaller number of initial candidates; (2) the symmetry conditions stored on the node limit the interval of possible candidate labels; (3) only those candidates that respect all the connections with V_{used} can remain.

When comparing with other methods, the potential gain in efficiency is essentially due to the fact the we are able to test at the same time several different subgraphs. In fact, when we have a partial match of graph vertices for a certain g-trie node, we have already an isomorphic partial match to all the subgraphs stored in descendants nodes. This contrasts with subgraph-centric methods that do not take advantage of the results of previous queries, and with network-centric methods that indeed need to enumerate all possible subgraphs instead of just the ones we are interested in. For querying with g-tries, in general, the more common substructure we are able to find, and the larger number of graphs we are looking for, then the more potential saving we will obtain in the execution time.

4. EXPERIMENTAL RESULTS

In order to evaluate our approach, we implemented the described algorithms in C++. All experiments were run on

an Intel Core i5 M 450 (2.40GHz) with 4GB of memory. We use three different social networks, with varied topological features, as described in Table 1, showcasing the general applicability of our approach. All networks used were converted if necessary to be undirected and unweighted.

Name	$\|V(G)\|$	$\|E(G)\|$	Brief Description	Ref
`citations`	395	994	citations between authors	[1]
`email`	1.133	5.451	e-mails interchanges	[8]
`coauthors`	8.361	15.751	co-authorship of papers	[15]

Table 1: Networks used for experimentation.

In order to show the universal applicability of our methodology we gathered the results when querying three very different subgraph sets:

- **Complete:** all possible 141 undirected subgraphs from sizes 3 to 6 (demonstrating full enumeration).
- **Cycle:** all 6 single cycles from sizes 3 to 8, that is, subgraphs connected in a closed chain, with each vertex connected to two other vertices (demonstrating the search for a specific type of graph).
- **Random:** 20 different random subgraphs obtained by uniformly sampling from the entire possible set of subgraphs from sizes 3 to 8 (demonstrating general applicability).

We measure the execution time taken for building the gtrie plus running the query that finds all induced occurrences of the subgraph set on the larger graph. We compare the results obtained with two different state-of-the art algorithms, both of them implemented by us in the same platform, using the same graph primitives, in order to make a fairer comparison so that the differences are really in the algorithm.

- **ESU:** one of the fastest [20] network-centric enumeration algorithms. Our implementations gives execution times even slightly smaller than the original author's C++ implementation [24]. Since ESU needs to do a full enumeration of all possible k-subgraphs, we only use it for querying `Complete`. For the other cases the algorithm would need to do full enumerations up to size 8, which would take much more time (more than 1000 times) than the other presented alternatives.
- **Grochow**: Grochow and Kellis [7] provide a state of-the-art subgraph-centric algorithm for querying a single subgraph on a single larger graph. The original implementation was in Java. For querying the subgraph sets, we execute successive individual queries for each subgraph and measure the total time spent.

We intended to compare our results with GraphGrepSX [2]. However, preliminary tests with version 3 have shown that for this particular task it was much slower. Note that this algorithm is specialized in searching for a single subgraph in a set of larger graphs, building an index file for the entire database of graphs. It is also thought primarily for non-induced subgraphs, even if it allows induced matches, and it is returning symmetric results. We have observed that even for simple cases, such as querying all 8 subgraphs from sizes 3 to 4, g-tries querying was more than two orders of magnitude faster, and for bigger sizes the difference in performance was growing even more.

Table 2 shows the obtained results for every combination of larger graph, query subgraph set and algorithm. `'#'` depicts the number of different subgraphs in the query set and `matches` the total number of real occurrences of these subgraphs in the graph. `Speedup` indicates the amount of times g-tries was faster than the other algorithms.

We can observe that g-tries perform consistently better than the two other algorithms. For the `complete` case, even when we are in fact finding all possible subgraphs, g-tries can perform much better (by more than one order oh magnitude) than ESU, an algorithm built specifically for enumeration. However, it should be noted, for a full enumeration of much larger subgraph sizes, the cardinality of the needed subgraph set may become too big for practical usage with this type of query (even if in that case the full network-centric enumeration will also be almost prohibitive).

When comparing to Grochow, we can see that in the `complete` case the speedup is larger than 10x. If we reduce the query set to a very small size, like with `cycle`, then the speedup decreases, as expected, since there is less to gain (less common subtructure and less number of subgraphs). However, looking at `random` case, we can see that quickly g-tries can really take advantage of the fact that can search simultaneously several subgraphs, with the speedups being even larger than in the `complete` case.

The actual speedup obtained depends on a series of factors and is not easily predictable. As said, the key is to have common structure that we use for avoiding redundancy in the search procedure, either because the subgraphs are of the same "family" or because the sheer number of subgraphs implies the existing of common topologies. For specific types of graphs, there may exist better specialized options, but g-tries remain as an efficient approach to any general type of query with multiple subgraphs.

In all the experiments of the table, the time needed for building the g-trie itself was negligible in the total amount of time, taking less than 0.1s. For a fairer comparison of times, we recomputed the g-trie in all experiments, but it is possible to save it for later reuse.

5. CONCLUSIONS

In this paper we have described a methodology for an application-independent querying of subgraph sets. We use g-tries, a data structure that allows the storage of a collection of subgraphs, with a customized canonical labeling that helps in identifying common-substructure. We described how to add symmetry breaking conditions that allow a single efficient search of the entire subgraph set over a larger graph. The results obtained show that not only our approach is feasible, but that it also significantly outperforms previous algorithms. G-Tries are best suited for cases in which we have a fair amount of subgraphs that we want to query, and when common topologies exist in that query set.

In the near future we plan to apply this strategy to more complex networks, including directed and colored graphs. We also intend to provide full support for a sampling version, trading accuracy for better execution times [18], and for a parallel version of the described algorithms [21].

6. ACKNOWLEDGMENTS

This work is in part funded by the ERDF/COMPETE Programme and by FCT within project FCOMP-01-0124-FEDER-022701. Pedro Ribeiro is funded by an FCT Research Grant(SFRH/BPD/81695/2011).

Graph	Subgraphs Set			Execution time (seconds)			Speedup of G-Tries vs	
	Name	#	Matches	G-Tries	Grochow	ESU	Grochow	ESU
citations	complete	141	927,461,151	19.07	292.88	2,925.76	15.4x	153.4x
	cycle	6	29,858	0.09	0.28	—	3.2x	—
	random	20	67,050,079	2.02	41.73	—	20.6x	—
email	complete	141	1,427,281,118	36.04	509.58	2,051.83	14.1x	56.9x
	cycle	6	51,625,766	84.44	396.26	—	4.7x	—
	random	20	29,791,398	1.63	56.85	—	34.9x	—
coauthors	complete	141	264,011,611	10.02	192.34	393.93	19.2x	39.3x
	cycle	6	901,850	9.25	22.75	—	2.5x	—
	random	20	5,460,888	0.71	19.03	—	26.9x	—

Table 2: Experimental results of querying subgraph sets with g-tries.

7. REFERENCES

[1] V. Batagelj and A. Mrvar. Pajek datasets. 2006. `http://vlado.fmf.uni-lj.si/pub/networks/data/`.

[2] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *5th IAPR Int. Conference on Pattern Recognition in Bioinformatics (PRIB)*, pages 195–203, Berlin, 2010. Springer.

[3] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38:1, 2006.

[4] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *18th Int. Symposium on Software Testing and Analysis*, pages 141–152, NY, USA, 2009. ACM.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *3rd ACM Symposium on Theory of computing (STOC)*, pages 151–158, NY, USA, 1971.

[6] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *16th International Conference on Pattern Recognition*, volume 2, pages 112–115, 2002.

[7] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. *Research in Computational Molecular Biology*, pages 92–106, 2007.

[8] R. Guimerà, L. Danon, A. D. Guilera, F. Giralt, and A. Arenas. Self-similar community structure in a network of human interactions. *Physical Review E*, 68(6):065103+, 2003.

[9] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *22nd International Conference on Data Engineering (ICDE)*, pages 38–, Washington, DC, USA, 2006. IEEE CS.

[10] E. Janssen, M. Hurshman, and N. Kalyaniwalla. Model selection for social networks using graphlets. *Internet Mathematics*, 2012.

[11] Z. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10(1):318, 2009.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE International Conference on Data Mining (ICDM)*, page 313, Los Alamitos, CA, USA, 2001. IEEE CS.

[13] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[14] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[15] M. E. J. Newman. The structure of scientific collaboration networks. *National Academy of Sciences of the USA*, 98(2):404–409, 2001.

[16] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23:e177–e183, Jan. 2007.

[17] P. Ribeiro. *Efficient and Scalable Algorithms for Network Motifs Discovery*. PhD thesis, University of Porto, 2011.

[18] P. Ribeiro and F. Silva. Efficient subgraph frequency estimation with g-tries. In *International Workshop on Algorithms in Bioinformatics (WABI)*, volume 6293 of *LNCS*, pages 238–249. Springer, September 2010.

[19] P. Ribeiro and F. Silva. G-tries: an efficient data structure for discovering network motifs. In *25th ACM Symposium on Applied Computing (SAC)*, pages 1559–1566. ACM, March 2010.

[20] P. Ribeiro, F. Silva, and M. Kaiser. Strategies for network motifs discovery. In *5th IEEE International Conference on e-Science*, pages 80–87, Oxford, UK, December 2009. IEEE CS.

[21] P. Ribeiro, F. Silva, and L. Lopes. Efficient parallel subgraph counting using g-tries. In *IEEE International Conference on Cluster Computing (Cluster)*, pages 1559–1566. IEEE CS, September 2010.

[22] F. Schreiber and H. Schwobbermeyer. Towards motif detection in networks: Frequency concepts and flexible search. In *International Workshop on Network Tools and Applications in Biology*, pages 91–102, 2004.

[23] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):347–359, 2006.

[24] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, May 2006.

[25] G. Wu, M. Harrigan, and P. Cunningham. Characterizing wikipedia pages using edit network motif profiles. In *3rd Int. workshop on search and mining user-generated contents (SMUC)*, pages 45–52, New York, NY, USA, 2011. ACM.

[26] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *2nd IEEE International Conference on Data Mining (ICDM)*, page 721, Washington, DC, USA, 2002. IEEE CS Press.