Scalable Motif Detection and Aggregation

Jens Dietrich  Catherine McCartin
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
Email: {j.b.dietrich,c.m.mccartin}@massey.ac.nz

Abstract

Motif search in graphs has become a popular field of research in recent years, mainly motivated by applications in bioinformatics. Existing work has focused on simple motifs: small sets of vertices directly connected by edges. However, there are applications that require a more general concept of motif, where vertices are only indirectly connected by paths. The size of the solution space is a major limiting factor when dealing with this kind of motif. We try to address this challenge through motif instance aggregation. It turns out that effective, parallel algorithms can be found to compute instances of generalised motifs in large graphs.

To expedite the process, we have developed GUERY, a tool that can be used to define motifs and find motif instances, in graphs represented using the popular JUNG graph library [10]. GUERY consists of two parts: a simple domain specific language that can be used to define motifs, and a solver. The main strengths of GUERY are 1. support for motif instance aggregation, 2. generation of query result streams, as opposed to (very large) static sets of matching instances, 3. support for effective parallelisation in the evaluation of queries.

The examples used for validation originate from problems encountered when analysing the dependency graphs of object-oriented programs for instances of architectural antipatterns.

1 Introduction

The detection of motifs has become a popular field of research in recent years, driven by applications in bioinformatics. For example, Milo and colleagues [9] have attempted to study the structural design principles of different networks from biochemistry, neurobiology, ecology, and engineering, by detecting network motifs: patterns of interconnections occurring in these networks at numbers that are significantly higher than those in randomized networks. The motifs considered in [9] are small sets of interconnected vertices. Interconnected there means that there are edges in the network directly linking the vertices participating in the motif. In other domains, similar structures have been studied. For instance, motifs are closely related to design patterns [7] - a concept widely used in software engineering to describe and communicate design. Use cases, such as pattern-based program comprehension, have led to research in design pattern formalisation [17], and the construction of tools to detect those patterns in network models representing the design of the software that is being analysed. Another class of patterns that have been studied in software engineering are architectural antipatterns - patterns representing design flaws in programs that have a negative effect on quality attributes of the respective programs. The classical architectural antipattern is the existence of a circular dependency between modules [16]. Several other patterns have been studied, including diamond inheritance [14, 15] and subtype knowledge [13]. These patterns can be defined with respect to the dependency graph, representing the high level design of the artefacts of the software and their relationships. Design patterns and antipatterns are similar to the motifs considered in [9], except that they do not assert that the participating vertices are directly connected. Instead, indirect connections via paths are permitted.

In [6] we presented results of analysing the dependency graphs of object-oriented programs for instances of architectural antipatterns. This analysis required, in part, the enumeration of instances of such patterns, often in very large graphs. In response to challenges encountered in our previous work, we have developed GUERY [1], a tool that can be used to define motifs and find motif instances, in graphs represented using the popular JUNG graph library [10]. GUERY consists of two parts: a simple domain specific language that can be used to define motifs, and a solver. The main strengths of GUERY are 1. support for motif instance aggregation, 2. generation of query result streams, as opposed to (very large) static sets of matching instances, 3. support for effective parallelisation in the evaluation of queries. We consider that the techniques employed in the development of GUERY may be widely applicable to motif detection in a variety of contexts.

The rest of this paper is organised as follows. In section 2 we review related work. We formally define some concepts needed later in section 3, and discuss a domain specific language that can be used to define motifs in section 4. Next we introduce motif aggregation in section 5. The algorithms we have developed for motif detection are covered in section 6, and some experiments we have conducted in order to validate these algorithms are presented in section 7. A discussion of the results presented concludes our contribution.

1 http://sites.google.com/site/gueryframework/
2 Related Work

One existing graph analysis tool closely aligned to our purpose of detecting generalised motifs in graphs is CrocoPat [3]. CrocoPat manipulates relations of any arity. Its query and manipulation language, RML, is a full programming language with syntax elements such as conditionals and loops, based on first-order predicate calculus. The implementation is based on binary decision diagrams (BDDs), well-known as a compact representation of large relations. CrocoPat queries are purely relational, each query returns facts for a certain predicate symbol. Paths cannot be directly represented in CrocoPat. However, reasoning about paths in CrocoPat is supported through the higher-order transitive closure predicate TC, and, in fact, each of the motif patterns used for validation (Section 7) in this paper can be described by queries expressible in CrocoPat. Based on experimental evidence, CrocoPat is rather less efficient memory-wise than GUERY. More importantly, however, the underlying design of CrocoPat does not lend itself to parallelisation, nor do current implementations of CrocoPat support the generation of query result streams.

GRoQL (Graph Repository Query Language) is another graph query language, built to extract information from TGraphs. TGraphs are typed, attributed, directed, and ordered graphs which conform to TGraph schemas. In turn, schemas are metamodels conforming to the metamodel of the graph modeling language grUML, a subset of UML class diagrams. Similar to SQL queries in relational databases, a typical GRoQL query specifies the range of some free variables, poses some conditions on these variables and describes the desired output. The conditions are specified using GRAL (Graph Specification Language), a predicate language based on Z. A central concept in GRoQL are regular path expressions. A regular path expression describes a path through a graph based on edge types and edge direction, and using basic regular expression syntax. A path expression can be used to test if pairs of nodes are connected via the specified path, to denote the set of vertices which can be reached from a starting vertex via the path, or to denote the set of vertices from which a particular vertex can be reached via the path. Similar functionality is incorporated into GUERY, but without the explicit regular expression syntax. In contrast to GUERY, currently available implementations of GRoQL do not support parallelisation in the evaluation of queries, nor do they support generation of query result streams.

SPARQL [12] supports a standardized query language and data access protocol for the semantic web. SPARQL is defined in terms of the W3C’s RDF data model and will work for any data source that can be mapped into RDF. SPARQL is an SQL-like language whose features include basic conjunctive patterns, value filters, optional patterns, and pattern disjunction. It works primarily via constraints on single vertices. SPARQL is considered a key semantic web technology, but it is insufficient for our purposes, since, in contrast to the tools introduced above, it does not support path constraints (transitive closure).

Several algorithms have been proposed for graph pattern matching [4], [20]. Our approach is more expressive as it supports the use of expressions on both vertex and edge labels (instead of just matching vertex labels), length constraints on individual connections (in [20], only static path length constraints that apply to all connections are considered), and aggregation clauses.

3 Motifs and Motif Instances

In this section we formally introduce motifs and related concepts.

Definition 1 (Path) Let \( G = (V, E) \) be a directed graph consisting of a set of vertices \( V \) and a set of directed edges (arcs) \( E \) (we use the term edge to denote a directed edge throughout this work.) A path is a finite sequence of edges \((e_1, \ldots, e_n)\) such that for adjacent edges \( e_i \) and \( e_{i+1} \) the target vertex of \( e_i \) is the source vertex of \( e_{i+1} \). \( \text{SEQ}(E) \) is the set of all paths that can be constructed from given set of edges \( E \). For a given path \( p \), \( \text{start}(p) \) denotes the first vertex (source vertex of \( e_1 \)), \( \text{end}(p) \) the last vertex (target vertex of \( e_n \)) and \( \text{length}(p) \) the number of edges in the path.

We will consider paths of length 0 - these are paths consisting of a single vertex and no edge. This single vertex is both the start and the end vertex of the empty path. A regular path is a path where any edge occurs only once in the sequence defining the path. We use \( \text{SEQ}_{\text{reg}}(E) \) to denote the set of all regular paths constructible from \( E \). Even though \( E \) is finite, \( \text{SEQ}(E) \) is possibly infinite. Therefore, we will often restrict our investigation to regular paths.

Definition 2 (Motif) Let \( G = (V, E) \) be a directed graph consisting of a set of vertices \( V \) and a set of directed edges \( E \). A motif over \( G \) is a structure \( M = (V_R, P_R, s, t, C_V, C_P) \) consisting of:

1. A set of vertex roles \( V_R \).
2. A set of path roles \( P_R \).
3. Two functions \( s : P_R \rightarrow V_R \) and \( t : P_R \rightarrow V_R \), associating path roles with vertex roles.
4. A set of vertex constraints \( C_V \), consisting of \( n \)-ary predicates between vertex tuples, \( C_V \subseteq \times_{i=1..n} V \), where \( n \) is the cardinality of \( V_R \).
5. A set of path constraints \( C_P \), consisting of \( n \)-ary predicates between tuples of paths, \( C_P \subseteq \times_{i=1..n} \text{SEQ}(E) \), where \( n \) is the cardinality of \( P_R \).

Intuitively, constraints restrict the sets of possible vertex and path assignments. While vertex constraints are always defined with respect to vertex labels, we consider two types of path constraints: cardinality constraints that restrict the length of permitted paths, and constraints defined with respect to labels defined on the edges in the path.

Usually, the vertex roles in motifs are cohesive. By this, we mean that they are linked via path roles. This can be defined as follows:

Definition 3 (Motif graph and connected motifs) Let \( M = (V_R, P_R, s, t, C_V, C_P) \) be a motif over a directed graph. The motif graph \( G_M = (V_M, E_M) \) is then defined as follows: \( V_M := VR \), \( E_M := P_R \), for each \( pr \in P_R \), \( \text{start}(pr) := s(pr) \) and \( \text{end}(pr) := t(pr) \). \( M \) is called connected if \( G_M \) is weakly connected.

A binding associates vertex roles with vertices and path roles with sequences of edges. The \( s \) and \( t \) functions associate path and vertex roles: path roles connect source and target vertex roles. The conditions define a structural homomorphism between the motif and the target graph, as shown in figure 1.
Definition 4 (Binding) Let $G = (V, E)$ be a directed graph and $M = (V, R, P, s, t, C_P)$ a motif. A binding is a pair of functions $(\text{inst}_V, \text{inst}_P)$, where $\text{inst}_V : V \rightarrow V$ and $\text{inst}_P : P \rightarrow \text{SEQ}(E)$, such that $\text{inst}_V(s(pr)) = \text{start}(\text{inst}_P(pr))$ and $\text{inst}_V(t(pr)) = \text{end}(\text{inst}_P(pr))$.

A binding does not necessarily satisfy the sets of constraints that are part of the motif definition. If it does, we call it valid.

Definition 5 (Valid binding) Let $G = (V, E)$ be a directed graph, $M = (V, R, P, s, t, C_P)$ and $\text{bind} = (\text{inst}_V, \text{inst}_P)$ a binding. $\text{bind}$ is called valid if the following two conditions are fulfilled:

1. $(\text{inst}_V(v_1), \ldots, \text{inst}_V(v_n)) \in C_V$ for all vertex constraints $C_V 
2. (\text{inst}_P(p_1), \ldots, \text{inst}_P(p_n)) \in C_P$ for all path constraints $C_P 

The question arises whether valid bindings are suitable representations for motif instances. The main problem is that path roles can be bound to arbitrary paths. Since the set of paths can be infinite even though the set of edges is finite, it is reasonable to restrict our consideration to bindings where even though the set of edges is finite, it can be easily shown that this converges to the Taylor series expansions of the exponential function.

For instance, consider the simple graph shown in Figure 2:

![One node graph](image)

Figure 2: One node graph

A possible solution is to identify as equivalent bindings that define the same vertex bindings, but potentially different path bindings. For many applications this is sufficient: users want to compute one or more paths to understand the relationships between vertices, but not necessarily the complete set. More precisely we define a motif instance as follows:

Definition 6 (Motif Instance) Let $G = (V, E)$ be a directed graph, $M = (V, R, P, s, t, C_P)$ a motif and $B = \{\text{inst}_V, \text{inst}_P\}$ the set of all valid bindings. Then the relationship $\sim \subseteq B \times B$ is defined as follows:

$(\text{inst}_V, \text{inst}_P) \sim (\text{inst}_V', \text{inst}_P')$ iff $\text{inst}_V = \text{inst}_V'$ and $\text{inst}_P = \text{inst}_P'$.

It is obviously an equivalence relation. We define a motif instance to be a class of valid bindings modulo $\sim$.

4 Defining Motifs

We have developed GUERY, a tool that can be used to define motifs and find motif instances, in graphs represented using the popular JUNG graph library [10]. GUERY consists of two parts - a language definition for motifs and a parser generated from this language definition, and a solver. We discuss the language definition in this section, the algorithms used by the solver are discussed in Section 6.

GUERY/DSL is a domain specific language that can be used to define motifs. Its syntax combines syntax elements from SQL and object-oriented expression languages. The language is formally defined using the ANTLR parser generator language [11].

An example motif definition is shown in listing 1. This motif represents "subtype knowledge" [13], a design flaw sometimes found in dependency graphs of object-oriented programs. This motif is an example of a so-called architectural antipattern. The motif definition supports one-line comments (line 1), and naming of motifs (line 2). Following the select keyword, vertex roles are defined (line 3). Path roles are defined following the connected by keyword (line 4). Each path role is defined by a unique name, followed by the source and target vertex roles separated by "->" and the definition of cardinality constraints restricting the length of the instantiating paths. This is done by pairs of values defining the minimum and the maximum value, with "*" representing unbound.

If no cardinality constraints are defined, the default $[1,*]$ is used.

The motif has several constraints (line 5). The constraints are string literals, interpreted using the existing object-oriented MVEL expression language [1]. The MVEL expression engine can compile these constraints dynamically into optimised Java byte code, making the evaluation of expressions very fast. The expressions refer to either edges or vertex and their labels. For instance, the expression "inherits.type=="extends" evaluates to true if the type label for each edge in a path instantiating the inherits path role equals "extends".

Note that labels can be complex structures. For instance, it is possible to refer to the first two characters of the type label using the expression "inherits.type.substring(2)".

The motif defined in listing 2 represents a circular dependency between name spaces. Name spaces are used in object-oriented programming to group related classes together. Circular dependencies between name spaces are widely regarded as design flaws. This motif describes a circular dependency through a path that starts at a vertex within a name space (in1),

---

3The formal definition of the grammar can be found here: [http://coda.google.com/p/gueryframework/source/browse/grammar/guery.g](http://coda.google.com/p/gueryframework/source/browse/grammar/guery.g)

4 Also known as packages
1 // subtype knowledge
2 motif stk
3 select type super
4 connected by inherits(type>super)[1,*] and uses(supertype)[1,*]
5 where "inherits .type= = 'extends ' | | inherits .type== 'implements ' and "uses.type== "uses "
6 group by "super"

Listing 1: Subtype knowledge

then goes through vertices in other name spaces (out1 and out2), returning to a vertex in the original name space (in2). Constraints are now defined on vertices with respect to their name space labels (line 4). The cardinality of the out and in path roles is set to 1, indicating that those paths are to be instantiated by a single edge. Those edges are exiting (out) and entering (in) the respective name space. Note that the cardinality constraint on the part of the path that connects the out vertices is [0,∗]. This means that an empty path is possible, and therefore the motif can be instantiated by a triangular pattern consisting of three vertices v

The third example (listing 3) represents a pattern where a client (directly) references a service specifica-
tion (listing 2) uses (super type) [1,∗] role. The circular dependency motif (listing 2) also uses one aggregation function (line 6). Not all path roles have length constraints, in this case the default [1,∗] is used.

5 Motif Aggregation

In our previous work [6], we have conducted an empirical study to investigate just how common antipat-
terns that can be represented by motifs are, in dependency graphs extracted from real world software. It turns out that the number of instances is generally very large. This makes our approach, as outlined so far, difficult to use in this target domain: engineers analysing the quality of software are overwhelmed by the large number of instances of a particular antipattern. However, it is very often the case that many motif instances are very similar to other instances, and are perceived by users as non-significant variations. This raises the following questions:

1. How can similarity of motif instances be formally defined?
2. How can similarity of instances be included in pattern definitions?
3. Can algorithms to find motif instances take advantage of instance similarity?

For all three motifs discussed above, the introduc-
tion of instance similarity makes sense for a software engineer who wants to use these motifs. For the sub-
type knowledge motif (listing 1), an engineer is often only interested in which super types reference their own subtypes. Hence, different instances binding the super role to the same vertex can be considered similar. For the circular dependency motif (listing 2), an engineer is often only interested in finding the packages that have circular dependencies. Here, different instances binding the in1 role to vertices within the same name space can be considered similar. Finally, for the abstraction without decoupling motif (listing 3), instances can be considered similar if they have identical bindings for client and the service roles, but not necessarily for the impl role.

This concept of similarity can be expressed by aggregation functions. Functions are applied to motif instances, and instances are considered similar if those functions yield the same value. This similarity relation is an equivalence relation, known as the equivalence kernel of the respective function(s).

Definition 7 (Aggregation Function) Let G = (V,E) be a directed graph, M a motif and INST the set of instances of M in G. An aggregation function is a mapping f : INST → VALUE that maps instances to values in some domain.

For a given set of aggregation functions \{f_i : INST → VALUE\}_i, we define \(\simeq \subseteq INST \times INST\) as follows: \(\text{inst}_1 \simeq \text{inst}_2\) iff \(f_i(\text{inst}_1) = f_i(\text{inst}_2)\) for all \(f_i\).

GUERY/DSL supports the definition of simple aggregation functions in the group by clause. In the subtype knowledge motif (listing 1), there is one function that maps a motif instance to the vertex instantiating the super role (line 6). The circular dependency motif (listing 2) also uses one aggregation function (line 6). This function maps the instance to the name space of the vertex instantiating the in1 (and therefore also the in2) role. Finally, the abstraction without decoupling motif (listing 3) uses two aggregation functions (line 6) - instances are considered similar if both the client and the service role are mapped to the same vertices.
6 Motif Detection

6.1 Main Algorithm

In this section we discuss some of the algorithms used by GUERY to find motifs in graphs. The algorithms are applicable to connected motifs.

The main algorithm (Algorithm 2) is based on depth-first constraint resolution. Bindings associating vertex roles with vertices and path roles with paths are maintained in two maps, vbindings and pbindings, respectively. The maps also keep track of positions for individual key-value pairs, and support a remove function that can be used to remove associations for a given position. The bind function adds new associations to a map, while the lookup function is used to query maps for values associated with a given key.

The algorithm first chooses an arbitrary initial role, \( r_0 \), with which analysis will start, and then uses the schedule function to arrange vertex constraints and path roles into an ordered list. Since the motif is connected, motif instances can be discovered by starting from bindings of an initial vertex role. Starting from this role, all other roles can be reached by either following the path roles directly, or by following them in reverse direction. This step can be seen as a query optimisation. It can be done statically, before the actual graph is analysed, and, therefore, does not depend on the size and the complexity of the actual graph. The main purpose is to avoid iterating over \( V \times V \) when instantiating path roles. Instead, we can always assume that, for each path role, either
1. Only the source vertex role is bound.
2. Only the target vertex role is bound.
3. Both the start and the target vertex role of the path role are already bound.
4. Only the source vertex role is bound.
5. Only the target vertex role is bound.

The algorithm first chooses an arbitrary initial role, \( r_0 \), with which analysis will start, and then uses the schedule function to arrange vertex constraints and path roles into an ordered list. Since the motif is connected, motif instances can be discovered by starting from bindings of an initial vertex role. Starting from this role, all other roles can be reached by either following the path roles directly, or by following them in reverse direction. This step can be seen as a query optimisation. It can be done statically, before the actual graph is analysed, and, therefore, does not depend on the size and the complexity of the actual graph. The main purpose is to avoid iterating over \( V \times V \) when instantiating path roles. Instead, we can always assume that, for each path role, either
1. Only the source vertex role is bound.
2. Only the target vertex role is bound.
3. Both the start and the target vertex role of the path role are already bound.
4. Only the source vertex role is bound.
5. Only the target vertex role is bound.

The algorithm first chooses an arbitrary initial role, \( r_0 \), with which analysis will start, and then uses the schedule function to arrange vertex constraints and path roles into an ordered list. Since the motif is connected, motif instances can be discovered by starting from bindings of an initial vertex role. Starting from this role, all other roles can be reached by either following the path roles directly, or by following them in reverse direction. This step can be seen as a query optimisation. It can be done statically, before the actual graph is analysed, and, therefore, does not depend on the size and the complexity of the actual graph. The main purpose is to avoid iterating over \( V \times V \) when instantiating path roles. Instead, we can always assume that, for each path role, either
1. Only the source vertex role is bound.
2. Only the target vertex role is bound.
3. Both the start and the target vertex role of the path role are already bound.
4. Only the source vertex role is bound.
5. Only the target vertex role is bound.
6.2 Using Aggregation

If only instance classes, modulo aggregation functions, are to be computed, Algorithm 2 can be significantly optimised. The key idea is to extract the vertex roles used in the aggregation functions, and try to resolve them first. This can be easily achieved by first selecting an initial vertex role from the set of roles used in aggregation functions, and then selecting the path roles in the scheduling function so that the motif graph is traversed in a way that visits those vertex roles first. The scheduling function also has to record the position in the rule list where all those vertex roles will have bindings. We refer to this number as the aggregation threshold. Once this is done, the back tracking section of the main algorithm can be rewritten to support back jumping: instead of reducing the position count by one, the position can be moved to the position just above the aggregation threshold, and all bindings below this point can be removed. This will make sure that the next instance computed will have a different binding for at least one of the vertex roles that occurs in the aggregation functions.

Note that this is no guarantee that only one instance per equivalence class will be computed. An example is the circular dependency motif (listing 2). Even if the in1 role is bound to different vertices, these vertices could still be in the same name space. However, the search space is significantly reduced. To consolidate results into classes, an extra reduce step may be necessary. A sufficient condition that no reduce step is needed is that all aggregation functions are injective. This is the case in the examples STK (listing 1) and AWD (listing 3) - in these motifs, the aggregation functions are instances of the identity function, mapping vertices to themselves.

6.3 Parallelisation

Parallelising Algorithm 2 is straightforward: search for different initial bindings can be executed in parallel. Using back jumping to optimise the search for classes of instances is still possible. The detailed algorithm is shown in Algorithm 3. The set of vertices is added to an agenda, and retrieved by threads as long as the agenda is not empty. Note that the query and element retrieval from the agenda (lines 14-15) must be synchronised (protected by a lock). Each thread has its own binding maps and position counter that are not shared with other threads.

Algorithm 3 For a given directed graph \( G = (V,E) \) and a motif \( M = (VR, PR, s, t, CV, CP) \), find all motif instances using \( N \) threads

1. choose initial role \( r_0 \in VR \)
2. agenda = empty stack
3. rules := schedule(M, r0)
4. count := 0
5. for all vertex in \( V \) do
6. push(agenda, vertex)
7. while count < \( N \) do
8. position := 0
9. vbindings = empty map
10. pbindings = empty map
11. thread = create_thread(job)
12. start(thread)
13. routine job
14. while \(|\text{agenda}| > 0\) do
15. next_vertex := pop(agenda)
16. bind(vbindings, \( r_0 \rightarrow \text{vertex,0} \))
17. resolve
18. end routine

6.4 Path Discovery

The main algorithm uses a function to find incoming and outgoing paths for a given vertex. The graph traversal strategy used to compute these paths can be configured, and is designed as a pluggable module in the implementation. By default, paths are computed through online computation using a simple breadth first traversal strategy. We have also implemented a more advanced strategy that caches paths. Naive caching of reachability information is not scalable as the size of the memory required to store reachability information is \(|V| \times |V|\). However, we have implemented a strategy that precomputes reachability between vertices and compresses the cache in two stages:

1. Compute the graph \( G_{SCC} \) consisting of the strongly connected components and their relationships using Tarjan’s algorithm [18].
2. Compute reachability for the (acyclic) graph \( G_{SCC} \) and store it using chain compression [8].

This strategy adds some initial overhead to the computation but can significantly improve the overall speed of the computation.

7 Validation

7.1 Implementation

We have implemented the algorithms discussed earlier as a Java class library “GUERY”. The library is based on JUNG2 [10], but uses special super classes for representing vertices and edges, respectively. The main purpose of these is to enable fast graph traversal, through vertices having direct references to incoming and outgoing edges. The correctness of the implementation is established through comprehensive test suites with high coverage.

We have used GUERY to validate thee aspects of the algorithms: the effectiveness of aggregation, the computation of partial result sets, and the benefits of using the parallel version of the algorithm with multicore processors.

7.2 Aggregation

The data we have used for validation are dependency graphs extracted from compiled Java programs by
Algorithm 2 (Motif Detection) For a given directed graph $G = (V, E)$ and a motif $M = (V_R, PR, s, t, C_V, C_P)$, find all motif instances.

1: choose initial role $r_0 \in V_R$
2: position := 0
3: rules := schedule(M, $r_0$)
4: for all vertex in $V$ do
5:   bind(vbindings, $r_0 \rightarrow$ vertex, 0)
6:   resolve
7: exit
8: routine resolve
9: if position=$|rules|$ then
10:   signal("instance found", vbindings, pbindings)
11: return
12: position := position+1
13: rule := rules[position]
14: if rule $\in C_V$ then
15:   instantiated_constraint := instantiate(rule, vbindings)
16: if check(instantiated_constraint) then
17:   resolve
18: else if rule $\in PR$ then
19:   source := lookup(vbindings, s(rule))
20:   target := lookup(vbindings, t(rule))
21: if exists(source) $\land$ exists(target) then
22:   path := find_first_path(source, target, $C_P$)
23:   bind(pbindings, role(rule) $\rightarrow$ path, position)
24:   resolve
25: else if exists(source) $\land$ exists(target) then
26:   paths := find_outgoing_paths(source, $C_P$)
27: for all path in paths do
28:   bind(vbindings, t(rule) $\rightarrow$ end(path), position)
29:   bind(pbindings, role(rule) $\rightarrow$ path, position)
30:   resolve
31: else if exists(target) $\land$ exists(source) then
32:   paths := find_incoming_paths(target, $C_P$)
33: for all path in paths do
34:   bind(vbindings, s(rule) $\rightarrow$ start(path), position)
35: bind(pbindings, role(rule) $\rightarrow$ path, position)
36:   resolve
37:   {backtrack}
38: remove(vbindings, position)
39: remove(pbindings, position)
40: position := position-1
41: end routine
means of byte code analysis. The programs used are the ten largest programs (measured by the number of vertices) from the Qualitas corpus [19], a collection of programs often used in experimental software engineering. The motifs used in the experiments are the motifs discussed earlier: AWD (listing 3), CD (listing 2) and STK (listing 1). Table 1 shows the respective graphs, identified by the names of the programs, and the number of vertices and edges. Table 2 shows the number of instances, and equivalence classes of instances modulo aggregation functions, found in those programs. Using classes instead of instances significantly reduces the search space. The average number of instances per class is 18.60 for AWD, 9759.25 for CD and 6.94 for STK. The number is particularly high for CD, the motif that uses a non-injective aggregation function.

<table>
<thead>
<tr>
<th>graph (program)</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>azureus-3.1.1.0.jar</td>
<td>6444</td>
<td>35392</td>
</tr>
<tr>
<td>jchempaint-2.0.12.jar</td>
<td>3342</td>
<td>17335</td>
</tr>
<tr>
<td>jrefactory-2.9.19.jar</td>
<td>2776</td>
<td>16007</td>
</tr>
<tr>
<td>rssowl-1.2.jar</td>
<td>2405</td>
<td>14249</td>
</tr>
<tr>
<td>jruby-1.0.1.jar</td>
<td>2093</td>
<td>11016</td>
</tr>
<tr>
<td>drjava-20050814.jar</td>
<td>2078</td>
<td>11916</td>
</tr>
<tr>
<td>weka-3.5.8.jar</td>
<td>2046</td>
<td>12230</td>
</tr>
<tr>
<td>sandmark-3.4.jar</td>
<td>1923</td>
<td>13574</td>
</tr>
<tr>
<td>jtopen-4.9.jar</td>
<td>1779</td>
<td>9936</td>
</tr>
<tr>
<td>hibernate-3.3.1.jar</td>
<td>1700</td>
<td>10093</td>
</tr>
</tbody>
</table>

Table 1: Size of graphs in data set

7.3 Stream Processing

To validate the benefits of the observer based API, we have measured the time needed to compute the first 10 instances and classes for the three motifs AWD, CD and STK and the first three graphs from our data set. For this experiment, we used a Power-Mac with an Intel Core 2 2.8 GHz Duo processor, the Java HotSpot(TM)64-Bit Server VM (build 14.3-b01-101, mixed mode), and a solver configured to use two threads. The results are shown in table 3. The first 10 results are usually returned in under 1 s and can be presented to the user for analysis. The main advantage is that instances are produced faster than users can analyse them, and that users do not have to wait for the entire set of instances to be computed.

Note that the computation of the first 10 classes is slower than the computation of the first 10 instances. In order to compute classes, the solver must traverse deeper branches of the derivation tree, starting with the nodes to where the solver back jumped.

7.4 Parallelisation

To assess the effectiveness of the parallel algorithm, we computed all instances and variants for the largest graph in the data set, extracted from azureus-3.1.1.0.jar byte code. We used a computer with two Intel Xeon E5440 2.83GHz quad core processors, using the Java SE 1.6.0_16-b-1 Linux 64-bit virtual machine. We executed the queries (motif searches) using different numbers of threads. The respective performance data are shown in figures 3 (for AWD), 4 (for CD) and 5 (for STK), respectively. The algorithm scales very well when using multi core processors, but performance flattens out if the number of threads used goes beyond 8 and even slightly increases for larger numbers of threads as the effects of the overhead related to inter-thread synchronisation kick in.
1. the use of aggregation that can be used to significan- tly speed up computation using a branch and bound technique (back-jumping). 2. the algorithm design that creates a results stream and allows users to query graphs in “google mode”, where the computation of the first few instances is very fast and often sufficient, and 3. the effective use of parallelisation that takes full advantage of modern multi-core processor architectures. We have validated our results using example graphs and queries from the domain of software analysis.

The optimisation of the algorithms based on defined aggregation is a unique feature of the GUERY software that is not available in similar packages such as GReQL and Crocopat. Also, these packages do not support multi-threaded processing of queries in their current versions.

The GUERY language we have developed to represent queries is rather simple and is optimised for use by people with a background in object-oriented programming. However, it does lack some of the expressiveness of GReQL, in particular the use of full quantification (we use only generalisation over edges in path constraints), and the use of path expressions. The path expressions in GReQL allow the user to express composite paths using a syntax similar to regular expressions. In GUERY, each part of such a composite path, and the respective source and target vertex roles, must be explicitly defined. We think that this is useful and often necessary anyway, as the respective role names must be used to retrieve the instantiating vertices and edges from the motif instance.

While the graphs we have analysed are fairly large, there are plenty of domains where much larger graphs have to be analysed. An example is the data sets extracted from web data and used in the billion triple dollar challenge [2]. To query data sets of this size, the respective graphs must be partitioned. We believe that the parallelisation technique we have used to optimise performance on multi-core processors can be adapted to execute queries on those partitioned graphs using clusters of networked computers, using a
map and reduce [5] style algorithm. Further research is necessary to explore this.

References


