Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection

Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen

Electrical and Computer Engineering Department, Iowa State University, USA

Abstract. Structure-oriented approaches in clone detection have become popular in both code-based and model-based clone detection. However, existing methods for capturing structural information in software artifacts are either too computationally expensive to be efficient or too light-weight to be accurate in clone detection. In this paper, we present Exas, an accurate and efficient structural characteristic feature extraction approach that better approximates and captures the structure within the fragments of artifacts. Exas structural features are the sequences of labels and numbers built from nodes, edges, and paths of various lengths of a graph-based representation. A fragment is characterized by a structural characteristic vector of the occurrence counts of those features. We have applied Exas in building two clone detection tools for source code and models. Our analytic study and empirical evaluation on open-source software show that Exas and its algorithm for computing the characteristic vectors are highly accurate and efficient in clone detection.

1 Introduction

The habit of copy-and-paste in programming is one of the sources for similar fragments of code in many software systems. Such fragments are called *clones*. It is found that cloning also occurs in model-based software development (MBD). Clones create difficulties for software maintenance, for example, changes to a cloned fragment must be carried out in several other places in the codebase.

Many approaches have been proposed to detect clones in traditional software as well as in model-based software. Among them, *structure-oriented* approaches are the most popular and successful in both code-based and model-based clone detection [11,25]. In structure-oriented approaches, software artifacts including source code and models are represented as tree-based and/or graph-based data structures. For example, abstract syntax trees (ASTs) or parse trees are used for source code and attributed, directed graphs are used for program dependence graphs (PDGs), call graphs, and models in MBD.

In those approaches, the common detection process include (1) modeling the artifacts and their fragments (i.e. potential clone parts) in the form of a structure-oriented representation, (2) extracting features of each fragment from its representation, (3) computing the similarity between fragments using a similarity

```
int sum(int l,h) {
  int s = 0;
  for(int i=l; i<h; i++)
    if (i%2 == 0)
      s = s + i;
  return s;
}
```

```
int power(int x,n) {
  int p = 0;
  if(x != 0) {
    p = 1;
    for(int i=1; i<n; i++)
      p = p * x;
  }
  return p;
}
```

Fig. 1. Different fragments with similar occurrence-count vectors of single node types

measure on the features, and (4) grouping similar fragments into clone groups. The phases (3) and (4) involve the comparison of the features extracted from the structure-oriented representation of fragments in phase (2). The key feature in those structure-oriented approaches is the internal structure of a (code or model) fragment. However, the methods for comparing tree-based or graph-based structures, such as tree editing distance measurement [1] or graph isomorphism [3], are computationally expensive for scalable and efficient clone detection.

To avoid that high complexity, several light-weight approaches for clone detection have been proposed [4,5,6,7,8]. However, the accuracy is not satisfactory. Methods for extracting structural features in artifacts are still simple. One of the state-of-the-art light-weight approaches, Deckard [8], uses a vector-based approach to extract the structural information in an AST or a parse tree. It proposes to use the occurrence counts of each q-level complete binary subtree in such a tree as characteristic vectors. However, for $q > 1$, this method does *not* work for *graph-based* representations. For $q = 1$ (the case that Deckard tool is implemented), a fragment is characterized by a vector in which each element is the occurrence counts of single AST node types. However, the occurrence counts of only single node types is insufficient to capture well structural information.

The illustrated example in Figure 1 shows two code fragments. They quite differ from each other because they have very different *nesting structures* of program constructs (e.g. a "for" encloses an "if" and vice versa). In addition, in two methods, the *sequential structures* (i.e. the orders) of statements are also different, despite that each contains a declaration, a "for", an "if", and a "return" statement. Unfortunately, the above two code fragments have very similar Deckard representation vectors since they have similar numbers of occurrences of single AST node types, e.g., method and variable declarations, "if" and "for" statements, expressions, literals, simple names, etc. Thus, two fragments would be incorrectly detected by Deckard's vector-based approach.

In brief, existing methods for capturing structural information in software artifacts for clone detection are either too computationally expensive to be efficient or too light-weight to be accurate. Importantly, no light-weight approach has been proposed for graph-based structure. In this paper, we present a novel approach, Exas, for better approximating the structure as a feature of tree-based and graph-based representations in software artifacts. At the same time, Exas achieves high levels of both accuracy and efficiency. In Exas, features are the sequences of labels and numbers built from nodes, edges, and paths of various lengths in a generic graph-based representation. A *structural* characteristic vector for a fragment contains the occurrence counts of all kinds of its features.

Our analytical study shows that in Exas, for the graph-based representation, isomorphic (sub)graphs have the same structural characteristic vectors. For the graph-based representation, the distance of Exas vectors of two (sub)graphs is bounded by their graph editing distance. That is, if two (sub)graphs are considered clones (i.e. having a small editing distance), their vector distance is small as well. This result also holds for trees and tree editing distance.

We also developed two clone detection tools using Exas: one for code (tree-based representation) and one for Simulink models (graph-based representation). Our empirical study shows that with Exas, the clone detection result is 3-12% more accurate than that of Deckard, the state-of-the-art vector-based approach, with less than a few seconds more in running time. Additionally, the result for model clones is also highly precise. The study shows that our vector computation algorithm is more time-efficient than the canonical labeling, the state-of-the-art method for graph isomorphism. The contributions of this paper include:

1. *Exas*: A novel, accurate and efficient characteristic feature extraction approach for clone detection in structure-based software artifacts. It provides a good approximation of a structure in a program or a model,
2. *An efficient algorithm* to compute Exas vectors,
3. *Two applications of Exas* in two clone detection tools for code and models,
4. *Analytical* and *empirical studies* of the accuracy and efficiency of Exas.

Section 2 presents Exas approach and an analytic study. Section 3 describes an efficient algorithm to compute Exas vectors. Section 4 discusses the application of Exas in two clone detection tools for code and models. Section 5 presents our empirical evaluation. Related work is in Section 6. Conclusions appear last.

2 Exas Approach

2.1 Structure-Oriented Representation

In our structure-oriented representation approach, a software artifact is modeled as a *labeled, directed graph* (tree is a special case of graph), denoted as $G = (V, E, L)$. V is the set of nodes in which a node represents an element within an artifact. E is the set of edges in which each edge between two nodes models their relationship. L is a function that maps each node/edge to a label that describes its attributes. For example, for ASTs, node types could be used as nodes' labels. For Simulink models, the label of a node could be the type of its corresponding block. Other attributes could also be encoded within labels. In existing clone detection approaches, labels for edges are rarely explored. However, for general applicability, Exas supports the labels for both nodes and edges.

The purpose of clone detection is to find cloned parts in software artifacts. Potential cloned parts in a software artifact are called *fragments*. In our approach,

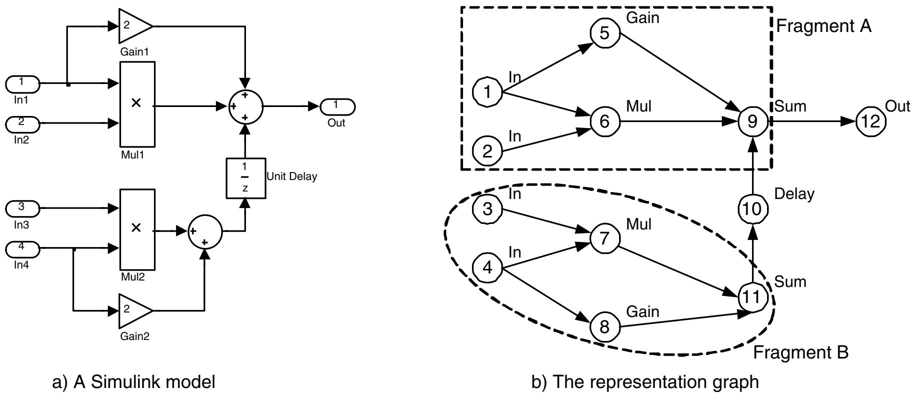

Fig. 2. An example: numbers are the indexes of nodes, texts are their labels

a fragment within a tree-based software artifact is considered as a subtree of the representation tree. For a graph-based software artifact, a fragment is considered as a weakly connected sub-graph in the corresponding representation graph.

Figure 2 shows an illustrated example of a Simulink model, its representation graph and two cloned fragments A and B.

2.2 Structural Feature Selection

Exas focuses on two kinds of *patterns* of structural information of the graph, called (p, q)*-node* and n*-path.*

A (p, q)**-node** is a node having p incoming and q outgoing edges. The values of p and q associated to a certain node might be different in different examined fragments. For example, node 9 in Figure 2 is a (3,1)-node if entire graph is currently considered as a fragment, but is a (2,0)-node if fragment A is examined.

An n**-path** is a directed path of n nodes, i.e. a sequence of n nodes in which any two consecutive nodes are connected by a directed edge in the graph. A special case is 1-path which contains only one node.

Structural feature of a (p, q)-node is the label of the node along with two numbers p and q. For example, node 6 in fragment A is $(2, 1)$-node and gives the feature `mul-2-1`. *Structural feature* of an n-path is a sequence of labels of nodes and edges in the path. For example, the 3-path 1-5-9 gives the feature `in-gain-sum`. Table 1 lists all patterns and features extracted from A and B. It shows that both fragments have the same feature set and the same number of each feature. Later, we will show that it holds for all isomorphic fragments.

2.3 Characteristic Vectors

An efficient way to express the property *"having the same or similar features"* is the use of vectors. The characteristic vector of a fragment is the occurrence-count vector of its features. That is, each position in the vector is indexed for

Table 1. Example of Patterns and Features: numbers are the indexes of nodes and texts are features

Pattern	Features of fragment A					Features of fragment B				
1-path	1 in	2 in	5 gain	6 mul	9 sum	4 in	3 in	8 gain	7 mul	11 sum
2-path	1-5 in-gain	1-6 in-mul	2-6 in-mul	6-9 mul-sum	5-9 gain-sum	4-8 in-gain	4-7 in-mul	3-7 in-mul	7-11 mul-sum	8-11 gain-sum
3-path	1-5-9 in-gain-sum		1-6-9 in-mul-sum		2-6-9 in-mul-sum	4-8-11 in-gain-sum		4-7-11 in-mul-sum		3-7-11 in-mul-sum
(p,q)-node	1 in-0-2		2 in-0-1		5 gain-1-1	4 in-0-2		3 in-0-1		8 gain-1-1
(p,q)-node (continued)	6 mul-2-1		9 sum-2-0			7 mul-2-1		11 sum-2-0		

Table 2. Example of Feature Indexing. Based on the occurrence counts of features in fragment A, the vector for A is (2,1,1,1,1,2,1,1,1,2,1,1,1,1,1).

Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts
in	1	**2**	in-gain	5	**1**	in-gain-sum	9	**1**	gain-1-1	13	**1**
gain	2	**1**	in-mul	6	**2**	in-mul-sum	10	**2**	mul-2-1	14	**1**
mul	3	**1**	gain-sum	7	**1**	in-0-1	11	**1**	sum-2-0	15	**1**
sum	4	**1**	mul-sum	8	**1**	in-0-2	12	**1**			

a feature and the value at that position is the number of occurrences of that feature in the fragment. Table 2 shows the indexes of the features, which are global across all vectors, and their occurrence counts in fragment A.

Two fragments having the same feature sets and occurrence counts will have the same vectors and vice versa. The vector similarity can be measured by an appreciably chosen vector distance such as 1-norm distance.

Note that 1-paths are equivalent to 1-level binary subtrees used in Deckard tool. Therefore, Deckard vector of a fragment is a part of the Exas vector for that fragment. For example, Deckard vector for fragment A would be (2,1,1,1). In other words, Exas uses more features. It implies that the vector distance between Deckard vectors of two fragments is no larger than that of Exas vectors. It also implies that Exas vector distance has better discriminative characteristic, i.e. is more accurate in measuring the fragments' similarity. These are also true when applying for tree structures. For example, in the illustrated example (Section 1), Exas vectors are able to distinguish two code fragments because Exas can better approximate the nesting and sequential structures of program elements.

2.4 Analytical Study

Given two (sub)graphs G and G'. Let V and V' be their vectors, respectively, d be the maximum degree of all nodes of all (sub)graphs, and N be the maximum size of all n-paths.

Lemma 1. *The number of n-paths containing a node is at most $P = \sum_{n=1}^{N} n.d^{n-1}$ and that of n-paths containing an edge is at most $Q = \sum_{n=2}^{N} n.d^{n-2}$.*

Due to space limit, we do not provide the proof for Lemma 1 since it is easy to be verified.

For brevity, we call n-paths and (p, q)-nodes *instances.* Let S and S' be the sets of instances of G and G', respectively. If G is edited to be G', S' is updated accordingly from S by removing some instances and/or inserting others. Let us call those removed and inserted instances as *"affected instances"*.

Lemma 2. *k graph editing operations affect at most $(2P + 4)k$ instances.*

Proof. We consider four types of graph editing operations: removing an edge, inserting an edge, relabeling a node, and relabeling an edge.

Removing (inserting) an edge removes (inserts) all n-paths containing it and replaces two (p, q)-nodes at its two ends with two new (p, q)-nodes. This replacement affects four instances. Thus, the total number of affected instances is at most $Q + 4$. Relabeling a node replaces its corresponding (p, q)-node and all n-paths containing it with new instances, thus, affects at most $2 + 2P$ instances. Similarly, relabeling an edge affects at most $2Q$ instances since no (p, q)-node is affected. In all cases, the total number of affected instances is at most $2P + 4$. Therefore, k editing operations affect at most $(2P + 4)k$ instances.

Lemma 3. *If there are M affected instances, $\|V - V'\|_1 \leq M$.*

Proof. If an instance is removed (inserted), the occurrence counts of its feature reduce (increase) by one. Since there are M affected instances, V' is obtained from V by the total of M units of such increment and/or decrement. Since $\|V - V'\|_1$ is the total differences of occurrence counts between V and V', it is at most M.

Two above lemmas imply the following theorem.

Theorem 1. *If graph edit distance of G and G' is k, $\|V - V'\|_1 \leq (2P + 4)k$.*

We could consider two isomorphic graphs as having the editing distance of zero. Therefore, applying Theorem 1, we have the following corollary.

Corollary 1. *If G and G' are isomorphic, they have the same vector,i.e. $V = V'$.*

The results can be applied directly to (sub)trees. However, tree editing distance can be defined in a different set of operations. The following results are for the case in which G and G' are (sub)trees and tree editing operations include relabeling, inserting, and deleting a node.

Lemma 4. *The number of n-paths containing a node in a (sub)tree is at most $R = \sum_{n=1}^{N} \sum_{i=1}^{n} d^{i-1}$.*

Lemma 5. *k tree editing operations affect at most $(2R + 3)k$ instances.*

Proof. Relabeling a node affects at most $2R+2$ instances (see the proof of Lemma 2). Removing a node u, i.e. connecting all its children to its parent v, removes all n-paths containing u, inserts some n-paths containing v, replaces (p, q)-node at v with a new one, and removes (p, q)-node at u. Thus, the total number of affected instances is $2R + 3$. Similar argument is applied for the case of inserting a node u. Therefore, a single tree editing operation affects at most $(2R + 3)$ instances, thus, k operations affect at most $(2R+3)k$ instances.

From Lemmas 4 and 5, we have the following theorem.

Theorem 2. *If tree editing distance of G and G' is k,* $\|V - V'\|_1 \leq (2R+3)k$.

2.5 Implications in Clone Detection

The aforementioned important properties of Exas characteristic vectors imply that they are very useful in the problems involving graph isomorphism or tree/graph similarity, especially in structure-oriented clone detection.

State-of-the-art graph-based clone detection approaches [11,32] require graph-based cloned fragments to be isomorphic. With **Corollary 1**, instead of checking isomorphism of two (sub)graphs, we could compare their Exas characteristic vectors to find cloned (sub)graphs. That corollary guarantees that all clone pairs will be detected. However, it is not a sufficient condition for absolute precision, i.e. two (sub)graphs with the same vectors might be non-isomorphic since nodes and edges which cannot be mapped between two (sub)graphs can make up n-paths or (p, q)-nodes with the same feature. Other criteria should be used along with Exas to increase the precision of detected results.

Theorem 1 shows that our approach is also useful for the problems involving graph editing distances such as similarly matched clone detection in graph-based representations or graph similarity measurement.

Theorem 2 is useful for clone detection approaches based on tree editing distance, i.e. two tree-based fragments are considered clones if their editing distance is smaller than a chosen threshold k. For a set of fragments, we can always find a common value R for any two fragments. Then, the vector distance of any two cloned fragments will be less than $(2R + 3)k$. In other words, the 1-norm distance of Exas characteristic vectors could be used as a necessary condition: to be a cloned pair, two fragments must have the distance of their vectors smaller than a chosen threshold $\delta = (2R + 3)k$. Of course, a small vector distance does not imply a small tree editing distance, i.e. this condition is not sufficient.

In our empirical evaluation (Section 5), the precision of only Exas characteristic vectors is evaluated for both tree-based and graph-based clone detection.

3 Vector Computing Algorithm

In this section, we describe an efficient algorithm to calculate Exas vectors from the structure-oriented representation. The key idea is that *the characteristic vector of a fragment is calculated from the vectors of its sub-fragments.* Of course, a node is the smallest fragment and its vector is calculated directly.

3.1 Key Computation Operation: *incrVector*

The key operation in our algorithm, *incrVector*(), is the computation of the vector for a fragment $g = f + e$ (i.e. g is built from f by extending f with an

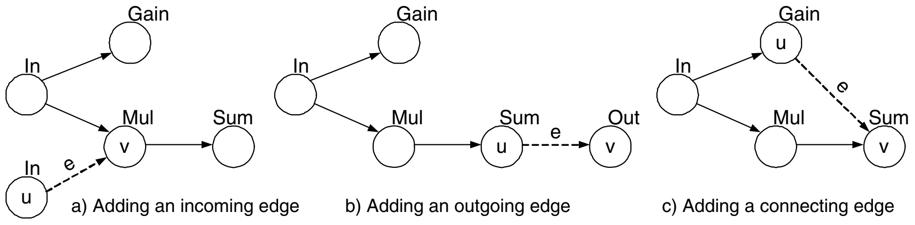
Gain
In
Mul
Sum
In
e
u
v
a) Adding an incoming edge
Gain
In
Mul
Sum
Out
e
u
v
b) Adding an outgoing edge
Gain
u
In
e
Mul
Sum
v
c) Adding a connecting edge

Fig. 3. Three cases of adding an edge to a fragment

edge e), given e and f (along with its vector) as inputs. In brief, the vector of g is derived from that of f by updating it with the occurrences of all new features of g created by the addition of the edge e into f.

Since we consider only weakly connected components as fragments, at least a node of e must belong to f. Let $e = (u, v)$. There are three following cases:

Case 1: incoming-edge, i.e. $u \notin f$ and $v \in f$. In this case, u is a newly added node. New features are created from the 1-path u, the 2-path $u - v$, the new $(0, 1)$-node at u. The (x,y)-node at v is replaced by the new $(x + 1, y)$-node because of the new incoming edge. All new n-paths of f will have the first node of u and the second node of v. Therefore, they are generated by adding u to the first of all $(n - 1)$-paths starting from v. These $(n - 1)$-paths can be achieved by a depth-first search (DFS) *within fragment* g from node v to the depth of $n - 2$.

Case 2: outgoing-edge, i.e. $u \in f$ and $v \notin f$. The situation is similar. However, new n-paths are generated from $(n - 1)$-path ending at u, i.e. DFS needs to expand in backward direction. Furthermore, (x, y)-node at v is replaced by a new $(x, y + 1)$-node.

Case 3: connecting-edge, i.e. both u and v were already in f. In this case, new n-paths are generated by the combination of any i-path ending at u (DFS in backward direction) and an j-path starting from v (DFS in forward direction), for all $i + j = n$. Both (x, y)-nodes at u and v are replaced by new (x', y')-nodes.

Time Complexity and Improvement. Assume that d is the maximum degree of the nodes and N is the maximum length of n-paths of interest. The number of n-paths searched by DFS is $O(d^{N-2})$ in all three cases (in the 3^{rd} case, two DFSs from u and v to level $n - 2$ are sufficient to find all those x-paths and y-paths). This seems to be exponential. However, instead of extracting features from n-paths of all sizes, we just extract features from *short* n-paths, i.e. n-paths having at most N nodes. This gains much efficiency and reduces little precision. In our experiments, in most subject systems, $N = 4$ gives the precision of almost 100% (Section 5). Moreover, in practice, representation graphs are generally not very dense, i.e. d is small. Thus, $O(N.d^{N-2})$ is indeed not very time-consuming.

3.2 Vector Computation for All Fragments in a Graph

Using *incrVector* operation, Exas calculates the vector of any individual fragment by starting from one of its nodes, adding one of its edges, then computing the vector, and so on. Thus, time complexity of computing vector for a fragment is $O(m.N.d^{N-2})$, with m as the fragment's size, i.e. the number of edges.

For the clone detection problem, the goal is to calculate the vectors for all potential cloned fragments in a graph. Generating all of its sub-graphs and then calculating their vectors as for individual fragments will not take advantage of *incrVector* operation. A more efficient approach is to generate the fragments with the increase in size by adding edge-by-edge and then to calculate the vector for the larger fragment from the vectors of the smaller ones.

However, the number of sub-graphs of a graph is exponential to its size. To increase efficiency, if graph isomorphism is used as a clone condition, we can take advantage of the following fact: to be a clone, a fragment should contain a smaller cloned fragment, i.e. two large isomorphic graphs should contain two smaller isomorphic sub-graphs.

Let C_k be the set of all cloned fragments of size k (i.e. with k edges). Observe that: (1) every fragment of size k can be generated from a fragment of size $k-1$ by adding a relevant edge; and (2) if two fragments of size k are a clone pair (isomorphic), there exists two cloned fragments of size $k-1$ within them, i.e. every clone pair of C_k can be generated from a clone pair of C_{k-1}.

Those facts imply that C_k can be generated from C_{k-1} by following steps: (1) extending all cloned fragments in C_{k-1} by one edge to have a candidate set D_k, (2) calculating vectors for all fragments in D_k by the *incrVector* operation, (3) grouping D_k into clone groups by characteristic vectors (i.e. all fragments in a group must have the same characteristic vectors), and (4) adding only the cloned fragments in D_k into C_k. By gradually generating C_0, C_1, C_2,..., C_k,..., we can find all cloned fragments precisely and completely. Note that, this strategy reduces significantly time complexity for fragment generation and vector computation for sparse graphs. In worst case (such as for a complete graph), time complexity is still exponential. More details can be found in another paper [12].

Vector Calculation for All Fragments in a Tree. For tree-based representations, a fragment is represented by a subtree. Since each node is the root of a subtree, i.e. each fragment corresponds to a node, the generation process is not needed. To compute the vectors for all subtrees in a tree, Exas traverses it in post-order. When a root p of a subtree $T(p)$ is traversed, the vector for $T(p)$ will be calculated as follows. Assume that $c_1, c_2, ..., c_k$ are the children of p, connecting from p by edges $e_1, e_2, ..., e_k$. Because of the post-order traversal, the vectors of the sub-trees $T(c_1), T(c_2), ..., T(c_k)$ are already calculated. Adding edge e_i to subtree $T(c_i)$ using *incrVector* operation gives the vector V_i for each branch. Then, the vector of $T(p)$ is derived from all vectors $V_1, V_2, ..., V_k$. By this strategy, time for computing vectors for all fragments of a tree is just $O(m.N.d^{N-2})$.

3.3 Vector Indexing and Storing

The potential number of features is huge. For example, if the number of different labels of nodes is L_v and that of edges is L_e, the total number of potential features generated from all n-paths of size no longer than N is $\sum_{n=1}^{N} L_v^n.L_e^{n-1}$. However, in practice, the actual features encountering in certain graph modeling for a software artifact is much smaller because there are not all combinations of nodes and edges that make sense with respect to the semantics of elements in artifacts.

Our experiment confirmed this fact. We conducted an experiment with WCD-MALIB, a real-world model-based system with 388 nodes and $L_v = 107$ labels ($L_e = 0$). With $N = 4$, the actual features encountered in the whole model is only 381, although the number of all possible features is more than 107^4.

More importantly, most fragments do not contain all features, especially small fragments. Thus, the *characteristic vectors are often sparse.* To efficiently process sparse characteristic vectors, a hashmap H is used to map between features and their indexes (i.e. their positions in vectors for storing their occurrence counts). H is global and used for all vectors. During vector calculation, if a feature has never been encountered before, it will be added into H with a next (increasing) available index. The vector of a fragment is also stored as a hashmap that maps the index of each feature into its corresponding counting value in the vector.

4 Applications of Exas

To demonstrate the usefulness of our structural feature extraction approach in clone detection, we implemented two tools. The first one, ClemanX, is for tree-based software artifacts. The second one, GemScan, is for graph-based artifacts.

4.1 ClemanX

ClemanX is a clone group management tool based on Cleman framework [9]. One key task of ClemanX is to detect clone groups of code fragments of a software project. Each source file managed by ClemanX is parsed and represented in ClemanX as an AST. The label of each node of the tree is its AST node type such as *class, method declaration, block, for statement, etc.* No label for edges is used. Each fragment is represented by a sub-tree of the AST. ClemanX uses Exas to extract the structural features in fragments as described in Section 3.

ClemanX detects not only exact-matched but also similar-matched code clones. Therefore, clone condition is defined based on the 1-norm distance of characteristic vectors: *two fragments are considered clones if their relative similarity* $r = 1 - \frac{2\|v_1 - v_2\|_1}{\|v_1\|_1 + \|v_2\|_1}$ *is greater than a chosen threshold* where v_1 and v_2 are their vectors. Relative similarity allows larger clones to have greater differences.

4.2 GemScan

GemScan is a clone detection tool for Simulink models [10]. The transformation of Simulink models into graphs in GemScan is carried out in the same manner as in CloneDetective [11]. Basically, it consists of three tasks: (1) *parsing* the model into a directed graph in which a node represents a block and an edge represents a signal connection, (2) *flattening* the subsystems, and (3) *labeling* nodes/edges with the labels depending on their attributes. The output is *a labeled, directed graph* where the set of nodes V represents Simulink blocks, the set of directed edges E represents the signal lines, and the labeling function L assigns labels to nodes and edges. Cloned fragments of a model are *weakly connected, non-overlapping, isomorphic sub-graphs* of the representation graph. Fragment generation and vector calculation in GemScan are described in Section 3.

5 Empirical Evaluation

Our experiment mainly focuses on the *accuracy* and *efficiency* of our structural feature extraction approach. We also evaluate the trade-off between those qualities and the limit length of extracted features.

To use ClemanX and GemScan for evaluating Exas, they are configured to use only Exas structural features as a sole criteria for detection. The efficiency is evaluated by the time of fragment generation and vector computation. The accuracy is evaluated by checking the precision of the detected clones, i.e. the ratio between the number of correct clone groups and the total number of clone groups. A group is considered to be correctly detected if all pairs of member fragments in that groups are clones. Clone groups detected by GemScan are checked by *canonical labeling* method, the state-of-the-art technique for checking graph isomorphism. Clone results of ClemanX are checked manually because it detects similar code clones. All experiments are conducted on a computer with AMD Athlon 64 X2 Dual Core 5200+ 2.70GHz,1.50GB RAM, and Windows XP.

5.1 Clone Detection on Graph-Based Artifacts

GemScan was run on three Simulink systems with different maximum sizes N of used n-paths. For example, $N = 4$ means that only n-paths with at most four nodes are processed. The number of generated fragments, clone groups, used features, and time for fragment generation and vector calculation (*FTime* in seconds) were reported. We also wanted to compare our approach with canonical labeling, one of the fastest techniques for checking graph isomorphism. Therefore, the canonical labeling module was run to get *Ctime*, time for generating canonical labels of all generated fragments. Those generated canonical labels were then also used to check the correctness of GemScan's detected clone groups.

Table 3 shows the result. *MSize* is the maximum size of generated fragments. The result shows that our approach is very fast (less than a second) and *thousands times faster* than the canonical labeling method. However, that level of efficiency does not sacrifice much precision. Especially when $N = 4$, the precision reaches 97-100% in almost all cases. Moreover, the precision is increased as the maximum size N of n-paths increases, i.e. extracting longer features achieves higher precision. This implies that Exas is accurate and efficient. Our experiment to evaluate Exas in similar-matched graph-based clone detection is in [12].

5.2 Clone Detection on Tree-Based Artifacts

ClemanX was run on six open-source projects of different sizes from medium to very large ones. Time was measured as before. Table 4 shows the result of

Table 3. Feature extraction time and Precision of GemScan

System	N	Fragment	Feature	MSize	FTime	CTime	Group	Correct	Precision
WCDMALIB	1	11597	139	9	0.84	3135	361	352	98%
388 nodes,410 edges	2	11415	314	9	1.00	3148	355	355	100%
107 labels	4	11391	382	9	1.10	3151	355	355	100%
Simulink_labs	1	4551	58	13	0.52	1017	739	720	97%
452 nodes, 415 edges	2	4492	181	13	0.64	1080	741	722	97%
39 labels	4	4492	334	13	0.85	1107	729	729	100%
Multiuav	1	7439	78	34	0.91	2000	542	490	90%
471 nodes, 573 edges	2	7316	238	34	0.92	2012	520	496	95%
52 labels	4	7276	465	34	1.00	1966	514	501	97%

Table 4. Feature extraction time of ClemanX

Log4J 1.2.14 (41 kLOC)				jEdit 4.2 (141 kLOC)				Axis 1.4 (227 kLOC)			
N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime
1	97	59	2.5	1	379	59	6.1	1	717	59	10.0
2	97	128	2.6	2	379	139	6.4	2	717	134	10.1
4	97	823	2.8	4	379	1388	7.0	4	717	1195	10.4
jFreeChart1.0.6(270kLOC)				JDK6 (3972 kLOC)				Eclipse3.2 (8318 kLOC)			
N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime
1	689	59	9.5	1	10,114	59	131	1	28,848	59	337
2	689	133	9.8	2	10,114	145	135	2	28,848	145	351
4	689	1007	10.1	4	10,114	2176	141	4	28,848	2302	361

running ClemanX on those systems. Its columns are similar to those of Table 3. The number of fragments (*kFrag.*) is shown in thousand units, e.g. the number of fragments in Eclipse3.2 is about 28,848,000. Those numbers (also equal to the number of AST nodes) do not change as N is varied because all fragments (i.e. all subtrees in an AST) are processed, regardless of the size of used features.

The result shows that ClemanX performed feature extraction very fast. It can scale up to very large projects with millions lines of code. For example, for Eclipse3.2 with more than 8 millions LOCs and 28 millions fragments, it took only about 6 minutes. Table 4 shows that Exas is very efficient and scalable.

ClemanX detects similar-matched code clone. Because the similarity of code clones is subjective, we have to check the precision of ClemanX manually. The checking was done on 100 randomly selected groups from the clone report for jFreeChart 1.0.6 on each experiment of N. Table 5 shows the result for two different choices for the threshold of similarity (TGroup is the total number of detected clone groups, CGroup is the number of checked groups). As we can see, the precision is very high. In addition, the longer the features are, the higher the precision is. The precision of 100% can be also achieved with $N = 4$.

Remind that for $N = 1$, the feature extraction of our approach is equivalent to Deckard (with $q = 1$). Running on the example in Section 1 and examining closely the fragments from the clone reports, we found that many fragments are wrongly reported as clones when $N = 1$, but are correctly reported as

Table 5. Precision of ClemanX by manual checking

threshold = 0.95					threshold = 0.90				
N	TGroup	CGroup	Correct	Precision	N	TGroup	CGroup	Correct	Precision
1	547	100	97	97%	1	2042	100	88	88%
2	528	100	99	99%	2	2188	100	95	95%
4	532	100	100	100%	4	1904	100	100	100%

```
public List getCategoriesForAxis(CategoryAxis axis){
  List result=new ArrayList();
  int axisIndex=this.domainAxes.indexOf(axis);
  List datasets=datasetsMappedToDomainAxis(axisIndex);
  Iterator iterator=datasets.iterator();
  while (iterator.hasNext()) {
    CategoryDataset dataset=(CategoryDataset)iterator.next();
    for (int i=0; i < dataset.getColumnCount(); i++) {
      Comparable category=dataset.getColumnKey(i);
      if (!result.contains(category))
        result.add(category);
    }
  }
  return result;
}
```

```
public List getCategories(){
  List result=new java.util.ArrayList();
  if (this.subplots != null) {
    Iterator iterator=this.subplots.iterator();
    while (iterator.hasNext()) {
      CategoryPlot plot=(CategoryPlot)iterator.next();
      List more=plot.getCategories();
      Iterator moreIterator=more.iterator();
      while (moreIterator.hasNext()) {
        Comparable category=(Comparable)moreIterator.next();
        if (!result.contains(category))
          result.add(category);
      }
    }
  }
  return Collections.unmodifiableList(result);
}
```

Fig. 4. A real example where N=1 could not capture nesting and sequential structures

non-clones when $N = 2$ or $N = 4$. Figure 4 shows such two fragments in a real case study with different nesting and sequential structures of program elements. In those real examples, our approach is more accurate than Deckard approach. The feature extraction time for $N = 4$ is only a couple tens of seconds longer than that of Deckard (i.e. when $N = 1$).

6 Related Work

Code clone detection approaches can be classified based on the representation of extracted features [2,25]. Text-based approaches [27,28] consider two code fragments as clones if their constituent texts match. Token-based approaches [5,6,29] view a code fragment as a sequence of program tokens. Similar or exactly matched sequences of tokens signify clones. Basit *et al.* [24] use suffix array on the token sequence. No structural features are extracted in text-based and token-based approaches. Tree-based approaches [1,4,7,30,31] represent a code fragment as a subtree in either an AST or a parse tree. Subtrees with similar extracted features are detected as clones. Kontogiannis *et al.* [1] use tree editing distance. Baxter *et al.* [4] compute the similarity between two subtrees in an AST by the ratio between the number of shared nodes and the total. No other structural information is used. In DMS [30], graph transformation and rewriting rules were applied. Wahler *et al.* [31] represent a Java program in XML and detect clones using frequent itemsets mining technique. Evans *et al.* [7] count the number of AST nodes, characters, tokens, LOCs in a fragment. Koschke *et al.* [26] use suffix trees on AST. Fluri *et al.* [23] propose a tree differencing algorithm for ASTs via matching nodes with a minimum edit script.

Deckard [8], a state-of-the-art tree-based approach, extracts characteristic vectors from parse trees by counting q-level binary subtree patterns. Its approach with $q = 1$ is equivalent to Exas with only 1-paths. When $q > 1$, the approach does not work for graphs. To detect semantic clones from Program Dependence Graph (PDG), it maps a subgraph into a forest of subtrees of the program's ASTs, and uses that technique for vector computation. Yang *et al.* also represent a tree by a set of q-level binary subtrees [14]. They transform a general tree to a binary tree and build a profile of all binary subtrees having the depth of q. This approach of q-level binary branches cannot be extended to support graphs.

For graph-based representations, existing approaches are too heavy. Komondoor and Horwitz [32] detect code clones in PDGs using subgraph isomorphism. To support similar clones in PDGs, program slicing is applied. In Datrix [33], to compare control or data flow graphs, a variety of software metrics are used including the number of arcs, loops, nodes, exits in a function, independent paths, etc. The state-of-the-art tool for clone detection in graph-based models is CloneDetective [11]. In CloneDetective, the structural feature extraction for clone detection is graph isomorphism. CloneDetective avoids graph isomorphism computation via its heuristic approach. Liu *et al.* [16] detect clones in a UML sequence diagram by representing it as an array of elements, and structural feature is a suffix tree. There has been much research on the methods for similar/exact matching of subgraphs [15]. However, similar to canonical labeling [3] for graph isomorphism, those approaches are too heavy to apply for clone detection.

There are also many approaches to support model evolution including the detection of differences between models [17,18,19], the merging of different models or different versions [19,20], and the management of consistency model changes [21]. The approaches in [17,18] represent a UML diagram as a tree. Those methods share a common strategy in which they try to match nodes from one graph to another via the matching of node labels and only the *local* connectivity of a node to its neighboring nodes. For example, the numbers of parents and/or children nodes of a node are considered. It is equivalent in spirit to our (p, q)-node pattern. Bergmann *et al.* [22] propose an approach for incremental graph-based pattern matching via a model transformation language. Our idea of using p-paths is inspired from the use of q-grams in Information Retrieval [13]. A q-gram refers to a sequence of continuous characters or words in text processing. q-grams are widely used to represent strings and effective in approximate string matching [13].

7 Conclusions

Structure-oriented approaches in clone detection have become popular. However, existing methods for capturing structural information in structure-oriented representation of software artifacts are either too computationally expensive to be efficient or too light-weight to be accurate in clone detection.

In this paper, we introduce Exas, a light-weight structural feature extraction approach that can approximate well the structure of tree-based and graph-based software fragments. In Exas, the characteristic features are extracted from the patterns of elements of the trees and graphs. The fragments are characterized by their counting vectors of those features. We also provided efficient strategies for fragment generation and algorithms for computing the vectors. We implemented two tools to show the applications and the usefulness of our approach.

Our analytical and empirical studies show that our structural characteristic features are accurate. The detection result is highly precise while it does not lose completeness. Importantly, our approach can be used as an approximation solution for other problems that involve graph isomorphism or tree/graph similarity.

Acknowledgment. This project was funded in part by a grant from the Vietnam Education Foundation (VEF) for the second author. The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of VEF. The fifth author is partially supported by the NSF award #0737029 and the Litton Industries Professorship.

References

1. Kontogiannis, K.A., Demori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. Reverse Engineering, 77–108 (1996)
2. Roy, C., Cordy, J.: Towards a mutation-based automatic framework for evaluating code clone detection tools. In: C3S2E 2008, pp. 137–140. ACM, New York (2008)
3. Read, R., Corneil, D.: The graph isomorph disease. Journal of Graph Theory 1, 339–363 (1977)
4. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM 1998, p. 368. IEEE CS, Los Alamitos (1998)
5. Li, Z., Lu, S., Myagmar, S.: CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. 32(3), 176–192 (2006)
6. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28(7), 654–670 (2002)
7. Evans, W.S., Fraser, C.W., Ma, F.: Clone detection via structural abstraction. In: WCRE 2007: Working Conference on Reverse Engineering, pp. 150–159. IEEE CS, Los Alamitos (2007)
8. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: scalable and accurate tree-based detection of code clones. In: ICSE 2007, pp. 96–105. IEEE CS, Los Alamitos (2007)
9. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Cleman: Comprehensive clone group evolution management. In: ASE 2008. IEEE CS, Los Alamitos (2008)
10. The MathWorks Inc. SIMULINK Model-Based and System-Based Design (2002)
11. Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.F., Teuchert, S.: Clone detection in automotive model-based development. In: ICSE 2008, pp. 603–612. ACM, New York (2008)
12. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and Accurate Clone Detection in Graph-based Models. In: ICSE 2009, International Conference on Software Engineering. IEEE CS, Los Alamitos (2009)
13. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Albert-Ludwigs University at Freiburg. Technical report (1991)

14. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: SIGMOD 2005: International conference on Management of data (2005)
15. Kuramochi, M., Karypis, G.: Finding frequent patterns in a large sparse graph*. Data Mining and Knowledge Discovery 11(3), 243–271 (2005)
16. Liu, H., Ma, Z., Zhang, L., Shao, W.: Detecting duplications in sequence diagrams based on suffix trees. In: APSEC 2006, pp. 269–276. IEEE CS, Los Alamitos (2006)
17. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. SIGSOFT Softw. Eng. Notes 28(5), 227–236 (2003)
18. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE 2005, pp. 54–65. ACM, New York (2005)
19. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: ASE 2005, pp. 204–213. ACM, New York (2005)
20. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: ICSE 2007, pp. 54–64. IEEE CS, Los Alamitos (2007)
21. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE 2007, pp. 164–173. ACM, New York (2007)
22. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: GRaMoT 2008: Proc. of international workshop on graph and model transformations, pp. 25–32. ACM, New York (2008)
23. Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. 33(11), 725–743 (2007)
24. Basit, H., Jarzabek, S.: Efficient token based clone detection with flexible tokenization. In: FSE 2007, pp. 513–516. ACM, New York (2007)
25. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. 33(9), 577–591 (2007)
26. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: WCRE 2006, pp. 253–262. IEEE CS, Los Alamitos (2006)
27. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 26(1), 28–42 (1996)
28. Johnson, J.H.: Identifying redundancy in source code using fingerprints. In: CASCON 1993, pp. 171–183. IBM Press (1993)
29. Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM J. Comput. 26(5), 1343–1362 (1997)
30. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS®: Program transformations for practical scalable software evolution. In: ICSE 2004, pp. 625–634. IEEE CS, Los Alamitos (2004)
31. Wahler, V., Seipel, D., Gudenberg, J.W., Fischer, G.: Clone detection in source code by frequent itemset techniques. In: SCAM 2004, pp. 128–135. IEEE CS, Los Alamitos (2004)
32. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
33. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: ICSM 1996, p. 244. IEEE CS, Los Alamitos (1996)