Dynamic Shape Analysis of Program Heap using Graph Spectra (NIER Track)

Muhammad Zubair Malik
University of Texas at Austin
zubair@mail.utexas.edu

ABSTRACT

Programs written in languages such as Java and C# maintain most of their state on the heap. The size and complexity of these programs pose a challenge in understanding and maintaining them; Heap analysis by summarizing the state of heap graphs assists programmers in these tasks. In this paper we present a novel dynamic heap analysis technique that uses spectra of the heap graphs to summarize them. These summaries capture the shape of recursive data structures as dynamic invariants or likely properties of these structures that must be preserved after any destructive update. Initial experiments show that this approach can generate meaningful summaries for a range of subject structures.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Class Invariants*

General Terms

Reliability, Verification

Keywords

Structural Invariant Generation, Shape Analysis, Deryaft

1. INTRODUCTION

Dynamically allocated data makes the space of software artifacts practically infinite which makes them difficult to characterize; also the ability of linked structures to change their shape makes them powerful programming constructs but at the same time extremely hard to analyze. Most of the existing dynamic and static shape analyses[8, 2, 5, 10, 1, 6] that check non-trivial properties of such programs impose a substantial burden on the users, e.g., by requiring the users to provide invariants, such as loop or representation invariants, or to provide complete executable implementations as well as specifications. Daikon [4] presented an alternative by detecting program invariants from traces of program executions. Since then dynamic analysis for generating likely program invariants has become an increasingly popular technique in software checking methodologies. Daikon however only detects primitive invariants. The other more interesting invariants that pervade object-oriented programs are data structure invariants, which define desired heap configurations of dynamically-allocated data, are detected by a recent tool Deryaft [7].

Given a small set of structure representations in heap as examples, Deryaft analyzes them to formulate local and global properties that the structures exhibit. Deryaft focuses on graph properties for effective formulation of structural invariants, including reachability, and views the program heap as an edge-labeled graph. Given a set of concrete structures Deryaft inspects them to formulate a set of hypotheses on the underlying structural as well as data constraints that are likely to hold. Next, it checks which hypotheses actually hold for the structures. Finally, it translates the valid hypotheses into a Java predicate that represents the structural invariants of the given structures. The resulting predicate takes an input structure, traverses it, and returns true if and only if the input satisfies the invariants. Deryaft uses instrumentation, reflection and generic programming support from language to perform its shape analysis. We find the Deryaft's approach of finding structural invariants quite fascinating, especially Deryaft's view of structures as edge-labeled graphs. It is known that any graph can be represented using a matrix whose shape invariants are its eigenvectors. Our work shows that eigenvectors of a matrix of a program heap are related to its invariant properties. This work is similar to Deryaft but uses a different data transformation to generate likely program invariants. Other areas in science such as signal processing have benefited from frequency domain processing of data even when similar results were possible using direct (time-domain) representation. This paper makes the following major contributions:

- **Graph Spectral Representation:** We have modified Deryaft invariant generation algorithm and tool to support a more efficient data representation which is naturally suited for the task of invariant detection.

- **Learning New Data Constraints:** Only a limited number (a few thousand) of spectral rules over data invariants are known but we provide a non-symbolic way of extending the rule book by automating the learning of novel constraints.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

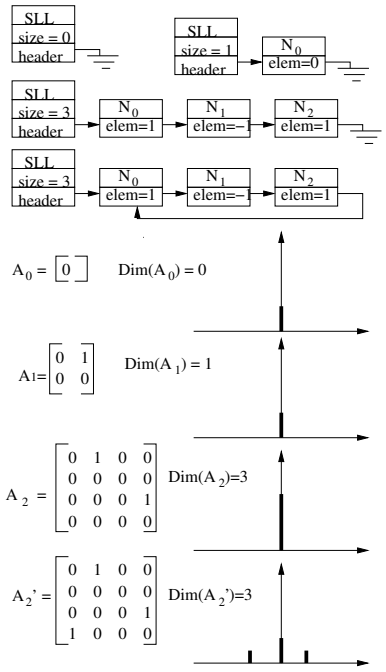

Figure 1: Three valid Linked Lists have similar spectral representation but for the cyclic list the spectra changes

- **Fast and Stable Eigenvalue Computation:** We present fast and stable alternative to prevalent QR decomposition used for eigenvalue computation, especially when eigenvalues for sub-matrix of current matrix are known.

2. ILLUSTRATIVE EXAMPLE

Given the following code of a singly linked list.

```
public class SinglyLinkedList{
// first list node
    private Node header;
// number of nodes in the list
    private int size;
    private static class Node{
        int elem;
        Node next;
    }
};
```

Figure 1 shows four concrete states of this list on the heap. Below the lists are directed adjacency matrix representations, matrix dimension and frequency count of eigenvalues of the matrices. It is to be noted that cyclic list has distinctly different spectra from valid lists. This result is no accident but follows directly from Spectral graph theory [3]: the directed adjacency spectra of any acyclic directed graph is a delta function whose magnitude depends on the size of the structure.

3. ALGORITHM

We take a relational view [6] of the program heap: we view the heap of a Java program as an edge-labeled directed graph whose nodes represent objects and whose edges represent fields. For languages, such as C and C++, that allow pointer arithmetic and arbitrary conversions between integer values and memory addresses, we would need a different view. For type-safe subsets of such languages, we can still use the relational view. The presence of an edge labeled f from node o to v says that the f field of the object o points to the object v (or is *null*) or has the primitive value v. Mathematically, we treat this graph as a set (the set of nodes) and a collection of relations, one for each field. We partition the set of nodes according to the declared classes and partition the set of edges according to the declared fields; we represent *null* as a special node. A particular program state is represented by an assignment of values to these sets and relations. Since we model the heap at the concrete level, there is a straightforward isomorphism between program states and assignments of values to the underlying sets and relations. The basic model of heap for our `SinglyLinkedList` example consists of three sets, each corresponding to a declared class or primitive type `SinglyLinkedList, Node, Int` and four relations corresponding to a declared field:

```
header:SinglyLinkedList x Node
size: SinglyLinkedList x int
next: Node x Node
```

We assume (without the loss of generality) that each structure in the given set has a unique pointer. Thus, the abstract view of a structure is a rooted edge-labeled directed graph, including properties that involve reachability, e.g., acyclicity. We partition the set of reference fields declared in the classes of objects in the given structures into two sets: core and derived. For a given set, **S**, of structures, let **F** be the set of all reference fields.

Definition 1. *A subset $C \subset F$ is a core set with respect to S if for all structures $s \in S$ the set of nodes reachable from the root of $\boldsymbol{r}$ of $\boldsymbol{s}$ along the fields in $\boldsymbol{C}$ is the same as the set of nodes reachable from $\boldsymbol{r}$ along the fields in $\boldsymbol{F}$.*

In other words, core set preserves reachability in terms of the set of nodes. Indeed, the set of all fields is itself a core set. We aim to identify a minimal core set, i.e., a core set with the least number of fields.
To illustrate, the set containing both the reference fields header and next in the example from Section 2 is a minimal core set with respect to the given set of lists.

Definition 2. *For a core set C the set of fields $F - C$ is a derived set.*

Our partitioning of the reference fields is inspired by the notion of back-bone in certain data structures.

A matrix is an efficient way of representing a graph for analysis. There is a correspondence between many graph-theoretic properties and matrix properties that makes the problem easier to visualize and solve [11]. For very simple graphs such as lists and trees we need various matrix representations because spectral disambiguation is hard using a one such representation. We have to consider these various representation because they all encode graphs differently, resulting in different spectral representation and allow us to discover different constraints over data. In the following discussion, we denote a graph by $G = (V, E)$ where V is the set

of nodes and $E \subset V \times V$ is the set of edges. the degree of a vertex u is the number of edges leaving the vertex u and is denoted by d_u.

The most basic representation of a graph is using the adjacency matrix B for the graph. The matrix is given by

$$B(u,v) = \begin{cases} 1 & (u,v) \in E \\ 0 & otherwise \end{cases}$$

Since the heap graph is directed, the adjacency matrix will not be symmetric.Undirected Adjacency matrix A from the directed adjacency matrix B is obtained using

$$A(u,v) = \begin{cases} 1 & B(u,v) = 1 or B(v,u) = 1 \\ 0 & otherwise \end{cases}$$

Clearly if the graph is undirected, the matrix A is symmetric. As a consequence, the eigenvalues of A are real. These eigenvalues may be positive, negative or zero and the sum of eigenvalues is zero. The eigenvalues maybe ordered by their magnitude and collected into a vector which describes the graph spectrum.

To find certain rules it is useful to have a positive semidefinite matrix representation of the graph. This may be achieved by using the Laplacian. We first construct the diagonal degree matrix D, whose diagonal elements are given by the node degrees $D(u,u) = du$. From the degree matrix and the adjacency matrix we then can construct the standard Laplacian matrix

$$L = D - A \tag{1}$$

i.e, the degree minus the undirected adjacency matrix. The Laplacian has at least one eigenvalue equal to zero, and the number of such eigenvalues is equal to the number of disjoint parts in the graph. The sign-less Laplacian has all the entries greater than zero and is given by

$$|L| = D - A \tag{2}$$

The normalized Laplacian matrix is defined to be the matrix given by

$$\hat{L} = \begin{cases} 1 & if u = v \\ -\frac{1}{\sqrt{d_u d_v}} & \text{if u and v are adjacent} \\ 0 & otherwise \end{cases}$$

We can also write it as $\hat{L} = D^{1/2} L D^{-1/2}$. As with the Laplacian of the graph, this matrix is positive semidefinite and so has positive or zero eigenvalues. the normalization factor means that the largest eigenvalue is less than or equal to 2, with quality only when G is bipartite. Again, the matrix has at least one zero eigenvalue. Hence all the eigenvalues are in the range $0 \leq \lambda \leq 2$.

The spectrum of the graph is obtained from eigen decomposition of one of the representations given above. Let X be the matrix representation in question. Then the eigen decomposition is

$$X = \Phi \Lambda \Phi^t \tag{3}$$

where

$$\Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_{|v|}) \tag{4}$$

is the diagonal matrix with the ordered eigenvalues as elements and

$$\Phi = diag(\phi_1, \phi_2, ..., \phi_{|v|}) \tag{5}$$

is the matrix with the ordered eigenvectors as columns. The spectrum is the set of eigenvalues $\{\lambda_1, \lambda_2, ..., \lambda_{|v|}\}$. The spectrum is particularly useful as a graph representation because it is invariant under the similarity transform PLP^T, where P is a permutation matrix. In other words, two isomorphic graphs will have the same spectrum. It is to be noted that the converse is not true, two non isomorphic graphs may share the same spectrum.

The spectrum of a graph has been widely used in graph theory to characterize the properties of graph and extract information from its structure [3]. The spectrum may change dramatically with a small change in structure, but for mining structural invariants our key observation is that if structural invariants are preserved, the normalized spectra remains similar in shape.

Given a graph representation of heap and a dictionary of spectral rules the overall algorithm performs these steps:(1) Identify core and derived fields (2) Apply spectral rule based engine to formulate global and local properties that are relevant (3) Compute of properties that actually hold (4) Generation of Java code that represents these properties

4. LEARNING SPECTRAL CONSTRAINTS

Are all graphs determined by their spectra? Is a very hard question and undecided at the moment. Zhu and Wilson [11] have shown that using a combination of representations mischaracterizing becomes highly unlikely. Nevertheless our rule based system is limited and spectral rules for some very interesting properties of complex data do not exist in literature. Can we learn such rules with reasonable accuracy?

We were able to learn the balance property of trees. Given a set of positive and negative examples, randomly generated from the class of structures exhibiting the desired property, so that they represent the actual distribution of the structural variance, we transform them into various matrix representations; we define kernels over the spectral decomposition of these matrixes which allow us to distinguish similar class of structures from a different one. We build a similarity matrix and train a support vector machine. This process is repeated for all possible combinations of matrix representations and valid kernels available for those representations. The best performing support vector machine over the test data is selected to form a rule over the property.

The accuracy of a rule learnt using machine learning techniques is better than a chance classifier and has only been possible because we used a novel heap representation for shape analysis.

5. FAST EIGENVALUE COMPUTATION

For our approach to be useful we need to compute eigenvalues efficiently. Various algorithms using shape analysis need to verify structures repeatedly with only one edge or one corresponding matrix entry changed. We use Cuppen's Divide and Conquer algorithm for calculating eigenvalues [9]. This hierarchical design enables very fast computation of eigenvalues from previously computed eigenvalues.

6. EVALUATION

Table1 describes some of the spectral rules used by our approach and table 2 demonstrate the summary of properties generated for various structures by applying our approach.

Table 1: Some Examples of Spectral Rules

Property	Matrix representation	Rule	Decision Procedure	Probabilistic
Acyclicity	Directed adjacency	All eigen values are zero	X	
Girth	Adjacency	Path length for which trance is non-zero	X	
Cyclicity	Directed adjacency	Girth = size	X	
Tree	Laplacian	All eigenvalues are between -2 and 2	X	
Balance	Heat Kernel	Trained SVM		X

Table 2: Structural properties discovered using our approach. The first column lists the subjects. The second column lists the properties discovered using the rule-based approach. The third column lists the properties learned using the support vector machine.

structure	Invariants	Learned
Singly Linked List	acyclicity, sizeOK	N/A
Circular Linked Lsit	circularity, sizeOK	N/A
doubly Linked Lsit	acyclicity sizeOK	N/A
Binary Tree	acyclicity, sizeOK	N/A
AVL Tree	acyclicity	balance

The singly linked list is the simplest of the structures. It only has one core field, and the only structural constraint is acyclicity. As presented in Section 2, the acyclicy property is determined by observing the spectra of the directed adjacency matrix of the linked list. For the circular list, the circularity property is detected by matching the girth of the graph with the size of the list. The girth of the graph is the length of the longest cycle in the graph. If the girth is equal to the number of nodes reachable using the next field, then the list is circular. For a doubly linked list, the transpose property is discovered by having a symmetrical matrix on the directed adjacency matrix which implies that the sum of the eigen values is zero. The acyclicity along the next field is determined in a similar way as that of the singly linked list. For the binary tree, the tree property is determined using the spectra when using the Laplacian matrix as a representation of the graph. The range of the magnitudes of the spectra should be above -2 or below 2. This indicates a tree property. Also similar to the linked list, the shape of the spectra when using the adjacency matrix representation determines the acyclicity property. For the AVL tree, the balance property is learned by the SVM when using a heat kernel to measure the similarity between the given example structures.

7. CONCLUSION

We have implemented a tool for generating likely invariants for complex data structures. It accepts a few concrete examples of the structures and outputs a Java predicate function containing structural invariants of the example structures. We view structures as edge labeled graphs and focus on core fields and the structure's backbone to analyze it. Our approach uses an efficiently designed inference engine based on graph spectra representation of structures to hypothesize its characteristics. Unlike conventional rule-based system that entirely depends on expert's knowledge our approach actively tries to learn new rules. And last but not the least we utilize stable logarithmic time eigenvalue evaluation methods. Experiments on some of the most commonly used complex data structures implemented as libraries and stand alone applications show that our approach is accurate and efficient.

8. ACKNOWLEDGEMENTS

The author wishes to thank his advisor Sarfraz Khurshid for suggesting the idea of using graph spectra for structural invariant generation and supervising this work. This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

9. REFERENCES

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, 2005.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, 2002.

[3] D. M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs: Theory and Applications*. John Wiley & Sons Inc, 1998.

[4] M. D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, 2000.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[6] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[7] M. Z. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid. Deryaft: a tool for generating representation invariants of structurally complex data. In *ICSE*, 2008.

[8] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, 2001.

[9] J. D. Rutter. A serial implementation of cuppen's divide and conquer algorithm. Technical report, Berkeley, CA, USA, 1991.

[10] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1), 1998.

[11] P. Zhu and R. C. Wilson. A study of graph spectra for comparing graphs. In *BMVC*, 2005.