A Hybrid Approach for Selecting and Optimizing Graph Traversal Strategy for Analyzing BigCode

by

Ramanathan Ramu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

Major: Computer Science

Program of Study Committee:

Dr. Hridesh Rajan, Major Professor

Dr. Andrew Miner

Dr. Wei Le

Iowa State University

Ames, Iowa

2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Dr. Hridesh Rajan, Dr.Hoan and Ganesha Upadhaya for their guidance, patience and support throughout this research and the writing of this thesis. Thanks are due to the US National Science Foundation for financially supporting this project. I would like to thank my committee members Dr. Wei Le and Dr. Andrew Miner for their efforts and contributions to this work. Also, I would like to thank the reviewers of ECOOP 2017 conference for their insightful feedback. I would like to extend my thanks to all the members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research.

I am very grateful to my parents Ramu and Meenal and my friends for their moral support and encouragement throughout the duration of my studies.

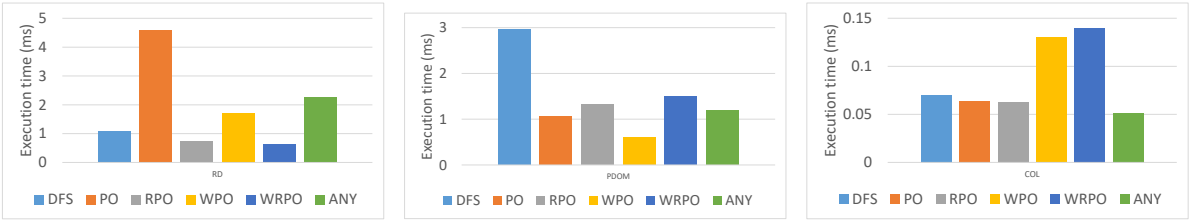
ABSTRACT

Our newfound ability to analyze source code in massive software repositories such as GitHub has led to an uptick in data-driven solutions to software engineering problems. Source code analysis is often realized as traversals over source code artifacts represented as graphs. Since the number of artifacts that are analyzed is huge, in millions, the efficiency of the source code analysis technique is very important. The performance of source code analysis techniques heavily depends on the order of nodes visited during the traversals: the traversal strategy. For instance, selecting the best traversal strategy and optimizing it for a software engineering task, that infers the temporal specification between pairs of API method calls, could reduce the running time on a large codebase from 64% to 96%. While, there exists several choices for traversal strategy, like depth-first, post-order, reverse post-order, etc., there exists no technique to choose the most time-efficient strategy for traversals. In this paper, we show that a single traversal strategy does not fit all source code analysis scenarios. Somewhat more surprisingly, we demonstrate that given the source code expressing the analysis task (in a declarative form) one can compute static characteristics of the task, which together with the runtime characteristics of the input, can help predict the most time-efficient traversal strategy for that (analysis task, input) pair. We also demonstrate that these strategies can be realized in a manner that is effective in accelerating ultra-large-scale source code analysis. Our evaluation shows that our technique successfully selected the most time-efficient traversal strategy for 99.99%-100% of the time and using the selected traversal strategy and optimizing it, the running times of a representative collection of source code analysis in our evaluation were considerably reduced by 1%-28% (13 minutes to 72 minutes in absolute time) when compared against the best performing traversal strategy. The overhead imposed by collecting additional information for our approach is less than 0.2% of the total running time for a large dataset that contains 287K Control Flow Graphs (CFGs) and less than 0.01% for an ultra-large dataset that contains 162M CFGs.

CHAPTER 1. Introduction

The availability of open source repositories like GitHub is driving data-driven solutions to software engineering problems, e.g. specification inference Nguyen et al. (2014), discovering programming patterns Thummalapenta and Xie (2009), suggesting bug fixes Livshits and Zimmermann (2005); Li et al. (2006), etc. These software engineering tasks analyze different source code representations, such as text, abstract syntax trees (ASTs), control flow graphs (CFGs), at massive scale. The performance of source code analysis over graphs heavily depends on the order of the nodes visited during the traversals: *the traversal strategy*. While graph traversal is a well-studied problem, and various traversal strategies exists; e.g., depth-first, post-order, reverse post-order, etc, no single strategy works best for different kinds of analyses and different kinds of graphs. Our contribution is *hybrid traversal selection*, a novel source code analysis optimization technique for source code analyses expressed as graph traversals.

Motivation and Key Observations. To motivate, consider a software engineering task that infers the temporal specifications between pairs of API method calls, i.e., a call to a must be followed by a call to b Engler et al. (2001); Ramanathan et al. (2007); Weimer and Necula (2005); Yang et al. (2006). A data-driven approach for inference is to look for pairs of API calls that frequently go in pairs in the same order at API call sites in the client methods' code. Such an approach contains (at least) three source code analyses on the control flow graph (CFG) of each client method: 1) identifying references of the API classes and call sites of the API methods which can be done using *reaching definition* analysis Nielson et al. (2010); 2) identifying the pairs of API calls (a, b) where b follows a in the client code which can be done using *post-dominator* analysis Aho et al. (2006); and 3) collecting pairs of temporal API calls by traversing all nodes in the CFG–let us call this *collector* analysis. These analyses need to be run on a large number of client methods to produce temporal specifications with high confidence.



(a) Reaching definition analysis. (b) Post-dominator analysis. (c) Collector analysis.

Figure 1.1: Running times (ms) of the three analyses on graph A using different traversal strategies.

Implementing each of these analyses involves traversing the CFG of each client method. The traversal strategy could be chosen from a set of standard strategies e.g depth-first search (DFS), post-order (PO), reverse post-order (RPO), worklist with post-ordering (WPO), worklist with reverse post-ordering (WRPO) and any order (ANY).

Figure 1.1 shows the performance of each of these three analyses when using standard traversal strategies. These runs are analyzing the CFG of a method in the DaCapo benchmark Blackburn et al. (2006). Actual implementation of this method is not important, but it suffices to know that the CFG, which we shall refer to as Graph A, has 50 nodes and has branches but no loops. Figure 1.1 shows that, for graph A, the WRPO performs better than other strategies for the reaching definition analysis while the WPO outperforms the others for the post-dominator analysis and the ANY traversal works best for the collector analysis.

No Traversal Strategy Fits All. The performance results are somewhat expected, but require understanding the subtleties of the analyses. Reaching definition analysis is a forward data-flow analysis where the output at each node in the graph is dependent on the outputs of their predecessor nodes. So, DFS, RPO and WRPO by nature are the most suitable. However, worklist is the most efficient strategy here because it visits only the nodes that are yet to reach fixpoint unlike other strategies that also visit notes that have already reached fixpoint. Post-dominator analysis, on the other hand, is a backward analysis meaning that the output at each node in the graph is dependent on the outputs of their successor nodes. Therefore, the worklist with post-ordering is the most efficient traversal. For the collector analysis, any order traversal works better than other traversal strategies for graph A. This is because for this

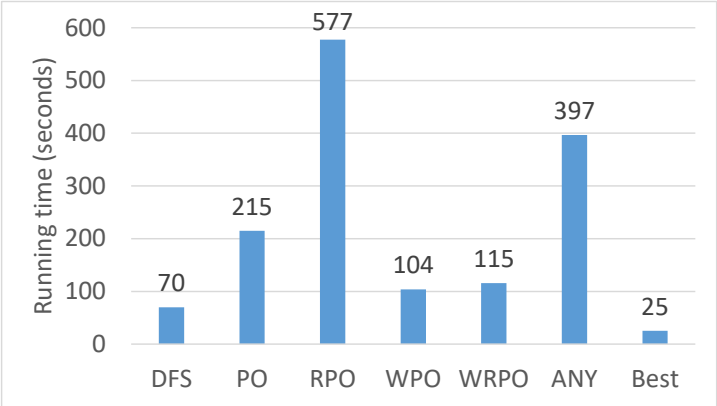


Figure 1.2: Running times of three analyses using different traversal strategies on a large codebase.

analysis the output at each node is not dependent on the output of any other nodes and hence it is independent of the order of nodes visited. The any order traversal strategy does not have the overhead of visiting the nodes in any particular order like DFS, PO, RPO nor the overhead to maintain the worklist. Therefore any order traversal performs better than other traversal strategies.

Properties of Input Graph Determine Strategy. For the illustrative example discussed above, DFS and RPO were worse than WRPO for the reaching definition analysis and PO was worse than WPO for post-dominator because they require one extra iteration of analysis to be performed and realize that fixpoint has been reached. However, since graph A does not have any loops, if the graph A's nodes are visited in such a way that each node is visited after its predecessors for reaching definition analysis and after its successors for post-dominator analysis, then the additional iteration is actually redundant. Given that graph A has no loops, one could optimize RPS or PO to bypass the extra iteration and fixpoint checking. Thus, the optimized RPS or PO would run the same number of iterations as the respective worklist-based ones and finish faster than them because the overhead of maintaining the list is eliminated.

The potential gains of selecting a suitable traversal strategy can be significant. To illustrate, consider Figure 1.2 that shows the performance of our entire illustrative example (inferring temporal specifications) on a large corpus of 287,000 CFGs extracted from the DaCapo benchmark dataset Blackburn et al. (2006). Figure 1.2 shows the bar chart for the total running

times of the three analyses. The **Best** strategy is an ideal one where we can always choose the most efficient with all necessary optimizations. The bar chart confirms that fixing a traversal strategies for running different analyses on different types of graphs is not efficient. Selecting and optimizing traversal strategy for each analysis on each graph is desirable. Such a strategy could reduce the running time on a large dataset from 64% (against DFS) to 96% (against RPO).

Our approach relies on our observations that a suitable traversal strategy is dependent on both the source code analyses and input graphs' properties. The former are static properties and the latter are runtime properties. More importantly, depending on the properties of analyses and graphs, existing traversals could be optimized to improve their performance. We have evaluated our technique using a set of 21 source code analysis that includes control and data-flow analysis, and analysis to find bugs. The evaluation is performed on two datasets: a dataset containing well-maintained projects from DaCapo benchmark (contains a total of 287K graphs), and a ultra-large dataset containing more than 380K projects from GitHub (contains a total of 162M graphs). Our evaluation shows that our technique successfully selected the most time-efficient traversal strategy for 99.99%–100% of the time and using the selected traversal strategy and optimizing it, the running times of a representative collection of source code analysis in our evaluation were considerably reduced by 1%-28% (13 minutes to 72 minutes in absolute time) when compared against the best performing traversal strategy. The overhead imposed by our approach is negligible (less than 0.2% of the total running time for a large dataset and less than 0.01% for an ultra-large dataset).

In summary, this paper makes the following contributions:

- It describes a system for expressing source code analysis as traversals. The constructs and operations in the system allows different source code analysis to be expressed in a manner that allows automatic selection of the best traversal strategies.

- It defines a set of novel properties about the traversal expressed in our system. It also describes algorithms for analysing traversals for inferring these properties.

- It describes a novel decision tree for selecting the most suitable traversal strategy. The

static and runtime properties also allows certain optimizations to be performed on the selected traversal strategy to further improve the performance.

- It demonstrates the potentially broad range of applications of hybrid traversal selection for optimizing source code analysis such as available expressions, local may alias, live variable, nullness analysis, post dominator, reaching definitions, resource status, very busy expression, etc.

CHAPTER 2. Contributions

In this work, we develop *hybrid traversal selection*, a novel program analysis optimization technique for BigCode analyses expressed as graph traversals. Our approach relies on our observations that a suitable traversal strategy is dependent on both the program analyses and input graphs' properties. The former are static properties and the latter are dynamic properties. More importantly, depending on the properties of analyses and graphs, existing traversals could be optimized to improve their performance. Hybrid traversal selection relies on several technical underpinnings:

[**Traversal Declaration and Traverse Expression**] Programmers can declare their program analyses as one or more **traversal** declarations and run them using **traverse** expression. The runtime implementation of the **traverse** expression selects a suitable traversal strategy based on the **traversal** declaration and the input graph. Main benefit of these linguistic abstractions is that they abstract away traversal related code so that the traversal strategy can be replaced as needed by the analysis runtime.

[**Data-Flow and Loop Sensitivity Analyses for Traversals**] We show that traversal strategy selection depends on three critical properties of the traversal: *data-flow sensitivity*, *loop sensitivity*, and *traversal direction*. We propose algorithms for computing these properties. Our analysis system implements these algorithms. These properties are computed statically and their values are stored as metadata to be utilized by the traversal selection at runtime.

[**Graph Cyclicity**] We have observed that the traversal strategy selection depends on one dynamic property of the input which is *graph cyclicity*. This property partitions the set of graphs into three categories: those that are sequential, those with branches but no cycles, and those with cycles. Our system computes this property at graph construction time and stores it as an attribute in the runtime graph representation.

[**Decision Tree for Traversal Strategy Selection**] We have devised a *decision tree* for traversal strategy selection that given data-flow sensitivity, loop sensitivity, and traversal direction properties of the analysis and the cyclicity property of the input graph produces a selection for traversal strategy. While the tree is utilized by our automated system, it could also be used by a programmer for manual traversal selection.

Hybrid traversal selection has two direct benefits. First, it improves the efficiency of BigCode analysis thus speeding up data-driven science in this important area. Second, it frees up programmers from having to write traversal related code and then optimizing it based on the analysis and the graph at hand.

We have evaluated our technique using a set of 21 source code analysis that includes control and data-flow analysis, and analysis to find bugs. The evaluation is performed on two datasets: a dataset containing well-maintained projects from DaCapo benchmark (contains a total of 287K graphs), and a ultra-large dataset containing more than 380K projects from GitHub (contains a total of 162M graphs). Our evaluation shows that our technique successfully selected the most time-efficient traversal strategy for 99.99%–100% of the time and using the selected traversal strategy and optimizing it, the running times of a representative collection of source code analysis in our evaluation were considerably reduced by 1%-28% (13 minutes to 72 minutes in absolute time) when compared against the best performing traversal strategy. The case studies show that hybrid traversal reduces 80–175 minutes in running times for three software engineering tasks. The overhead imposed by our approach is negligible (less than 0.2% of the total running time for a large dataset and less than 0.01% for an ultra-large dataset).

In summary, this paper makes the following contributions:

- It describes a system for expressing source code analysis as traversals. The constructs and operations in the system allows different source code analysis to be expressed in a manner that allows automatic selection of the best traversal strategies.

- It defines a set of novel properties about the traversal expressed in our system. It also describes algorithms for analysing traversals for inferring these properties.

- It describes a novel decision tree for selecting the most suitable traversal strategy. The

static and runtime properties also allows certain optimizations to be performed on the selected traversal strategy to further improve the performance.

- It demonstrates the potentially broad range of applications of hybrid traversal selection for optimizing source code analysis such as available expressions, local may alias, live variable, nullness analysis, post dominator, reaching definitions, resource status, very busy expression, etc.

CHAPTER 3. Hybrid Traversal Selection for Efficient Source Code Analysis

In this section we first provide a brief overview of our technique, followed by an overview of the constructs used for expressing source code analyses. We then describe properties, analyses, and a decision tree that are the technical underpinnings of our selection technique.

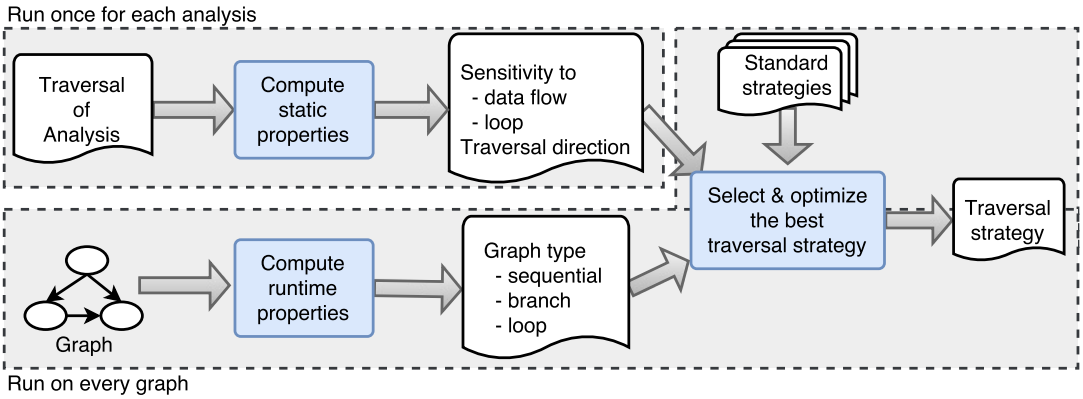


Figure 3.1: Overview of the hybrid approach for selecting and optimizing graph traversal strategy.

Figure 3.1 provides an overview of our approach and its key components. Inputs to our approach are source code analysis that contains one or more traversals (§3.1), and a graph. Output of our technique is an optimal traversal strategy for every traversal in the analysis. For selecting an optimal traversal strategy for a traversal, our technique computes a set of static properties of the analysis (§3.2), such as data-flow sensitivity, loop sensitivity, and extracts a runtime property about the graph that defines the cyclicity in the graph (sequential/branch/loop). Upon computing the static and runtime properties, our approach selects a traversal strategy from a set of candidate strategies (§3.3) for each traversal in the analysis (§3.4) and optimizes it (§3.5). The static properties of the traversals are computed only once for each analysis, whereas

graph cyclicity is determined for every input graph.

3.1 A System For Expressing Source Code Analysis As Traversals

A source code analysis is performed on various source code artifacts such as source code text, intermediate representations like abstract syntax trees (ASTs), graph-based representations like control flow graphs (CFGs) and call graphs (CGs), etc. In our system, source code analysis such as control- and data-flow analysis are expressed as traversals over CFGs.

Definition 1 *A control flow graph (CFG) is a directed graph* $G = (N, E, n_{start}, N_{end})$ *with a set of nodes* N *representing the program statements and a set of edges* $E \subseteq N \times N$ *representing the control flow relation between the program statements. A CFG has a single start node,* n_{start}*, and a set of end nodes,* N_{end}*.*

For any node $n \in N$, $n.preds$ is a set of immediate predecessors, $n.succs$ is a set of immediate successors, $n.stmt$ provides the program statement at the node, and $n.id$ is a unique identifier of the node. Here on, we use graph to refer to CFG.

A source code analysis over a graph visits nodes in the graph in certain order and collects information at nodes (aka, analysis facts or outputs). For instance, the *reaching definition* analysis over a CFG, visits every node in the CFG and collects the variable definitions at nodes as analysis facts. An analysis may require multiple traversals over a graph and each traversal may visit nodes multiple times (for fixpoint). For instance, the *reaching definition* analysis requires two traversal of the CFG: an *initialization* traversal for collecting the variable definitions at nodes as analysis facts, and a *propagation* traversal for propagating the analysis facts along the graph. The *initialization* traversal visits every node exactly once, whereas the *propagation* traversal may visit the nodes multiple times until a fixpoint is reached and the analysis facts at nodes does not change further.

In our system, a source code analysis over a graph is expressed by defining and invoking one or more traversals. A traversal is defined using a special `traversal` block:

```
t := traversal(n : Node) : T { tbody }
```

In this traversal block definition, `t` is the name of the traversal that takes a single parameter `n` representing the graph node that is being visited. A traversal may define a return type `T` representing the output type. The output type can be a primitive or a collection data type. A block of code that generates the traversal output at a graph node is given by `tbody`. The `tbody` may contain common statements and expressions, such as variable declarations, assignments, conditional statements, loop statements, and method calls, along with some special expressions discussed in this section.

A traversal can be invoked using a special `traverse` expression:

```
traverse(g, t, d, df, ls, fp)
```

A `traverse` expression takes six parameters: `g` is the graph to be traversed, `t` is the traversal to be invoked, `d` is the traversal direction and `df`, `ls`, `fp` are optional parameters. `df` is of boolean type which indicates whether the analysis is data flow sensitive or not. `ls` is also an boolean variable, indicating whether the analysis is loop sensitive or not. If `df` is not provided, Algorithm 1 in Section 2.2.2 will be used to compute this property. Similarly, if `ls` is not provided, Algorithm 2 in Section 2.2.4 will be used to compute this property. `fp` is a variable name of the user defined fixpoint function. A traversal direction is a value from the set {`FORWARD`, `BACKWARD`, `ITERATIVE`}, where `FORWARD` is used to represent a forward analysis (predecessors of a node are processed before the node), `BACKWARD` is used to represent a backward analysis (successors of a node are processed before the node), and `ITERATIVE` is used to represent a sequential analysis (visits nodes as they appear in the nodes collection). A user defined fixpoint function can be defined using the `fixp` block:

```
fp := fixp(...) : bool { fbody }
```

In this `fixp` block, `fixp` is a keyword for defining a fixpoint function. A fixpoint function can take any number of parameters, and it must always return a boolean. The body of the fixpoint function is defined in the `fbody` block. A fixpoint function can be assigned a name, which can be passed in the `traverse` expression.

Accessing Facts of Other Nodes. We also provide a special expression `output(n, t)` for querying the traversal output associated with a graph node `n`, in the traversal `t`.

Table 3.1: Operations on collections.

Operation	Description
`add(C, e)`	Adding an element e to collection C
`addAll(C1, C2)`	Adding all elements from collection C2 to collection C1
`remove(C, e)`	Removing an element e from collection C
`removeAll(C1, C2)`	Removing all elements from collection C1 that are also present in collection C2
`get(C, i)`	Element at index i from collection C is accessed
`has(C, e)`	Checking if collection C has element e
`equals(C1, C2)`	Checking if collection C1 and collection C2 has the same elements
`C1 = C2`	Assigning collection C2 to collection C1
`union(C1, C2)`	Returns the union of the elements in collection C1 and collection C2
`intersection(C1, C2)`	Returns the intersection of the elements in collection C1 and collection C2

Data Types and Collections. Our system for expressing source code analysis as traversals provides primitive and collection data types. Primitive types include: `bool, int, string` and collection types include: `Set and Seq`, where `Set` is a collection with distinct and unordered elements, whereas, `Seq` is a collection with distinct and ordered elements. A set of operations that can be performed on collection types is described in Table 3.1.

To summarize, we described a system for expressing source code analysis as traversals over graphs using two special constructs: `traversal` for defining a traversal, and `traverse` for invoking a defined traversal. A traversal may visit graph nodes multiple times (in case of fixpoint) and it can be invoked using several parameters specifying the direction of the traversal, a user defined fixpoint function, etc. A traversal output associated with graph nodes can be queried using a special expression `output()`. To be able to express a variety of source code analysis, our system provides primitive and collection datatypes with well-defined operations. Later in this section we demonstrate how the constructs and operations of the system enables determining properties of the source code analysis expressed in our system, such that optimal traversal strategies can be automatically selected.

An Example: Post dominator analysis. We now describe how to use our system to express source code analysis as traversals using an example source code analysis. Post dominator analysis is a backward control flow analysis that collects node ids of all nodes that post dominates every node in the CFG Aho et al. (2006). This analysis can be expressed using our system as shown in Listing 3.1.

Listing 3.1: Post dominator analysis: an example source code analysis expressed using our system.

```
1 allNodes: Set<int>;
2 initT := traversal(n: Node) {
3   add(allNodes, n.id);
4 }
5 domT := traversal(n: Node): Set<int> {
6   Set<int> dom;
7   if (output(n, domT) != null) {
8     dom = output(n, domT);
9   } else {
10    if (node.id == exitNodeId) {
11      dom = {};
12    } else {
13      dom = allNodes;
14    }
15  }
16  foreach (s : n.succs)
17    dom = intersection(dom, output(s, domT))
18  add(dom, n.id);
19  return dom;
20 }
21 fp := fixp(Set<int> curr, Set<int> prev): bool {
22  if(equals(curr, prev))
23    return true;
24  return false;
25 }
26 traverse(g, initT, ITERATIVE);
27 traverse(g, domT, BACKWARD, fp);
```

Listing 3.1 mainly defines two traversals `initT` (lines 2-4) and `domT` (lines 5-20), and invokes them using `traverse` expressions (lines 26 and 27). Line 21-25 defines a fixpoint function using `fixp` block, which is used in the `traverse` expression in line 27. Line 1 defines a variable `allNodes` of collection type `Set`, where `Set<int>` defines a collection type `Set` with elements of type `int`. Line

3 uses an operation `add` (defined in Table 3.1) on collection `allNodes`. The common statements and expressions used in the language to express the analysis are not described in our system, however all standard statements and expressions are allowed. For instance, `if-else` statements are used in lines 7-15, `foreach` iteration is used in lines 16-17, and so on. Lines 26 and 27 provides two flavors of invoking traversals using `traverse` expressions: one without a fixpoint and other with a user-defined fixpoint function. A usage of special expression `output(n, domT)` can be seen in line 8. The traversal `initT` does not define any output for CFG nodes, whereas, the traversal `domT` defines an output of type `Set<int>` for every node in the CFG. For managing the analysis output of nodes, `domT` traversal maintains an internal map that contains analysis output for every node, which can be queried using `output(n, domT)`. A pre-defined variable `g` that represents the CFG is used in the `traverse` expressions in lines 26 and 27.

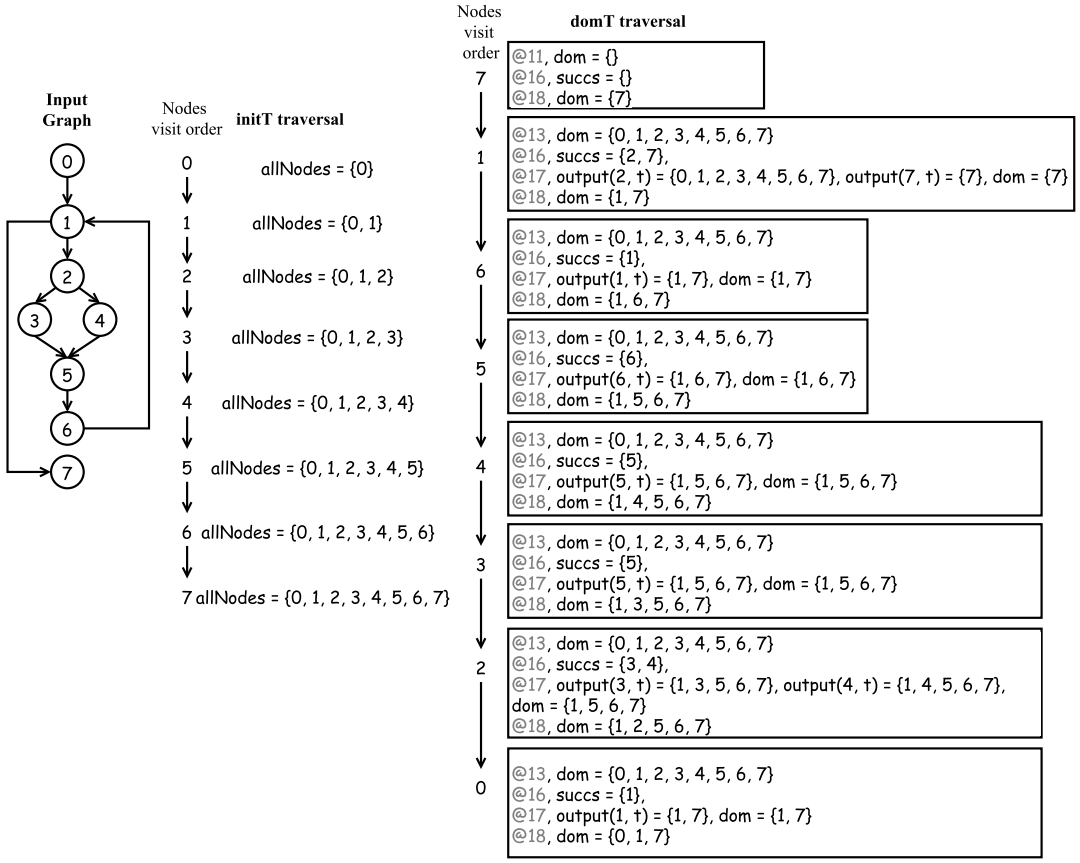


Figure 3.2: Running example of applying the post dominator analysis on an input graph containing branch and loop.

Figure 3.2 takes an example graph, and shows the results of `initT` and `domT` traversals. Our

example graph is a CFG containing seven nodes with a branch and a loop. The `initT` traversal visits nodes sequentially and adds node id to the collection `allNodes`. The `domT` traversal visits nodes in the post-order[1] and computes a set of nodes that post dominate every visited node (as indicated by the set of node ids). For instance, node 7 is post dominated by itself, hence the output at node 7 is {7}. In Figure 3.2, under `domT` traversal, for each node visited, we show the key intermediate steps indicated by @ line number. These line numbers correspond to the line numbers shown in Listing 3.1. We will explain the intermediate results while visiting node 2. In the `domT` traversal, at line 13, the output set `dom` is initialized using `allNodes`, hence `dom` = {0, 1, 2, 3, 4, 5, 6, 7}. At line 16, node 2 has two successors: {3, 4}. At line 17, the set `dom` is updated by performing an `intersection` operation using the outputs of successors 3 and 4. The output of 3 and 4 are {1, 3, 5, 6, 7} and {1, 4, 5, 6, 7} respectively. By performing the intersection of these two sets, the `dom` set becomes {1, 5, 6, 7}. At line 18, the id of the visited node is added to the `dom` set and it becomes {1, 2, 5, 6, 7}. Hence, the post dominator set for node 2 is {1, 2, 5, 6, 7}. Similarly, the post dominator set for other nodes can be calculated.

3.2 Static and Runtime Properties

While it is known in the literature that choosing a right traversal strategy for the source code analysis can significantly improve the performance Atkinson and Griswold (2001), how to choose a right traversal strategy, what factors influence the selection of the right traversal strategy, what properties of the analysis and graph are important, and how to determine them, were not known.

In this section, we describe the factors that influence the selection of the optimal traversal strategy for any given traversal. These factors include: the static properties of the analysis and the runtime properties of the graph. We also describe how the challenge of computing these properties is solved with the help of the constructs and operations proposed in our system of expressing source code analysis as traversals (§3.1).

[1] The traversal strategies chosen for `initT` and `domT` traversals is explained in §3.4.1.

3.2.1 Data-Flow Sensitivity

The *data-flow sensitivity* property of a traversal models the dependence of the traversal outputs of nodes in the input graph. A traversal is data-flow sensitive if the output of a node is computed using the outputs of other nodes. For instance, in the reaching definition data-flow analysis, the outputs of nodes are computed using the outputs of predecessors and in live variable data-flow analysis, the outputs of nodes are computed using the outputs of successors. Hence, both reaching definition and live variable analysis are data-flow sensitive.

Definition 2 *$P_{DataFlow}$ (**Data-flow sensitivity**). Given a traversal `t` with body `tbody`, a map O that collects and maintains the traversal output of nodes (O is indexed using node ids), and F, a function representing the computation of $O[.]$ in the traversal body `tbody`, if for any node n, its output $O[n]$ is computed by applying F over one or more of $O[n']$, where $n' \neq n$, then `t` is data-flow sensitive. That is $P_{DataFlow}$ is true otherwise false.*

For instance, consider the lines 16 and 17 of the `domT` traversal shown in Listing 3.1. Here, a variable `dom` holds the traversal output of a node n and it is computed by applying an `intersection` operation on the traversal outputs of successors of n. Here, the function F is `intersection` over the outputs of all successors of a node.

Algorithm 1: Algorithm to detect data-flow sensitivity

Input: `t := traversal(n : Node): T { tbody }`
Output: *true/false*

```
1 A ← getAliases(tbody, n);
2 foreach stmt ∈ tbody do
3     if stmt = output(n', t') then
4         if t' == t and n' ∉ A then
5             return true;
6 return false;
```

3.2.2 Computing Data-Flow Sensitivity

To determine the data-flow sensitivity property of a traversal, the operations performed in the traversal needs to be analyzed to check if the output of a node is computed using the

outputs of other nodes. In our system of expressing analysis as traversals, the only way to access the output of a node is via `output()` expression, hence given a traversal `t := traversal(n : Node) : T { tbody }` as input, Algorithm 1 parses the statements in the traversal body `tbody` to identify method calls of the form `output(n', t')` that fetches the output of a node `n'` in the traversal `t'`. If such method calls exists, they are further investigated to determine if `n'` does not point to `n` and `t'` points to `t`. If such method calls exists, it means that the traversal output for the current node `n` is computed using the traversal outputs of other nodes (`n'`) and hence the traversal is data-flow sensitive. For performing the points to check, Algorithm 1 assumes that an alias environment is computed by using must alias analysis Jagannathan et al. (1998). Algorithm 1 requires that the must alias analysis computes all names in the `tbody` that must alias each other at any program point. The must alias information ensures that Algorithm 1 never classifies a data-flow sensitive traversal as data-flow insensitive. A `tbody` may contain more than one `output()` statement, however Algorithm 1 requires only one `output()` statement that fetches the output of other nodes than the current node, to classify the traversal as data-flow sensitive. The control and loop statements in the `tbody` do not have any impact on Algorithm 1 for computing the data-flow sensitivity property.

3.2.3 Loop Sensitivity

The loop sensitivity property models the effect of the loops in the input graph. If an input graph contains loops and if the traversal is affected by the loop, the traversal may require multiple iterations to compute the output of nodes. In the multiple iterations, the traversal outputs of nodes either shrinks or expands to reach a fixpoint. Hence, we define a traversal as loop sensitive, if the traversal output of nodes in subsequent iterations shrinks or expands.

Definition 3 P_{Loop} *(**Loop sensitivity**). Given a traversal* `t`*, a map* O *that collects and maintains the traversal output of nodes (*O *is indexed using node ids), and* $O^i[n]$ *represents the output of node* n *in the* i^{th} *iteration, if* $O^{i+1}[n] \lll O^i[n]$ *or* $O^{i+1}[n] \ggg O^i[n]$*, then* t *is loop sensitive, i.e.* P_{Loop} *is true otherwise false. The relation* $\lll$ *represents* `shrink` *and it is given by,* $O^{i+1}[n] \lll O^i[n]$*, if* $|O^{i+1}[n]| < |O^i[n]|$*, and the relation* $\ggg$ *represents* `expand` *and it is given*

by, $O^{i+1}[n] \ggg O^{i}[n]$*, if* $|O^{i+1}[n]| > |O^{i}[n]|$*, where* $|C|$ *is the cardinality of the output collection* C*.*

Since the loop sensitivity property is defined only for data-flow sensitive traversals, we know that the traversal output of nodes in each iteration is computed using the traversal output of other nodes (possible neighbors), we have $O^{i}[n] = F(O^{i}[n'])$ and $O^{i+1}[n] = F(O^{i+1}[n'])$, where n, n' are any two nodes such that $n' \neq n$. By substituting these in the `shrink` relation $O^{i+1}[n] \lll O^{i}[n]$, we get, $F(O^{i+1}[n']) \lll F(O^{i}[n'])$. For this relation to be `true`, 1) the output of any node n' in any two subsequent iterations i and $i+1$ must shrink and 2) the function F has the shrink property. Similarly, for expand relation, 3) the output of any node n' in any two iterations i and $i+1$ must expand and 4) the function F has the expand property. As we know F represents the function in the traversal body that computes the outputs of nodes, if F has the property of shrink or expand, then the traversal can be classified as loop sensitive.

To give an example, consider the `domT` traversal shown in Listing 3.1. Since `domT` is data-flow sensitive, we can check the loop sensitivity property. There are two functions that contributes to the traversal output of any node n in `domT` traversal body. These are `intersection` (line 16) and `add` (line 17). For `domT` to be loop sensitive, we require that both `intersection` and `add` have either shrink or expand property. However, `intersection` has the shrink property and `add` has the expand property, hence we cannot classify `domT` to be loop sensitive.

3.2.4 Computing Loop Sensitivity

In general, computing the loop sensitivity property statically is challenging in the absence of an input graph, however the constructs and operations of our system enables static inference of this property.

A traversal is loop sensitive, if the output of any node in any two subsequent iterations either shrinks or expands. To determine if the traversal output expands or shrinks in the subsequent iterations, the operations performed in the traversal needs to be analyzed. Table 3.1 provides several operations that can be performed on the traversal outputs. The operations `add`, `addAll`, and `union` always expands the output and the operations `remove`, `removeAll`, and `intersection`

Algorithm 2: Algorithm to detect loop sensitivity

```
Input: t := traversal(n: Node): T { tbody }
Output: true/false
1  V ← {} // a set of output variables related to n;
2  V' ← {} // a set of output variables not related to n;
3  expand ← false;
4  shrink ← false;
5  gen ← false;
6  kill ← false;
7  A ← getAliases(n);
8  foreach stmt ∈ tbody do
9      if stmt is v = output(n', t') then
10         if t' == t then
11             if n' ∈ A then
12                 V ← V ∪ v;
13             else
14                 V' ← V' ∪ v;
15 foreach stmt ∈ tbody do
16     if stmt = union(c1, c2) then
17         if (c1 ∈ V and c2 ∈ V') || (c1 ∈ V' and c2 ∈ V) then
18             expand ← true;
19     if stmt = intersection(c1, c2) then
20         if (c1 ∈ V and c2 ∈ V') || (c1 ∈ V' and c2 ∈ V) then
21             shrink ← true;
22     if stmt = add(c1, e) || addAll(c1, c2) then
23         if c1 ∈ V then
24             gen ← true;
25     if stmt = remove(c1, e) || removeAll(c1, c2) then
26         if c1 ∈ V then
27             kill ← true;
28 if (expand and gen) || (shrink and kill) then
29     return true;
30 else
31     return false;
```

always shrinks the output.

Given a traversal `t` := `traversal(n : Node) : T { tbody }`, Algorithm 2 determines the loop sensitivity of `t`. Algorithm 2 investigates the statements in the `tbody` to determine if the traversal outputs of nodes in multiple iterations either expands or shrinks. For doing that, first it parses the statements to collect all output variables related and not related to input node `n` using the must alias information as in Algorithm 1. This is determined in lines 8-14, where all output variables are collected (output variables are variables that gets assigned by the `output` operation) and added to two sets V (a set of output variables related to n) and V' (a set of output variables not related to n). Upon collecting all output variables, Algorithm 2 makes another pass over all statements in the `tbody` to identify six kinds of operations: `union`, `intersection`, `add`, `addAll`, `remove`, and `removeAll`. These operations are defined in Table 3.1[2]. In lines 16-18, the algorithm looks for `union` operation, where one of the variables involved is an output variables related to `n` and the other variable involved is not related to `n`. These conditions are simply the true conditions for the data-flow sensitivity, where the output of the current node is computed using the outputs of other nodes (neighbors). Similarly, in lines 19-21, the algorithm looks for `intersection` operation. The lines 22-27, identifies add and remove operations that adds or removes elements from the output related to node `n`. Finally, if there exists `union` and `add` operations, the output of a node always expands, and if there exists `intersection` and `remove` operations, the output of a node always shrinks. For a data-flow traversal to be loop sensitive, the output of nodes must either expand or shrink, not both (lines 28-29).

3.2.5 Graph Cyclicity

So far we have described the two static properties of the analysis that influences the traversal strategy selection. A property of the input graph also influences the selection. This property is the cyclicity in the graph. Based on the cyclicity, we classify graphs into four categories: {sequential, branch only, loop w/o branch, loop w/ branch}. In case of sequential graphs, all nodes in the graph have no more than one successor and predecessor. In case of graphs with branches, nodes may have more than one successor and predecessor. In case of graphs with

[2]The operations not listed here do not expand or shrink the output.

loops, there exists cycles in the graph. The graph cyclicity is determined during the construction of the graph.

In a source code analysis, traversal output of nodes may depend on each other. For instance, in forward data-flow analysis, output of a node is computed using the outputs of its predecessors. Similarly, in the backward data-flow analysis, output of the successors is required. Graph cyclicity plays an important role in the selection of the appropriate traversal strategy. In case of graphs with branches and loops, the outputs of all dependent nodes of a node (predecessors or successors) may not be available at the time of visiting the node, hence a traversal strategy must be selected that guarantees that the outputs of all dependent nodes of a node are available prior to computing the node's output.

3.3 Traversal Strategies - Candidates

We have picked seven traversal strategies as candidates for choosing an optimal traversal strategy for given a traversal and an input graph. The selected candidate strategies were arrived at by carefully reviewing compilers textbooks, implementations, and source code analysis frameworks. We also made sure that the selected candidate strategies are applicable to any graphs and analysis. We did not consider strategies like chaotic iteration based on Weak Topological Ordering because they are effective only for computing fixed points of continuous function over lattices of infinite height Bourdoncle (1993). The selected traversal strategies are describe below:

- **Any order (*ANY*)**: In this traversal strategy, nodes can be visited in any order. In our implementation, we visit the nodes in the order they appear in the nodes list N (Definition 1).

- **Increasing order of node ids (*INC*)**: In this traversal strategy, the nodes are visited in the increasing order of their node ids. The node ids are assigned during the construction of the graph. For instance, while constructing a CFG, the node ids are assigned in the control flow order.

- **Decreasing order of node ids (*DEC*)**: In this traversal strategy, the nodes are visited in the reverse order of their node ids (decreasing order of node ids).

- **Post-Order (*PO*)**: In this traversal, the successors of any node are visited before visiting the node.

- **Reverse Post-Order (*RPO*)**: In this traversal, the predecessors of any node are visited before visiting the node.

- **Worklist with Post-Order (*WPO*)**: In this traversal, the nodes are visited in the order they appear in the worklist. A worklist is a data structure used to keep track of nodes to be visited. In WPO, worklist is initialized with post-ordering of nodes. The worklist is maintained as follows: whenever a node from the worklist is removed and visited, all its successors (for forward traversals) or predecessors (for backward traversals) are added to the worklist as done in Atkinson and Griswold (2001).

- **Worklist with Reverse Post-Order (*WRPO*)**: In this traversal, the nodes are visited in the order they appear in the worklist. The worklist is initialized with nodes in the reverse post-order.

3.4 Decision Tree for Traversal Strategy Selection

At this point, we know the factors that influence the traversal strategy selection: the static properties of the analysis, and the runtime property of the graph. Our goal is to check these properties in certain order to quickly decide the best traversal strategy for a given analysis and a graph, such that only relevant properties are checked and the overhead of static/runtime check is minimized[3]. To that end, we carefully devised a decision tree as shown in Figure 3.3 for traversal strategy selection.

The leaf nodes of the tree are one of the seven traversal strategies and non-leaf nodes are static/runtime checks. The decision tree has eleven paths marked P_1 through P_{11}. Given a

[3]Our evaluation shows that the overhead is less than 0.2% of the total running time for a large dataset and less than 0.01% for an ultra-large dataset.

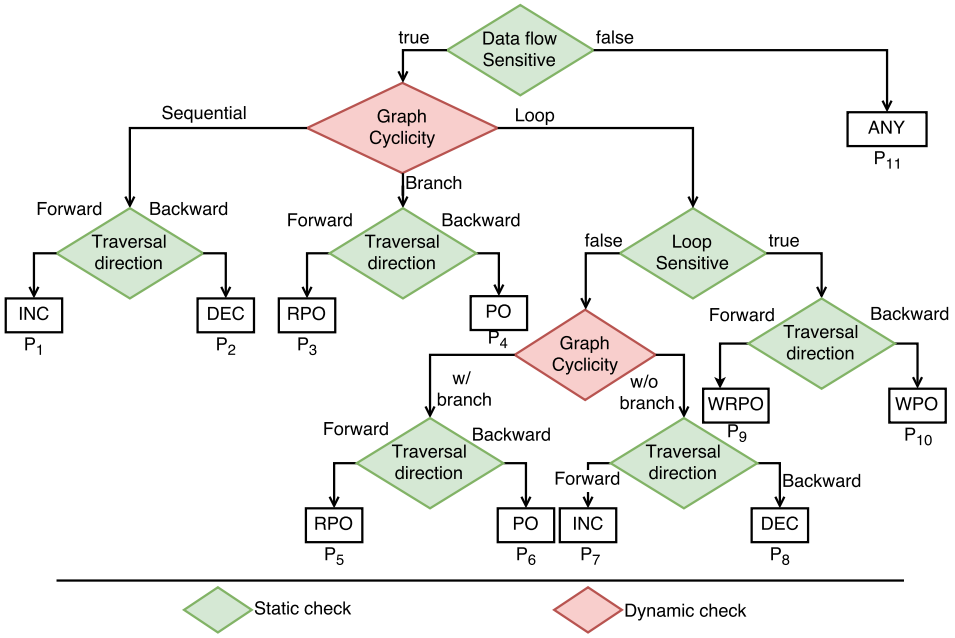


Figure 3.3: Traversal strategy selection decision tree.

traversal and an input graph, one of the eleven paths will be taken to decide the best traversal strategy. The longest paths P_5, P_6, P_7, and P_8 requires five checks and the shortest path P_{11} requires only one check. The static checks are marked green and the runtime checks are marked red. The static properties that are checked are: data-flow sensitivity ($P_{DataFlow}$), loop sensitivity (P_{Loop}), and traversal direction. The runtime property that is checked is the graph cyclicity: sequential, branch, loop w/ branch, and loop w/o branch. We now provide rationale for arranging the decision tree as shown in Figure 3.3.

The first property that is checked is the data-flow sensitivity of the traversal. This is a static property determined by analyzing the traversal that indicates whether the traversal output of any node is dependent on the traversal output of its neighbors (successors or predecessors). This property is defined in Definition 2 and an algorithm to compute this property is given in Algorithm 1. The rationale for checking this property first is that, if a traversal is data-flow insensitive ($P_{DataFlow}$ is *false*), irrespective of the type of the input graph, the traversal can finish in a single iteration (no fixpoint computation is necessary). In such cases, visiting nodes in any order will be efficient, hence we assign any order (ANY) traversal strategy (path P_{11}).

For traversals that are data-flow sensitive ($P_{DataFlow}$ is *true*), further checks are performed

to determine the best traversal strategy. Next property that is checked is the input graph cyclicity. This is because the loop sensitive property is applicable to only graphs with loops.

- *Sequential Graphs (paths P_1 and P_2)*: In this type of graphs, no branches or loops exists, and all nodes have a single successor and predecessor. At this point, we know that the traversal is data-flow sensitive and it requires output of the neighbors to compute the output for any node. As sequential graphs have only one neighbor (successor or predecessor), a traversal strategy that visits the neighbor prior to visiting any node is sufficient to produce an optimal traversal order. To determine which neighbor (successor or predecessor), we check the traversal direction property. For `FORWARD` traversal direction, predecessor of the node must be visited before the node and for `BACKWARD` traversal direction, successor of the node must be visited before the node. These two traversal orders are provided by our *INC* and *DEC* traversal strategies. The corresponding paths in the decision tree are: P_1 and P_2.

- *Graphs with branches (paths P_3 and P_4)*: In this type of graphs, branches exists, however loops don't exists, which means that a node may have more than one successor or predecessor. At this point, we know that our traversal is data-flow sensitive and it requires output of all neighbors (successors or predecessors) to compute the output for any node, we need a traversal order that ensures that all successors and predecessors are visited prior to visiting any node. This traversal order is given by the post-order (PO) and reverse post-order (RPO) traversal strategies. To pick between PO and RPO, we check the traversal direction. For `FORWARD` traversal direction, we need to visit all predecessors of any node prior to visiting the node, hence we pick the RPO traversal strategy. For `BACKWARD` traversal direction, we need to visit all successors of any node prior to visiting the node, hence we pick the PO traversal strategy.

- *Graphs with loops (paths P_5 to P_{10})*: In this type of graphs, loops exists and in addition branches may also exists. We first need to check if the traversal is sensitive to the loop (the loop sensitive property). At this point, we know that our analysis is data-flow sensitive and the input graph has loop based control flow.

– *Loop sensitive (paths P_9 and P_{10})*: When the traversal is loop sensitive, for correctly propagating the output, the traversal visits nodes multiple times until a fix point condition is satisfied (user may provide a fix point function). No iterative traversal strategy can guarantee that fix point will be reached in a single traversal of the nodes, hence we adopt a worklist based traversal strategy that visits only required nodes (property of the worklist strategy). The worklist traversal strategy requires that the worklist (a data structure) is initialized with nodes. For picking the best order of nodes for initialization, we further investigate the traversal direction. We know that, for `FORWARD` traversal direction, *RPO* traversal strategy gives the best order of nodes and for `BACKWARD` traversal direction, *PO* traversal strategy gives the best order of nodes, we pick worklist strategy with reverse post-order *WRPO* for `FORWARD` and worklist strategy with post-order *WPO* for `BACKWARD` traversal directions.

– *Loop insensitive (paths P_5 to P_8)*: When the traversal is loop insensitive, the selection problem reduces to the sequential and branch case that is discussed previously, because for loop insensitive traversal, the loops in the input graph is irrelevant and what remains relevant is the existence of the branches.

3.4.1 An Example

In this section we explain the decision tree using an example source code analysis and a graph. The example analysis that we choose is the *Post Dominator Analysis* shown in Listing 3.1 and the graph that we choose is shown in Figure 3.2. The *Post Dominator Analysis* contains two traversals: `initT` and `domT`. The `initT` traversal is data-flow insensitive ($P_{DataFlow}$ is *false*) and loop insensitive (P_{Loop} is *false*). The `domT` traversal is data-flow sensitive ($P_{DataFlow}$ is *true*) and loop insensitive (P_{Loop} is *false*). The traversal directions of `initT` and `domT` are `ITERATIVE` and `BACKWARD` respectively. Our example graph shown in Figure 3.2 has branches and loops, meaning the graph has nodes with more than one successor or predecessor and has cycles.

For selecting the best traversal strategies for `initT` and the graph with loops and branches, we check the data-flow sensitivity property of the traversal. As `initT` is data-flow insensitive, *ANY* traversal strategy is picked as shown by the path P_{11} in Figure 3.3 and no further checks

are required. The traversal strategy ANY represents nodes visited in any order.

For selecting the best traversal strategies for `domT` and the graph with branch and loop, we check the data-flow sensitivity property of the traversal. As `domT` is data-flow sensitive, the next property to be checked is the graph cyclicity. As our input graph has loops, the next property to be checked is whether `domT` is loop sensitive ie sensitive to the loops present in the graph. Since `domT` is loop-insensitive, we ignore the loops present in the graph and investigate the rest of the graph structure. As our graph contains branches, the next property to be checked is the traversal direction. The traversal direction for `domT` is `BACKWARD`, we pick PO traversal strategy for `domT` traversal, as shown by the path P_6 in Figure 3.3. The traversal strategy post-order (PO) visits all successors of a node before visiting that node. This is most suitable for backward analysis like `domT`, because backward analysis analyzes successors of a node prior to analyzing the node.

3.5 Optimizing the Selected Traversal Strategy

The properties of the analysis and the input graph not only helps us select the best traversal strategy, it also helps to perform several optimizations to the selected traversal strategies.

Running an analysis on a cyclic graph may require multiple iterations of the nodes to compute the fixpoint solution. At least one additional traversal of the nodes is required before the fixpoint check that compares the result of the last two traversals to to ensure that results at nodes have stabilized. This additional traversal and the fixpoint checks at nodes can be eliminated based on the selected traversal strategy.

For data-flow insensitive traversals, we know that the traversal outputs of nodes does not depend on the outputs of other nodes, hence both the additional traversal and the fixpoint checks can be eliminated irrespective of the graph cyclicity (path P_{11} in our decision diagram Figure 3.3). In case of data-flow sensitive traversals, the traversal output of a node is computed using the traversal outputs of other nodes (predecessors or successors). For data-flow sensitive analysis on a acyclic graph provides an opportunity to eliminate the additional traversal and fixpoint checks iff the selected traversal strategy guarantees that the nodes whose outputs are required to compute the output of a node are visited prior to visiting the node (paths P_1 to P_4).

In case of data-flow traversals, but loop insensitive traversals, and acyclic graphs, the additional traversal and fixpoint checks can be eliminated iff the selected traversal strategy guarantees that the nodes whose outputs are required to compute the output of a node are visited prior to visiting the node (paths P_5 to P_8). The optimizations does not apply for traversals that are both data-flow and loop sensitive, and the graph has cycles (paths P_9 and P_{10}). For such traversals, our technique selects the worklist-based traversal strategy which must perform fixpoint checks every time a node is visited.

CHAPTER 4. Empirical Evaluation

We conducted an empirical evaluation on a set of 21 basic source code analyses and 2 public massive code datasets to evaluate several factors about our hybrid approach for selecting and optimizing traversal strategies. First, we show the benefit of using our optimized selected traversal over standard ones by evaluating the *reduction in running times* of our hybrid approach over the standards ones (§4.2). Then, we evaluate the correctness of the analysis results using our hybrid approach to show that the decision analyses and optimizations in our approach does not affect the correctness of the source code analyses (§4.3). We also evaluate the precision of our selection algorithm by measuring how often the hybrid approach selects the most time-efficient traversal (§4.4). Finally, we evaluate how the different components in the approach and different kinds of static and runtime properties impact the overall performance? This is done in §4.5 and §4.6, and via various insight analysis of our results.

4.1 Analyses, Datasets and Experiment Setting

4.1.1 Analyses.

We collected source code analyses that traverse control flow graphs from textbooks and source code analysis tools. We also made sure that the analyses list covers all the static properties discussed in §3.2, i.e., data-flow sensitivity, loop sensitivity and traversal direction. We ended up with 21 source code analyses as shown in Table 4.1. They include 10 basic ones (analyses 1, 2, 8, 9, 10, 11, 12, 14, 15 and 19) from textbooks Aho et al. (2006); Nielson et al. (2010) and 11 tothers for detecting source code bugs, and code smells from the Soot framework Vallée-Rai et al. (1999) (analyses 3, 4, 5, 13, 17 and 18), and FindBugs tool Ayewah et al. (2007) (analyses 6, 7, 16, 20 and 21). Table 4.1 also shows the number of traversals

Table 4.1: List of source code analyses and the properties of their involved traversals.

	Analysis	Ts	t_1			t_2			t_3		
			Flw	Lp	Dir	Flw	Lp	Dir	Flw	Lp	Dir
1	Copy propagation (CP)	3	✗	✗	—	✓	✓	→	✗	✗	—
2	Common sub-expression detection (CSD)	3	✗	✗	—	✓	✓	→	✗	✗	—
3	Dead code (DC)	3	✗	✗	—	✓	✓	←	✗	✗	—
4	Loop invariant code (LIC)	3	✗	✗	—	✓	✓	→	✗	✗	—
5	Upsafety analysis (USA)	3	✗	✗	—	✓	✓	→	✗	✗	—
6	Valid FileReader (VFR)	3	✗	✗	—	✓	✓	→	✗	✗	—
7	Mismatched wait/notify (MWN)	3	✗	✗	—	✓	✓	→	✗	✗	—
8	Available expression (AE)	2	✗	✗	—	✓	✓	→			
9	Dominator (DOM)	2	✗	✗	—	✓	✗	→			
10	Local may alias (LMA)	2	✗	✗	—	✓	✓	→			
11	Local must not alias (LMNA)	2	✗	✗	—	✓	✓	→			
12	Live variable (LV)	2	✗	✗	—	✓	✓	←			
13	Nullness analysis (NA)	2	✗	✗	—	✓	✓	→			
14	Post dominator (PDOM)	2	✗	✗	—	✓	✗	←			
15	Reaching definition (RD)	2	✗	✗	—	✓	✓	→			
16	Resource status (RS)	2	✗	✗	—	✓	✓	→			
17	Very busy expression (VBE)	2	✗	✗	—	✓	✓	←			
18	Safe Synchronization (SS)	2	✗	✗	—	✓	✓	→			
19	Used and defined variable (UDV)	1	✗	✗	—						
20	Useless increment in return (UIR)	1	✗	✗	—						
21	Wait not in loop (WNIL)	1	✗	✗	—						

Table 4.2: Statistics of the generated control flow graphs from two datasets.

Dataset	All graphs	Sequential	Branches	Loops		
				All graphs	Branches	No branches
DaCapo	287K	186K (65%)	73K (25%)	28K (10%)	21K (7%)	7K (2%)
GitHub	161,523K	111,583K (69%)	33,324K (21%)	16,617K (10%)	11,674K (7%)	4,943K (3%)

each analysis contains and their static properties as described in §3.2. The sets of traversals cover all types of static properties for flow-sensitivity, loop-sensitivity and direction (forward, backward and iterative). All analyses are intra-procedural. We implemented all twenty one of these analysis using constructs described in §3.1. In Table 4.1, Ts denotes total number of traversals, t_i denotes properties of traversal i-th, *Flw* denotes data-flow sensitive, *Lp* denotes loop sensitive, *Dir* denotes traversal direction where —, → and ← mean iterative, forward and backward, respectively.

4.1.2 Datasets.

We ran the analyses on two datasets: DaCapo 9.12 benchmark Blackburn et al. (2006), DaCapo for short, and a ultra-large-scale dataset containing projects from GitHub, GitHub for short. DaCapo dataset contains the source code of 10 open source Java projects: Apache Batik,

Table 4.3: Time contribution of each phase (in miliseconds).

Analysis	Avg. Time		Static	Runtime			
	DaCapo	GitHub		DaCapo		GitHub	
				Avg.	Total	Avg.	Total
CP	0.21	0.008	53	0.21	62,469	0.008	1359K
CSD	0.19	0.012	60	0.19	56,840	0.012	1991K
DC	0.19	0.010	45	0.19	54,822	0.010	1663K
LIC	0.21	0.006	69	0.20	60,223	0.006	992K
USA	0.19	0.006	90	0.19	54,268	0.009	1444K
VFR	0.18	0.007	42	0.18	52,483	0.007	1142K
MWN	0.18	0.006	36	0.18	52,165	0.006	1109K
AE	0.18	0.007	43	0.18	53,290	0.007	1169K
DOM	0.21	0.008	35	0.21	62,416	0.008	1307K
LMA	0.18	0.008	76	0.18	52,483	0.008	1346K
LMNA	0.18	0.008	80	0.18	53,182	0.008	1407K
LV	0.17	0.007	32	0.17	49,231	0.007	1273K
NA	0.16	0.008	64	0.16	46,589	0.008	1398K
PDOM	0.20	0.012	34	0.20	57,203	0.012	2040K
RD	0.20	0.007	48	0.20	57,359	0.007	1155K
RS	0.16	0.006	28	0.16	46,367	0.006	996K
VBE	0.17	0.006	44	0.17	49,138	0.006	1062K
SS	0.17	0.006	32	0.17	48,990	0.006	1009K
UDV	0.14	0.005	10	0.14	41,617	0.005	928K
UIR	0.14	0.006	14	0.14	41,146	0.006	1020K
WNIL	0.14	0.007	15	0.14	41,808	0.007	1210K

Apache FOP, Apache Aurora, Apache Tomcat, Jython, Xalan-Java, PMD, H2 database, Sunflow and Daytrader. GitHub dataset contains the source code of more than 380K Java projects collected from GitHub.com. Each method in the datasets was used to generate a control flow graph (CFG) on which the analyses would be run. The statistics of the two datasets are shown in Table 4.2. DaCapo dataset contains 287K non-empty CFGs while GitHub dataset contains more than 162M. Both have similar distributions of CFGs over graph cyclicity. Most CFGs are sequential and only 10% have loops.

4.1.3 Setting.

We compared our *hybrid* approach against the six standard traversal strategies in §3.3: DFS, PO, RPO, WPO, WRPO and ANY. The running time for each analysis is measured from the start to the end of the analysis which includes constructing CFGs and traversing CFGs. The running time for our hybrid approach also includes the time for computing the static and runtime properties, making the traversal strategy decision, optimizing it and then using the optimized traversal strategy to traverse the CFG and run the analysis. The analyses on DaCapo dataset were run on a single machine with 24 GB of memory and 24-cores, running on Linux

3.5.6-1.fc17 kernel. Running analyses on GitHub dataset on a single machine would take weeks to finish, so we run them on a distributed cluster which runs a standard Hadoop 1.2.1 with 1 name and job tracker node, 10 compute nodes with totally 148 cores, and 1 GB of RAM for each map/reduce task.

4.2 Running Time and Time Reduction

We first report the running times and then study the achieved reductions against standard traversal strategies.

4.2.1 Running Time

Table 4.3 shows the running times for 21 analyses on the two datasets. On average (column **Avg. Time**), each analysis took 0.14–0.21 ms and 0.005–0.012 ms to analyze a graph in Dacapo and GitHub datasets, respectively. The variation in the average analysis time is mainly due to the difference in the machines used to run the analysis for DaCapo and GitHub datasets, as described in Section 3.1. Also, the average sizes of graphs in DaCapo are much larger than the average sizes of the graphs in the GitHub. Columns **Static** and **Runtime** show the time contributions for different components of the hybrid approach: the time for determining the static properties of each analysis which is done once for each analysis, and the time for constructing the CFG of each method and traversing the CFG which is done once for every constructed CFG. We can see that the time for collecting static information is negligible, less than 0.2% for DaCapo dataset and less than 0.01% for GitHub dataset, when compared to the total runtime information collection time, as it is performed only once per traversal. When compared to the average runtime information collection time, the static time is quite significant. However, the overhead introduced by static information collection phase diminishes as the number of CFGs increases and becomes insignificant when running on those two large datasets. This result shows the benefit of our hybrid approach when applying on ultra-large-scale analysis.

Analysis	DaCapo						GitHub					
	DFS	PO	RPO	WPO	WRPO	ANY	DFS	PO	RPO	WPO	WRPO	ANY
CP	17%	83%	9%	66%	11%	72%	17%	88%	12%	80%	5%	82%
CSD	41%	93%	39%	74%	4%	89%	31%	–	24%	–	12%	–
DC	41%	30%	89%	7%	64%	81%	25%	22%	–	7%	–	–
LIC	17%	84%	8%	67%	7%	73%	19%	89%	15%	81%	19%	88%
USA	36%	92%	34%	72%	9%	87%	22%	–	17%	–	9%	–
VFR	20%	41%	18%	51%	15%	62%	15%	40%	10%	44%	9%	53%
MWN	21%	35%	16%	35%	22%	49%	17%	31%	12%	33%	11%	46%
AE	40%	14%	39%	73%	14%	87%	16%	–	16%	–	11%	–
DOM	54%	97%	48%	70%	6%	95%	27%	–	32%	–	6%	–
LMA	35%	46%	28%	74%	6%	46%	22%	–	13%	–	6%	–
LMNA	29%	39%	22%	68%	9%	41%	21%	–	15%	–	7%	–
LV	38%	30%	84%	11%	56%	75%	25%	21%	68%	11%	69%	80%
NA	26%	88%	30%	50%	10%	80%	13%	87%	12%	71%	10%	85%
PDOM	51%	41%	95%	10%	72%	95%	24%	20%	–	24%	–	–
RD	15%	80%	7%	62%	9%	68%	19%	91%	10%	79%	5%	86%
RS	31%	31%	30%	31%	28%	30%	16%	40%	9%	31%	7%	49%
VBE	40%	36%	88%	13%	76%	81%	28%	24%	–	10%	–	–
SS	26%	39%	22%	37%	25%	57%	20%	35%	13%	34%	10%	50%
UDV	6%	5%	6%	10%	9%	3%	3%	4%	2%	7%	6%	0%
UIR	2%	2%	1%	3%	3%	0%	2%	5%	4%	7%	7%	0%
WNIL	3%	4%	5%	6%	8%	2%	3%	6%	5%	5%	6%	0%
Overall	31%	83%	70%	55%	35%	81%	–	–	–	–	–	–

(a) Time reduction for each analysis.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	ANY
Data-flow	32%	84%	72%	57%	36%	83%
¬Data-flow	4%	4%	4%	6%	6%	2%

(b) Overall reduction over analysis properties.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	ANY
Sequential	20%	74%	63%	55%	28%	72%
Branch	31%	81%	66%	58%	40%	92%
Loop	53%	88%	75%	62%	37%	95%

(c) Overall reduction over graph properties.

Figure 4.1: Reduction in running times.

4.2.2 Time Reduction

To evaluate the efficiency in running time of the hybrid approach over other traversal strategies, we ran the 21 analyses on DaCapo and GitHub datasets using hybrid approach and other candidate traversals. When comparing the hybrid approach to a standard strategy S, we computed the reduction rate $R = (T_S - T_H)/T_S$ where T_S and T_H are the running times using the standard and the hybrid strategy, respectively. Some analyses have some worst case traversal strategies which might not be feasible to run on dataset at the scale of 162 million graphs as in GitHub dataset. For example, using post-order for forward data-flow analysis will visit the CFG in the direction which is opposite to the natural direction of the analysis and hence takes a lot of time to complete the analysis. For such combinations of analyses and

traversal strategies, the map and the reducer tasks time out in the cluster setting and, thus, we did not provide the running times. The corresponding cells in Figure 4.1a are denoted with symbol –.

The result in Figure 4.1a shows that the hybrid approach helps reduce the running times in almost all cases. The values indicates the reduction in running time by adopting hybrid approach compared against the standard strategies. Most of positive reductions are from 10% (light yellow cells) or even from 50% (light green cells). More importantly, the most time-efficient and the worst traversal strategies vary across the analyses which supports the need of our hybrid traversal strategy. Over all analyses, the reduction was highest against any order and post-order (PO and WPO) strategies. The reduction was lowest against the strategy using depth-first search (DFS) and worklist with reverse post-ordering (WRPO). When compared with the next best performing traversal strategy for each analysis, hybrid approach reduces the overall execution time by about 13 minutes to 72 minutes on GitHub dataset. We do not report the overall numbers for GitHub dataset due to the presence of failed runs.

Figure 4.1b shows time reductions for different types of analyses. For *data-flow sensitive* ones, the reduction rates are high ranging from 32% to 84%. The running time was not improved much for *non data-flow sensitive* traversals, which correspond to the last three rows in Figure 4.1a with mostly one digit reductions (light orange cells). We actually perform almost as same as Any order traversal strategy for analyses in this category. This is because Any order traversal strategy is the best strategy for all the CFGs in these analyses. Hybrid approach also chooses any order traversal strategy and hence there is no scope for performance gain.

Figure 4.1c shows time reduction for different cyclicity types of input graphs. We can see that reductions over graphs with loops is highest and those over any graphs is lowest.

4.2.3 Time reduction against hand optimized analysis

Another way to extract more performance is to hand optimize the analysis. Figure 4.2 compares Hybrid approach against hand optimized analysis. Hand optimized analysis has single best optimized traversal strategy applied for each analysis. For data-flow analysis, hand optimized analysis uses WPO/WRPO guideline while for non data flow analysis, it uses ANY

Analysis	DaCapo	GitHub
CP	9%	5%
CSD	4%	12%
DC	7%	7%
LIC	7%	19%
USA	9%	9%
VFR	15%	9%
MWN	15%	9%
AE	14%	11%
DOM	6%	6%
LMA	6%	6%
LMNA	9%	7%
LV	11%	11%
NA	10%	10%
PDOM	10%	24%
RD	9%	5%
RS	28%	7%
VBE	13%	10%
SS	22%	10%
UDV	3%	0%
UIR	0%	0%
WNIL	2%	0%

Figure 4.2: Reduction in running times against hand optimized analysis.

traversal strategy, as it is the best traversal strategy for non data-flow analysis. WPO/WRPO guideline suggests that if the direction of the traversal is backward, use WPO else use WRPO. This guideline simplifies the hybrid approach, where the decision is based on only traversal direction while Hybrid approach uses the decision tree in Figure 3.3. Figure 4.2 shows the comparison of hybrid approach against hand optimized analysis. We can see that for about half of the data-flow analysis, we gain at least 10% reduction. For analysis like RS, SS, VFR and MWN, the gain is much higher since these analyses are selective in terms of the program statements that they analyze. WPO and WRPO would not work for these analyses since in case of both WPO and WRPO, there exists a fixed cost of creating and maintaining a worklist of size equals to the number of CFG nodes. When analyses selectively analyzes a small subset of all nodes in the CFGs, this overhead becomes substantial. Our hybrid strategy is not only able to select an alternative strategy instead of WPO/WRPO for such selective analyses, whenever possible (when the input graphs do not contain loops), but also optimize WPO/WRPO (in case of input graphs with loops), such that the overheads are minimized. For non-data flow analysis, there is no gain against hand optimized analysis since ANY is the best traversal strategy for such analysis and both Hybrid approach and hand optimized analysis applied ANY traversal strategy for all the graphs for non-data flow analysis.

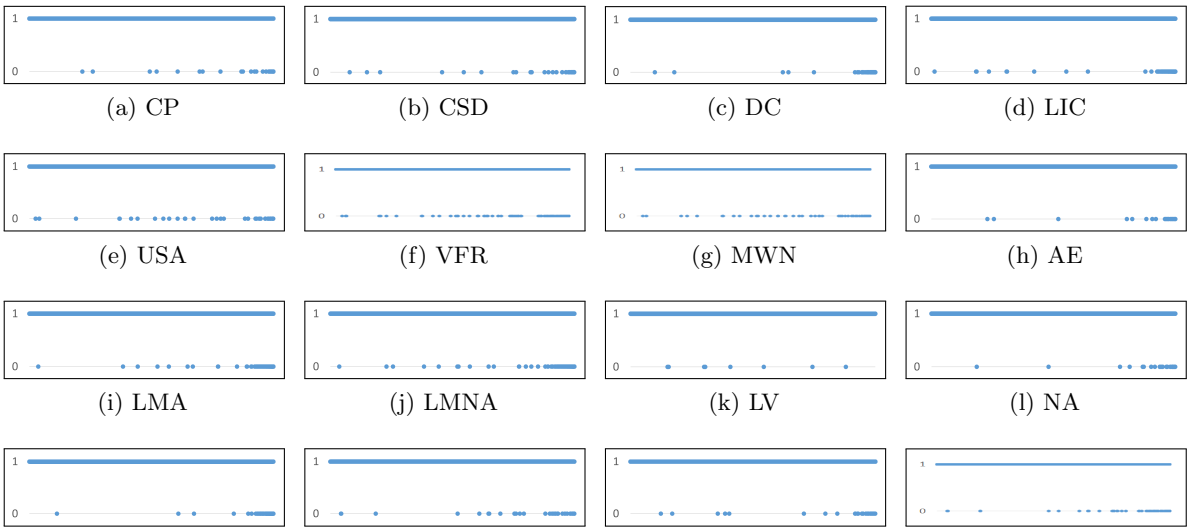
Table 4.4: Traversal strategy prediction precision.

Analysis	Precision
DOM, PDOM, WNIL, UDV, UIR	100.00%
CP, CSD, DC, LIC, USA, VFR, MWN, AE, LMA, LMNA, LV, NA, RD, RS, VBE, SS	99.99%

4.3 Correctness of Analysis Results

To evaluate the correctness of analysis results, we first chose worklist as standard strategy to run analyses on DaCapo dataset to create the groundtruth of the results. We then ran analyses using our hybrid approach and compared the results with the groundtruth. In all analyses on all input graphs from the dataset, the results from our hybrid approach always exactly matched the corresponding ones in the groundtruth.

4.4 Traversal Strategy Selection Precision



(a) CP (b) CSD (c) DC (d) LIC

(e) USA (f) VFR (g) MWN (h) AE

(i) LMA (j) LMNA (k) LV (l) NA

(m) RD (n) RS (o) VBE (p) SS

Figure 4.3: Scatter charts for analyses that have loop sensitive traversals.

In this experiment, we evaluated how well the hybrid approach picks the most time-efficient strategy. We ran the 21 analyses on the DaCapo dataset using all the candidate traversals and the one selected by the hybrid approach. One selection is counted for each pair of a traversal and an input graph where the hybrid approach selects a traversal strategy based on the properties

of the analysis and input graph. A selection is considered correct if its running time is at least as good as the running time of the fastest among all candidates. The precision is computed as the ratio between the number of correct selections over the total number of all selections. As shown in Table 4.4, the selection precision is 100% for all analyses that are not *loop sensitive*. For analyses that involve *loop sensitive* traversals, the prediction precision is 99.99%.

We further analyzed the result to see what contributed to these mispredictions. Let us break the CFGs in the DaCapo dataset by the graph cyclicity: sequential CFGs, CFGs with branches and no loops, and CFGs with loops, and discuss the selection precision.

For **sequential CFGs & CFGs with branches and no loops**, the selection precision is 100%—the hybrid approach always picks the most time-efficient traversal strategy.

For **CFGs with loops**, the selection precision is 100% for *loop insensitive* traversals. The mispredictions occur with *loop sensitive* traversals on CFGs with loops. Figure 4.3 shows scatter charts for the traversal selection results for 16 analyses that are *loop sensitive*. In the chart, 1 indicates a correct selection and 0 indicates a misprediction. CFGs are organized along the x- axis in the increasing order of their sizes measured as the numbers of nodes. The scatter charts show that the mispredictions tend to happen with larger CFGs. The reason is that, for *loop sensitive* traversals, the hybrid approach picks worklist as the best strategy. The worklist approach was picked because it visits only as many nodes as needed when compared to other traversal strategies which visit redundant nodes. However using worklist imposes an overhead of creating and maintaining a worklist containing all nodes in the CFG. This overhead is negligible for small CFGs. However, when running analyses on large CFGs, this overhead could become higher than the cost for visiting redundant nodes. Therefore, selecting worklist for *loop sensitive* traversals on large CFGs might not always result in the best running times.

Figure 4.4 shows the Hybrid approach's performance against best approaches for mis-predicted graphs for 16 analyses that has *loop sensitive* traversals. We can see that for majority of the mis-predicted graphs, Hybrid approach's performance is comparable to the best approaches.

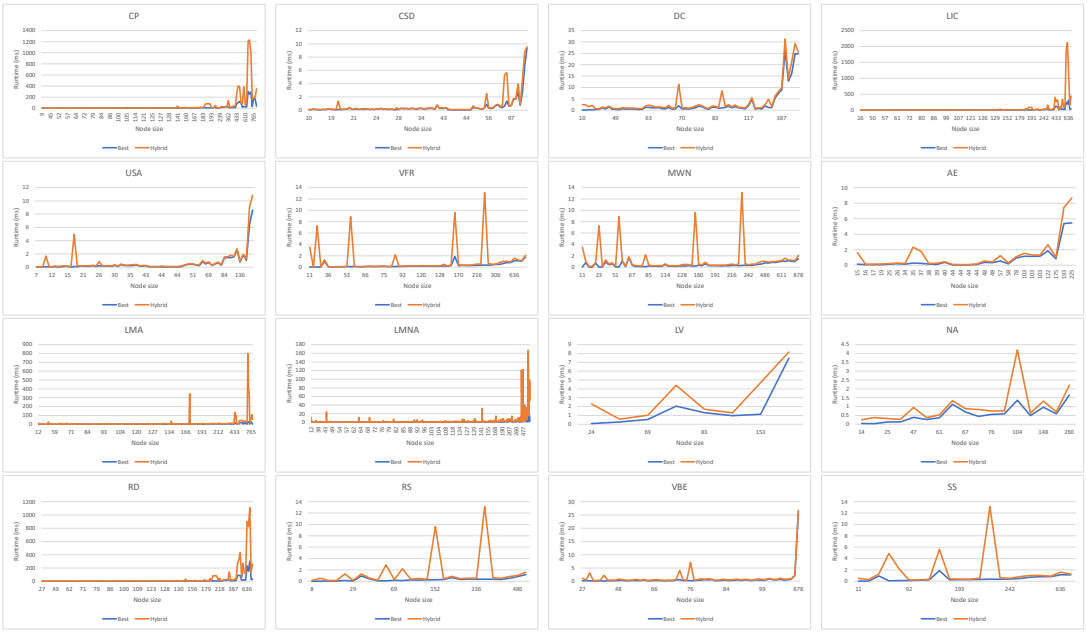


Figure 4.4: Hybrid approach's performance against best approaches for mis-predicted graphs.

4.5 Analysis on the Decision Tree Distribution

Decision tree is the key component in our hybrid approach. Given an analysis traversal and an input graph, a path along the check points in the tree will be used to determine the traversal strategy at the corresponding leaf node. There are such 11 paths leading to 11 leaf nodes as shown in Figure 3.3. In this experiment, we want to study the contribution of each path in determining strategies for CFGs from the two datasets. Two tables in Figure 4.5 show the result for 21 analyses. Background colors indicate the ranges of values: 0%, (0%, 1%), [1%, 10%) and [10%, 100%]. The result shows a trend which is consistent between two datasets that 5 paths (P1, P2, P3 and P11) in the decision tree were used often—more than average. Paths P4, P9 and P10 are less frequently used and paths P5, P6, P7 and P8 were rarely used. These four paths were taken less often than the others because they are only used for CFGs with loops which are only 10% of the CFGs in the datatsets. In addition, these paths are taken when the traversal is data-flow sensitive and loop insensitive. Only two of our analyses contains such traversal. It is also worth to note that, from Figure 3.3, those four paths (P5–P8) are the longest paths in the tree. The fact that these longest paths are rare (less than 1% for both DaCapo and GitHub datasets) shows that most analyses and graphs are classified by our technique using

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
CP	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
CSD	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
DC	0%	35%	0%	10%	0%	0%	0%	0%	0%	5%	50%
LIC	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
USA	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
VFR	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
MWN	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
AE	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
DOM	69%	0%	21%	0%	7%	0%	3%	0%	0%	0%	0%
LMA	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
LMNA	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
LV	0%	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%
NA	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
PDOM	0%	69%	0%	21%	0%	7%	0%	3%	0%	0%	0%
RD	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
RS	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
VBE	0%	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%
SS	69%	0%	21%	0%	0%	0%	0%	0%	10%	0%	0%
UDV	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
UIR	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
WNIL	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
Overall	34.54%	9.87%	10.32%	2.95%	0.26%	0.26%	0.11%	0.11%	4.78%	1.10%	35.71%

Figure 4.5: Distribution of decisions over the paths of the decision tree.

fewer dynamic checks.

4.6 Analysis on Traversal Optimization

We evaluated the importance of optimizing the chosen traversal strategy by comparing the hybrid approach with the non-optimized version. We computed the reduction rate on the running times for the 21 analyses. Figure 4.7 shows the reduction in execution time due to traversal strategy optimization. For analyses that involve at least one *data-flow sensitive* traversal, the optimization helps to reduce at least 60% of running time. This is because optimizations in such traversals reduce the number of iterations of traversals over the graphs by eliminating the redundant result re-computation traversal and the unnecessary fixpoint condition checking traversal. For analyses involving only *data-flow insensitive* traversal, there is no reduction in execution time, as hybrid approach does not attempt to optimize.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
CP	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
CSD	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
DC	0%	32%	0%	13%	0%	0%	0%	0%	0%	5%	50%
LIC	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
USA	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
VFR	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
MWN	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
AE	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
DOM	65%	0%	25%	0%	7%	0%	2%	0%	0%	0%	0%
LMA	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
LMNA	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
LV	0%	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%
NA	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
PDOM	0%	65%	0%	25%	0%	7%	0%	2%	0%	0%	0%
RD	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
RS	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
VBE	0%	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%
SS	65%	0%	25%	0%	0%	0%	0%	0%	10%	0%	0%
UDV	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
UIR	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
WNIL	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
Overall	32.46%	9.27%	12.69%	3.62%	0.26%	0.26%	0.10%	0.10%	4.50%	1.04%	35.70%

Figure 4.6: Distribution of decisions over the paths of the decision tree for the DaCapo Dataset.

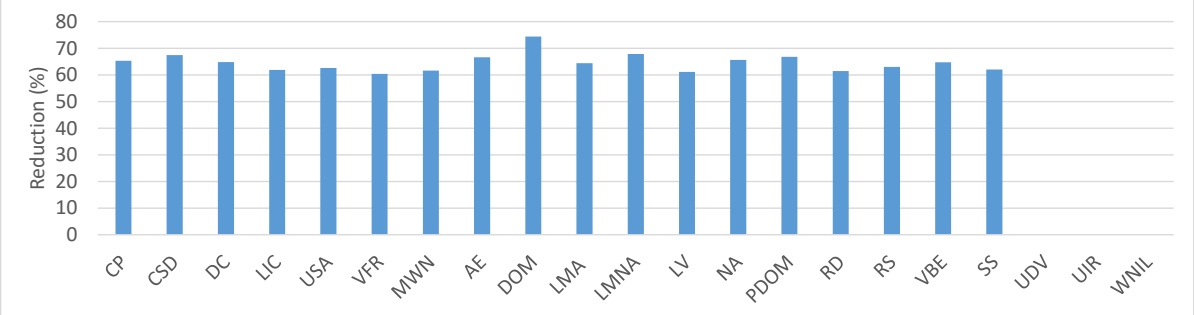


Figure 4.7: Reduction in execution time of the hybrid approach due to traversal optimization.

CHAPTER 5. Case Studies

We implemented three case studies using our formalism for source code analysis and evaluated using hybrid and WRPO traversal. We are comparing against only WRPO since it is the next best performing traversal.

API Precondition Mining (APM). This case study mines a large corpus of API usages to derive potential preconditions for API methods Nguyen et al. (2014). The key idea of this work is that API preconditions would be checked frequently in a corpus with a large number of API usages, while project-specific conditions would be less frequent. This case study analysis mined the preconditions for all methods of `java.lang.String`.

API Usage Mining (AUM). This case study analyzes API usage code and mines API usage patterns Zhong et al. (2009). The mined patterns help developers understand and write API usages more effectively with less errors. Our analysis mined usage patterns for `java.util` APIs.

Finding Security Vulnerabilities with Tainted Object Propagation (SVT). This case study formulated a variety of widespread SQL injections, as tainted object propagation problems Livshits and Lam (2005). Our analysis looked for all SQL injection vulnerabilities matching the specifications in the statically analyzed code.

Figure 5.1 shows that hybrid traversal helps reduce running times significantly by 80–175 minutes, which is from 6%–10% relatively.

Case	Hybrid	WRPO	Reduce
APM	1527	1702	10%
AUM	883	963	8%
SVT	1417	1501	6%

Figure 5.1: Running time (minutes) of the case studies on GitHub data.

CHAPTER 6. Threats to Validity

Our first threat to validity is our selection of source code analysis used in our evaluation. While there exists no source for a standard set of analysis, we relied mainly on text books and source code analysis tools to select analysis. We have selected basic control and data-flow analyses, and analyses to find bugs or code smells. We made sure to include analysis that covers all the properties of interest. For instance, our analysis set includes: both forward/backward analysis, data-flow sensitive and insensitive analysis, loop sensitive and insensitive analysis.

Our next threat to validity is our selection of ultra-large-scale datasets that provide graphs for running the analyses. The datasets do not contain a balanced distribution of different graph cyclicity (sequential, branch and loop). Both DaCapo and GitHub datasets contains majority of sequential graphs (65% and 69%, respectively) and only 10% are graphs with loops. The impact of this threat can be seen in our evaluation of the importance of paths and decisions in our decision tree. Paths and decisions along sequential graphs are taken more often. This threat is not easy to mitigate, as it is hard to find and difficult to expect a real-world code dataset to contain a balanced distribution of graphs of various types. Nonetheless, our evaluation shows that the selection and optimization of the best traversal strategy for these 35% of the graphs (graphs with branches and loops) plays an important role in improving the overall performance of the analysis over a large dataset of graphs.

CHAPTER 7. Related Work

To the best of our knowledge, our proposal to leverage information about the program analysis code, and the nature of the data on which analysis is applied to select appropriate traversal strategies has not been explored previously. Below we discuss works that are related to various aspects of our proposal.

Mixing static and dynamic information. The general philosophy of mixing static and dynamic information has a long history Ernst (2003) in both the software engineering and the programming languages communities, with examples such as DSD-Crasher Csallner et al. (2008), Palus Zhang et al. (2011), segmented symbolic execution Le (2013), guided dynamic symbolic execution Christakis et al. (2016), gradual typing Siek and Taha (2007) , hybrid type checking Flanagan (2006), intensional polymorphism Harper and Morrisett (1995); Crary et al. (1998), etc. While our proposal also mixes static information about program analysis with dynamic information about the data, none of the previous works have proposed utilizing this information for selecting appropriate traversal strategies for realizing the program analysis.

*Optimizing program analysis.*Atkinson Atkinson and Griswold (2001) presented techniques that reduce the time and space required to perform data-flow analysis of large programs. While their techniques proposed modifications to the underlying data-flow analyses that yield improvement in performance and also proposed reclamation of the data-flow sets during data-flow analysis that result in saving space, hybrid approach gives performance gain by analyzing the user written algorithm and the input graph received.

Kildall Kildall (1973) presented an algorithm which, in conjunction with various optimizing functions, provides global program optimization, Optimizing functions have been described which provide constant propagation, common sub-expression elimination, and a degree of register optimization. While their approach provides unified approach to global program optimization,

we concentrate on optimizing the process that does program optimization using the program's structure and the algorithms characteristics.

*Ultra-large-scale source code mining.*In terms of ultra large scale processing, Boa Dyer et al. (2013) is a language and infrastructure for analyzing ultra-large-scale software repositories. Boa provides a different kind of performance gain through its infrastructure and eases testing MSR-related hypotheses, it is not suitable for graph processing algorithms and does not leverage information from algorithms written in Boa.

*Graph traversal optimization.*There have been many works that targeted graph traversal optimization through various ways. Green-Marl Hong et al. (2012) provides performance benefits by using domain specific knowledge in applying optimizations. It uses high-level algorithmic description written in Green-Marl to exploiting the exposed data level parallelism. While green marl provides performance benefits by taking algorithm written into consideration, hybrid approach takes both algorithm and graph structure into account. And in ultra large scale dataset, containing millions of graphs with different structures, the gain that we can incur by taking graph structure into account is significant.

Pregel Malewicz et al. (2010) is a MapReduce like framework that aims to bring distributed processing to graph algorithms. While Pregel's performance gain is through parallelism and handles large graphs processing through vertex centric approach, our approach achieves performance gain by traversing the graph efficient suitable to the algorithm.

There have also been few libraries that support parallel or distributed graph analysis: Parallel BGL Gregor and Lumsdaine (2005) is a distributed version of BGL while SNAP Bader and Madduri (2008) is a stand-alone parallel graph analysis package.

CHAPTER 8. Conclusion

Improving the performance of source code analyses that runs on massive code bases is an ongoing challenge. One way to improve the performance of source code analysis expressed as traversals over graphs like CFGs, is by picking the optimal traversal strategy that defines the order of nodes visited. The selection of the best traversal strategy depends both on the properties of the analysis and the input graph on which the analysis is run. We proposed a hybrid technique for selecting and optimizing graph traversal strategies for source code analysis expressed as traversals over graphs. Our solution includes a system for expressing source code analysis as traversals, a set of static properties of the analysis and algorithms to compute them, a decision tree that checks static properties along with graph properties to select the most time-efficient traversal strategy. Our evaluation shows that the hybrid technique successfully selected the most time-efficient traversal strategy for 99.99%–100% of the time and using the selected traversal strategy and optimizing it, the running times of a representative collection of source code analysis in our evaluation were considerably reduced by 1%-28% (13 minutes to 72 minutes in absolute time) when compared against the best performing traversal strategy. The case studies show that hybrid traversal reduces 80–175 minutes in running times for three software engineering tasks. The overhead imposed by collecting additional information for our approach is less than 0.2% of the total running time for a large dataset and less than 0.01% for an ultra-large dataset.

Bibliography

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Atkinson, D. C. and Griswold, W. G. (2001). Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 52–, Washington, DC, USA. IEEE Computer Society.

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y. (2007). Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA. ACM.

Bader, D. A. and Madduri, K. (2008). SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA. ACM.

Bourdoncle, F. (1993). Efficient Chaotic Iteration Strategies With Widenings. In *Proceedings*

of the International Conference on Formal Methods in Programming and their Applications, pages 128–141. Springer-Verlag.

Christakis, M., Müller, P., and Wüstholz, V. (2016). Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 144–155, New York, NY, USA. ACM.

Crary, K., Weirich, S., and Morrisett, G. (1998). Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming*.

Csallner, C., Smaragdakis, Y., and Xie, T. (2008). Dsd-crasher : A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA. IEEE Press.

Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001). Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA. ACM.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27.

Flanagan, C. (2006). Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA. ACM.

Gregor, D. and Lumsdaine, A. (2005). The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18.

Harper, R. and Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA. ACM.

Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. (2012). Green-marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA. ACM.

Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. (1998). Single and Loving It: Must-alias Analysis for Higher-order Languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, New York, NY, USA. ACM.

Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA. ACM.

Le, W. (2013). Segmented symbolic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 212–221, Piscataway, NJ, USA. IEEE Press.

Li, Z., Lu, S., and Myagmar, S. (2006). CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, 32(3):176–192.

Livshits, B. and Zimmermann, T. (2005). Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM.

Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

Nguyen, H. A., Dyer, R., Nguyen, T. N., and Rajan, H. (2014). Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–177. ACM.

Nielson, F., Nielson, H. R., and Hankin, C. (2010). *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

Ramanathan, M. K., Grama, A., and Jagannathan, S. (2007). Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 240–250, Washington, DC, USA. IEEE Computer Society.

Siek, J. and Taha, W. (2007). Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP '07, pages 2–27, Berlin, Heidelberg. Springer-Verlag.

Thummalapenta, S. and Xie, T. (2009). Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE Computer Society.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot : Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press.

Weimer, W. and Necula, G. C. (2005). Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg. Springer-Verlag.

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. (2006). Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA. ACM.

Zhang, S., Saff, D., Bu, Y., and Ernst, M. D. (2011). Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363, New York, NY, USA. ACM.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending api usage patterns. *ECOOP 2009–Object-Oriented Programming*, pages 318–343.