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

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

Performance of program analysis expressed as traversals over graphs like control flow graph (CFG) heavily depends on the order of nodes visited during the traversals: *the traversal strategy*, more so in case of BigCode analysis that performs analysis over a large collection of input graphs. 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. This work proposes a hybrid technique that utilizes the static properties of the analysis, and the dynamic properties of the input graphs to select a most time-efficient strategy for each traversal on a graph. Our contributions are: a system for expressing program analysis as traversals, a set of static and dynamic properties that influence the traversal strategy selection, a set of static analyses to compute the properties, and a decision tree that checks the properties to select and optimize 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, the running times of the analyses on BigCode in our evaluation were considerably reduced by 23%–79%. The overhead imposed by collecting additional information for our hybrid 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.

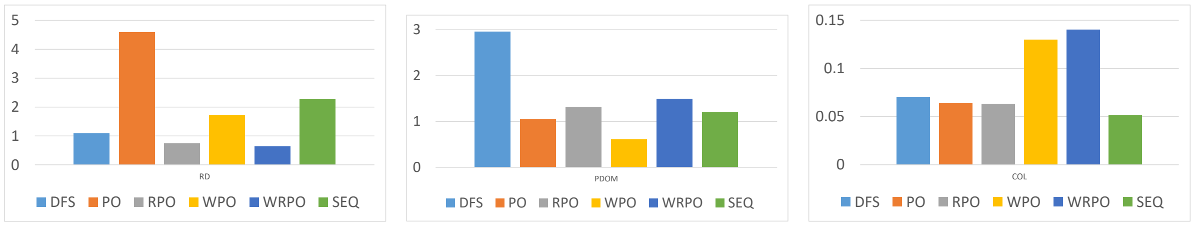
CHAPTER 1. Introduction

BigCode is a term used to represent hundreds of billions of lines of open source code hosted in services like GitHub, SourceForge, etc, and *BigCode analysis* refers to techniques that analyze source code and related artifacts at this scale. Performance of BigCode analysis over graphs like CFGs heavily depends on the order of nodes visited during the traversal: *the traversal strategy*. Graph traversal is a well-studied problem, and various standard traversal strategies exists; e.g., depth-first, post-order, reverse post-order, etc. No single traversal strategy works best for different kinds of analyses and different kinds of graphs. When an analysis is run on millions of program artifacts, selection of traversal strategies can have a great impact on the performance; however, there exists no technique to select a suitable traversal strategy to maximize performance.

This work is motivated by two fundamental questions. First, can a program analysis and input graphs' properties help automatically select a suitable traversal strategy for that analysis and graph? If so, which properties? Second, is there a principled design for expressing program analyses as traversals, so that programmers are endowed with powerful abstractions for expressing their analysis while in the meantime the runtime is provided with capabilities to compute relevant properties to make traversal strategy selection?

1.1 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



(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.

in the same order at API call sites in the client methods' code. Such an approach contains (at least) three program 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 three analyses need to be run on a large number of client methods to produce temporal specifications with high confidence.

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

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 worklist with reverse post-ordering performs better than other strategies for the reaching definition analysis while the worklist with post-ordering outperforms the others for the post-dominator analysis and the sequential order traversal works best for the collector analysis.

The performance results are somewhat expected, but require understanding the subtleties of

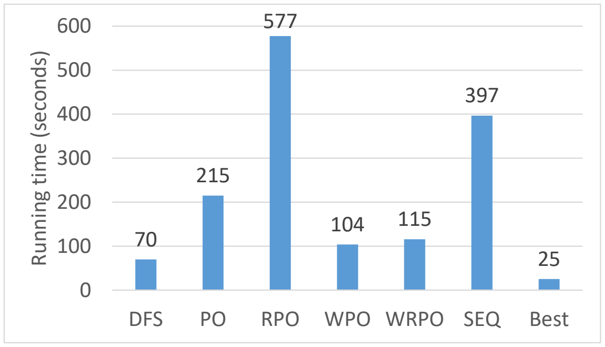


Figure 1.2: Running times of three analyses using different traversal strategies on a large codebase.

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, sequential order traversal works better than other traversal strategies for graph A. This is because for this 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 sequential 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 sequential order traversal performs better than other traversal strategies.

Observation 1 There is no single most suitable traversal strategy for different types of analyses on a graph. The selection depends on the **properties of the analysis**.

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 property of graph A of having 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.

Observation 2 The **properties of the input graphs** can be used to optimize the traversal strategy and, thus, determine the selection of the most time-efficient strategy.

The potential gains of selecting a suitable traversal strategy can be significant for BigCode analysis. 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).

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 hybrid technique using a set of 18 program 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 BigCode 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, the running times of the analyses on BigCode in our evaluation were considerably reduced by 23%–79%. 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 work makes the following contributions:

- It describes a system for expressing program analysis as traversals. The constructs and operations in the system allows different program 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 dynamic 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 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 BigCode Analysis

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

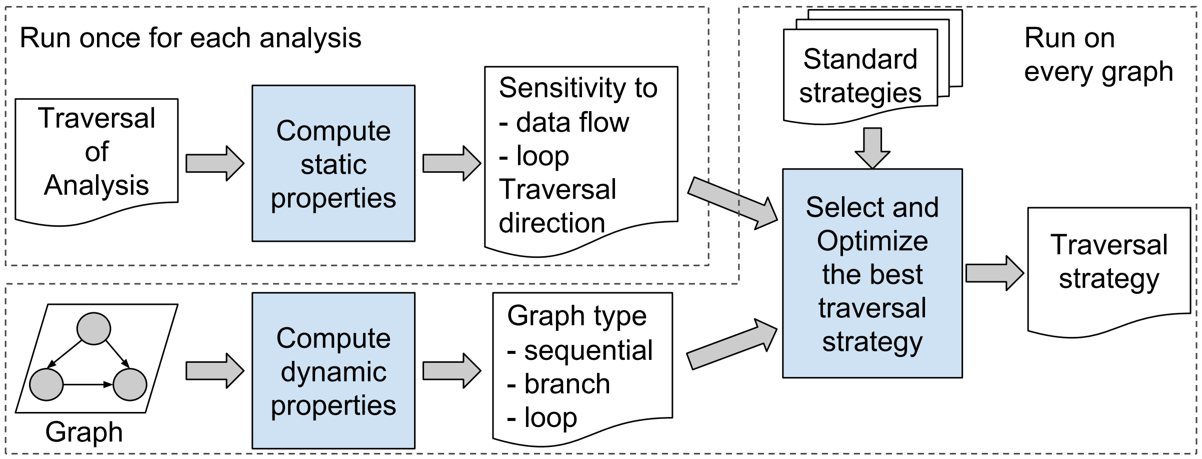


Figure 3.1: Overview of the hybrid approach for selecting and optimizing graph traversal strategy.

Figure 3.1 provides an overview of our approach and its key components. Inputs to our approach are source code for 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 dynamic property about the graph that defines the cyclicity in the graph (sequential/branch/loop). Upon computing the static and dynamic 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). 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 Program Analysis As Traversals

A program analysis such as control flow analysis, data-flow analysis, etc can be expressed as traversals over program graphs. A challenge here is to devise abstractions that lead to traversal-strategy-agnostic specification of the traversals so that alternative traversal strategies could be deployed as needed.

Definition 1 *A* ***Program Graph*** *of a program is defined as $G = (N, E)$, where G is a directed graph with a set of nodes N representing program statements, and a set of edges $E \subseteq N \times N$ representing analysis relevant relationship between the nodes in the graph. A program graph has a single start, n_{start}, and end, n_{end}, node. All the nodes in a program graph are reachable from the n_{start} and the n_{end} node is reachable from all nodes in the program graph.*

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. Note that, depending on the analysis, node ids are assigned while constructing the graph. For example, for a control flow graph, node ids may be assigned in the order of the flow of execution. Examples of program graph includes control flow graph (CFG), control dependence graph (CDG), program dependence graph (PDG), etc. Hereon, we refer to program graphs simply as graphs.

A program analysis visits nodes in the graph, in certain order, and collects information at nodes (aka, analysis facts or node outputs). For instance, the *reaching definition* analysis over a CFG, visits every node in the CFG and collects variable definitions at nodes as analysis facts. An analysis may require multiple traversals over a graph. For instance, the *reaching definition* analysis requires two traversal of CFG. In the *initialization* traversal, it collects variable definitions at nodes as analysis facts, and in the *propagation* traversal, it updates the analysis facts until a fixpoint is reached, where no further changes to analysis facts at nodes is required.

In our system, a program analysis is expressed by defining and invoking one or more traversals. A traversal is defined using a special `traversal` block as shown below:

Listing 3.1: Traversal block.

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

The traversal block shown above defines a variable `t` of traversal type and the traversal visits every node in the graph in certain order and while visiting a node `n`, it executes a block of code `tbody` to produce an output of type `OType` for the visited node.

A `traversal` block always take one parameter of type `Node` representing a graph node, and it has an optional output type, `OType`. The output type can be a primitive or a collection data type. Semantically, a traversal that defines `OType`, generates and collects analysis facts for nodes in the graph. A block of code that generates the analysis facts at a graph node is given by `tbody`. The instructions or operations in the `tbody` block is defined in the language for expressing the program analysis. A traversal block can be assigned a name, `t` in the above listing, which can be used to invoke the traversal. The type of `t` is a special traversal type.

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

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

A `traverse` expression take four parameters: `g` is the graph to be traversed, `t` is the traversal, `d` is the traversal direction, and `fp` is an optional parameter, which 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, as shown below:

Listing 3.2: Fixpoint block.

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

In the listing above, `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 using the instructions of the analysis language. 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 analysis output or fact associated with a graph node `n`, in the traversal `t`.

Data Types and Collections. Our system for expressing program analysis as traversals provides primitive and collection data types. Primitive types include: `bool, int, string` and collection types include: `Set, 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.

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

To summarize, we described a system of expressing program analysis as traversals using two special constructs: `traversal` for defining a traversal, and `traverse` for invoking a defined traversal. We allow a traversal to be invoked using several parameters, such as an analysis direction, a user-defined fixpoint function, etc. We also provide a special expression `output()` to query the analysis output of nodes and a set of operations to compose and manipulate traversal output of nodes.

An Example: Post Dominator Analysis. We now describe how to use our system to express program analysis as traversals using an example program 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 3.3. Listing 3.3 mainly defines two traversals `initT` (lines 2-4) and `domT` (lines 5-20), and invokes them using `traverse` expressions (lines 26 and 27). Lines 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 expressions used in the language to express the analysis is not described in our system, however all standard expressions are allowed. For instance, `if-else` branch expressions are used in lines 7-15, `foreach` iteration expression 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 takes an example graph, and shows the results of `initT` and `domT` traversals. Our example graph has five nodes with a branch. 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 5 is post dominated by itself, hence the output at node 5 is {5}. In Figure 3.2, under `domT` traversal, for each node visited, we show the key intermediate steps indicated by @ line number. These line numbers corresponds to the line numbers shown in Listing 3.3. We will explain the intermediate results while visiting node 1. In the `domT` traversal, at line 13, the output set `dom` is initialized using `allNodes`, hence `dom` = {0, 1, 2, 3, 4, 5}. At line 16, node 1 has two successors: {2, 3}. At line 17, the set `dom` is updated by performing an `intersection` operation using the outputs of successors 2 and 3. The output of 2 and 3 are {2, 4, 5} and {3, 4, 5} respectively. By performing the intersection of these two sets, the `dom` set becomes {4, 5}. At line 18, the id of the visited node is added to the `dom` set and it becomes {1, 4, 5}. Hence, the post dominator set for node 1 is {1, 4, 5}.

[1] Why sequential order is chosen for `initT` and post-order is chosen for `domT` is explained in §3.4.1.

3.2 Static and Dynamic Properties

While it is known in the literature that choosing a right traversal strategy for a program 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 thes right traversal strategy. These factors include: the static properties of the analysis and the dynamic 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 program analysis as traversals (§3.1).

3.2.1 Data-Flow Sensitivity

Our first property of interest is *data-flow sensitivity* that we compute for every traversal in the analysis. This property models the dependence of the traversal output of a node on the traversal outputs of other nodes in the graph.

Definition 2 $P_{DataFlow}$ *(**Data-flow sensitivity**). Let t be a traversal with body tbody. Let O be a map that collects and maintains the traversal output of nodes, where O is indexed by node id n. Let F be some function representing computation of $O[n]$ in the traversal body tbody. If $O[n]$ is computed by applying F over one or more of $O[n']$, where $n' \in n.preds$ or $n.succs$, then t is data-flow sensitive, i.e. $P_{DataFlow}$ is true otherwise false.*

For instance, consider the lines 16 and 17 of the `domT` traversal shown in Listing 3.3. Here, a variable `dom` holds the traversal output of a node n and it is computed by applying an `intersection` operation on traversal outputs of successors of n. Here, the function F is `intersection` over all successors of a node.

3.2.2 Computing Data-Flow Sensitivity

We provide an algorithm for computing the data-flow sensitivity property of a traversal. To determine the data-flow sensitivity property, the operations performed in the traversal needs to be analyzed. In our system of expressing program analysis as traversal, the traversal output of nodes can be fetched only using a special expression `output()`.

The key idea of our algorithm to determine data-flow sensitivity is as follows: Given a traversal definition, we parse the statements in the traversal body to identify calls to a special function `output(n, t)` that queries the traversal output of a node `n` in a traversal `t`. Given such a query, we want to determine if the queried node is not the node that is being visited and the traversal is same as the current traversal. This means that, traversal output of some other node (not the current node that is being visited) is fetched in the same traversal. This indicates that for computing the traversal output of the current node, the traversal output(s) of other nodes is required, and hence, the traversal is data-flow sensitive.

Algorithm 1: Algorithm to detect data-flow sensitivity

```
Input: t := traversal(n : Node): OType { tbody }
Output: true/false
1 $A \leftarrow getAliases($tbody, n$)$;
2 foreach stmt ∈ tbody do
3     if $stmt =$ output(n', t') then
4         if t' == t and n' $\notin A$ then
5             return true;
6 return false;
```

Algorithm 1 provides an algorithm for determining the data-flow sensitivity of a traversal defined using our system. A traversal is defined in our system using a `traversal` block: `t` := `traversal(n : Node) : OType` { `tbody` }. Given this traversal definition as input, our algorithm returns *true* or *false* indicating the data-flow sensitivity of the traversal. The algorithm iterates over the statements in the `tbody` to identify method calls of the form `output(n', t')`. The method call `output(n', t')` is a special instruction to fetch the traversal output of any node `n'` in the traversal `t'`. In the method call `output(n', t')`, if `t'` is same as the id of the current traversal that is being analyzed and the queried node `n'` is different from the traversal input node `n`, it means that traversal output of some other node is queried. For finding whether the queried node `n'` is not `n`, we compute an alias set of node `n` using a light-weight intra-procedural alias analysis loc () and see if `n'` is in the alias set of `n`. If there exists no call to `output()` that queries the output of other nodes in the traversal, the traversal is deemed data-flow insensitive.

3.2.3 Loop Sensitivity

Loop sensitivity is a property defined for data-flow sensitive traversals in an analysis (data-flow insensitive traversals do not have the loop sensitivity property). A traversal is said to be loop sensitive, if the data-flow in the traversal is affected by the loops in the input graph.

If a traversal is affected by the loop in the input graph, the traversal may require multiple iterations to compute the output for nodes. In these 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).*** *Let t be a traversal and O be a map that collects and maintains the traversal output of nodes, $O^i[n]$ represents the output of node n in the i^{th} iteration, where O is indexed by node id n. 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.*

As our traversal is data-flow sensitive, the traversal output of nodes in each iteration is computed using the traversal output of the neighbors (predecessors or successors), we have $O^i[n]$ $= F(O^i[n'])$ and $O^{i+1}[n] = F(O^{i+1}[n'])$, where $n' \in n.preds$ or $n.succs$. By substituting these in the `shrink` relation $O^{i+1}[n] \lll O^i[n]$, we get, $F(O^{i+1}[n']) \lll F(O^i[n'])$. This relation means that after applying the function F over the outputs of the neighbors in any two subsequent iterations i and $i+1$, the output must shrink. For this to be true, the outputs of neighbors must also shrink between any two subsequent iterations i and $i+1$, given by, $O^{i+1}[n'] \lll O^i[n']$, and

F has the property that it shrinks the output, given by, $F(O^i[n']) \lll O^i[n']$. Similarly, $O^{i+1}[n'] \ggg O^i[n']$ and $F(O^i[n']) \ggg O^i[n']$ holds for the expand case.

To summarize, for the traversal output of any node to shrink between any two subsequent iterations, the traversal outputs of neighbor (predecessors or successors) must also shrink in the subsequent iterations, and the functions applied must have the shrink property. Similarly, for the traversal output of any node to expand between any two subsequent iterations, the traversal outputs of neighbor (predecessors or successors) must also expand in the subsequent iterations, and the functions applied must have the expand property.

To give an example, consider the `domT` traversal shown in Listing 3.3. For this traversal P_{Loop} is `false` and the rationale is as follows. For this traversal, we want to check if the traversal output shrinks or expands. The `intersection` operation in line 17 always shrinks the output, hence the condition $F(O[n']) \lll O[n']$ is true, where F is an `intersection` operation. However, the second condition, $O^{i+1}[n'] \lll O^i[n']$ is `false`, because in the body of the traversal (lines 5-20), there exists an `add` operation that expands the output, hence subsequent iterations the traversal output cannot shrink.

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` always shrinks the output. We provide an algorithm to determine this property by analyzing the operations performed in the traversal body in Algorithm 2.

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 enable static inference of this property.

The key idea of our algorithm to determine loop sensitivity is as follows: When the input graph contains a loop, a traversal may visit nodes multiple times, until a fixpoint is reached (where the output of nodes do not change further). *Our key observation is that, for traversal* *output to get affected by the loops present in the input graph, output of nodes in multiple iterations either expands or shrinks.* It is possible to determine statically, whether the output of nodes in multiple iterations either shrinks or expands by investigating the operations used in the traversal body. Operations like `add`, `addAll`, `union`, etc, always expands the output, and operations like `remove`, `removeAll`, `intersection`, always shrinks the output.

Algorithm 2 provides an algorithm for determining the loop sensitivity of a traversal defined using our system. A traversal is defined in our system using a `traversal` block: `t` := `traversal(n : Node) : OType` { `tbody` }. Given this traversal definition as input, our algorithm returns *true* or *false* indicating the loop sensitivity of the traversal. 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`. 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 (lines 28-29).

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

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

In a program analysis, traversal output of nodes may depend on each other. For instance, in forward control flow analysis, output of the predecessors of a node flows to the node and it is consumed to construct the output of the node. Similarly, in the backward control flow analysis, output of the successors of a node flows to the node. Also, the output of all dependent nodes of a node (predecessors or successors) may not be available at the time of visiting the node in case of graphs with branches and loops. For this reason, a traversal may take multiple iterations to correctly compute and propagate the results. Hence, the graph cyclicity plays an important role in selecting an appropriate order of visiting the nodes (the traversal strategy), such that output of the dependent nodes are available for computing the output of a node.

3.3 Traversal Strategies - Candidates

We have picked seven traversal strategies as candidates for choosing an optimal traversal strategy given a traversal and an input graph[3]. These traversal strategies are describe below:

- **Sequential order (*SEQ*)**: This is the most simplest of all traversal strategies, which visits nodes in the order they appear in the nodes list (sequentially).

- **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

[3]This list is by no means exhaustive. We have picked the commonly used traversal strategies in program analysis and optimizations.

of the graph. For instance, while constructing a CFG, the node ids are assigned in the order of the control flow between nodes.

- **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. A worklist is initialized with nodes in the post-order. 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.
- **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 dynamic 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/dynamic check is minimized. [4]. To that end, we carefully devised a decision tree as shown in Figure 3.3 for traversal strategy selection.

[4]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.

The leaf nodes of the tree are one of the seven traversal strategies and non-leaf nodes are static/dynamic checks. The decision tree has eleven paths marked P_1 through P_{11}. Given a 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 dynamic checks are marked red. The static properties that are checked are: data-flow sensitivity ($P_{DataFlow}$), loop sensitivity (P_{Loop}), and the traversal direction. The dynamic 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 fix point computation is necessary). In such cases, visiting nodes in any order will be efficient, hence we assign sequential order (*SEQ*) 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 program analysis and a graph. The example analysis that we choose is the *Post Dominator Analysis* shown in Listing 3.3 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, meaning the graph has nodes with more than one successor or predecessor, but no cycles exists in the graph.

For selecting the best traversal strategies for `initT` and the graph with branches, we check the data-flow sensitivity property of the traversal. As `initT` is data-flow insensitive, *SEQ* traversal strategy is picked as shown by the path P_{11} in Figure 3.3 and no further checking is required. The traversal strategy *SEQ* represents sequential order that visits nodes in the order they appear in the nodes list.

For selecting the best traversal strategies for `domT` and the graph with branches, 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 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_4 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

Checking the static and dynamic properties not only determines the best traversal strategies, it also helps to perform several optimizations to the selected traversal strategies.

In the traversal-based program analysis with a fixpoint function, in addition to the analysis traversal, two additional traversals of all nodes is required: one traversal to re-compute the analysis results at graph nodes, and another traversal to perform the fixpoint check to ensure that results at nodes have stabilized. The fixpoint check traversal compares the outputs of two traversals of each nodes. The two additional traversals can be eliminated and we formulate them as two optimizations, as described below:

- *[`Opt1`] Eliminating the result re-computation traversal*: A traversal that re-computes the results at graph nodes for enabling a fixpoint traversal to compare the two results (the analysis traversal result and the re-computation result) can be eliminated, if it is known that results have stabilized and not going to change.
- *[`Opt2`] Eliminating the fixpoint check traversal*: A fixpoint check traversal that checks the results of the two traversals at graph nodes can be eliminated, if it is known that results are stabilized and not going to change.

The same static and dynamic checks that are performed to determine a traversal strategies also helps to determine if the optimizations can be performed. For instance, consider path P_1 in our decision tree shown in Figure 3.3. This path selects *INC* traversal strategy that visits nodes in the increasing order of the node ids. While selecting this strategy, we came to know that the analysis is data-flow sensitive (which means the analysis output of neighbors (predecessors or successors) is required for computing the output of a node), the graph is sequential, (which means there is only one successor or predecessor), and the analysis is a forward analysis (which means the output of the predecessor is required to compute the output of a node). Together, we know that if a traversal strategy ensures that the predecessor is visited before visiting any node, the analysis output can be computed in one traversal and no fixpoint check is necessary. The traversal strategy selected for this path is *INC* and it ensures this property. Hence, both `Opt1` and `Opt2` can be applied to the selected traversal strategy to further improve the performance. As we show in our evaluation (§4.6), upto 60% of the analysis time can be saved by performing these optimizations.

In our decision tree shown in Figure 3.3, all paths from P_1 to P_8 are eligible for both the optimizations. The paths P_9 and P_{10} uses a worklist-based traversal strategy that visits only relevant nodes (relevant nodes are the nodes whose outputs have changed from the last visit) and must perform a fixpoint check every time a node is visited, hence the optimizations cannot be performed. Finally, for path P_{11}, the optimizations are not applicable, because data-flow insensitive traversals requires only one traversal to compute the results and no fixpoint check is performed.

Listing 3.3: Post Dominator analysis: an example program 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);
```

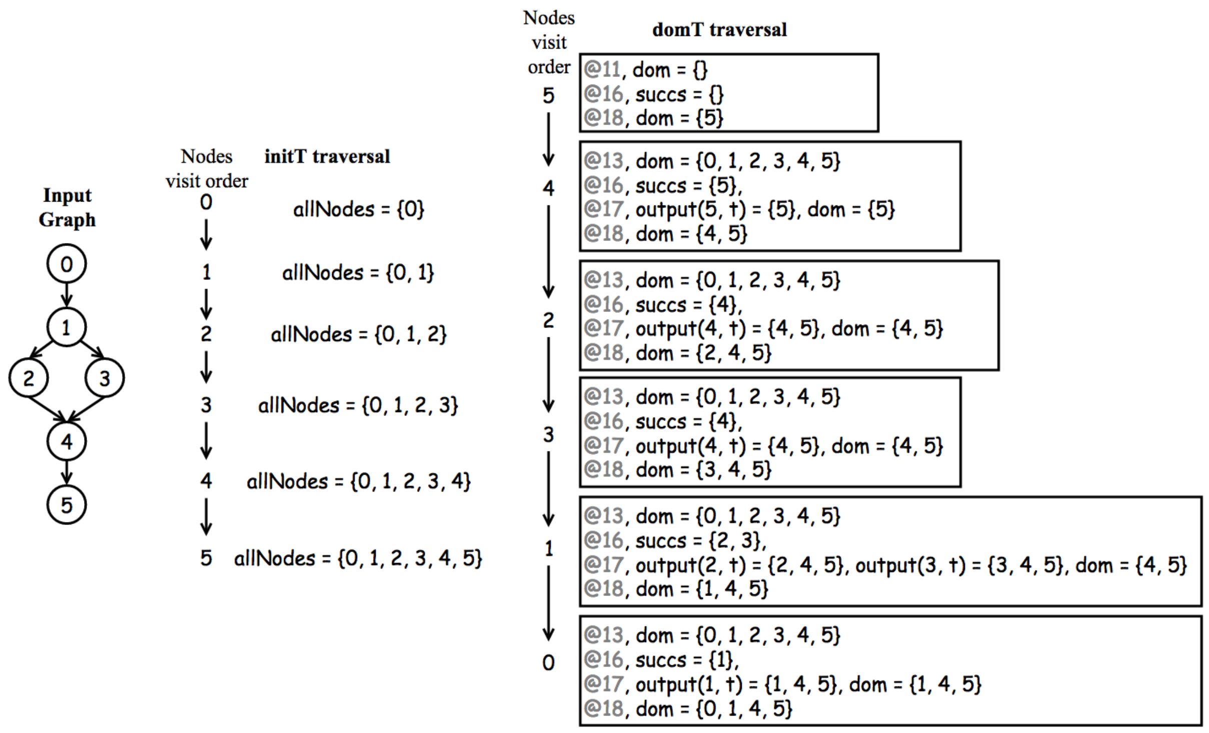

Input Graph
0
1
2
3
4
5
Nodes visit order
initT traversal
0
allNodes = {0}
1
allNodes = {0, 1}
2
allNodes = {0, 1, 2}
3
allNodes = {0, 1, 2, 3}
4
allNodes = {0, 1, 2, 3, 4}
5
allNodes = {0, 1, 2, 3, 4, 5}
Nodes visit order
domT traversal
5
@11, dom = {}
@16, succs = {}
@18, dom = {5}
4
@13, dom = {0, 1, 2, 3, 4, 5}
@16, succs = {5},
@17, output(5, t) = {5}, dom = {5}
@18, dom = {4, 5}
2
@13, dom = {0, 1, 2, 3, 4, 5}
@16, succs = {4},
@17, output(4, t) = {4, 5}, dom = {4, 5}
@18, dom = {2, 4, 5}
3
@13, dom = {0, 1, 2, 3, 4, 5}
@16, succs = {4},
@17, output(4, t) = {4, 5}, dom = {4, 5}
@18, dom = {3, 4, 5}
1
@13, dom = {0, 1, 2, 3, 4, 5}
@16, succs = {2, 3},
@17, output(2, t) = {2, 4, 5}, output(3, t) = {3, 4, 5}, dom = {4, 5}
@18, dom = {1, 4, 5}
0
@13, dom = {0, 1, 2, 3, 4, 5}
@16, succs = {1},
@17, output(1, t) = {1, 4, 5}, dom = {1, 4, 5}
@18, dom = {0, 1, 4, 5}

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

Algorithm 2: Algorithm to detect loop sensitivity

```
Input: t := traversal(n : Node): OType { tbody }
Output: true/false
1  $V \leftarrow \{\}$ // a set of output variables related to n;
2  $V' \leftarrow \{\}$ // a set of output variables not related to n;
3  expand ← false;
4  shrink ← false;
5  gen ← false;
6  kill ← false;
7  $A \leftarrow getAliases(n)$;
8  foreach stmt ∈ tbody do
9      if stmt is v = output($n'$, $t'$) then
10         if $t' == t$ then
11             if $n' \in A$ then
12                 $V \leftarrow V \cup v$;
13             else
14                 $V' \leftarrow V' \cup v$;
15 foreach stmt ∈ tbody do
16     if stmt = union($c_1$, $c_2$) then
17         if ($c_1 \in V$ and $c_2 \in V'$) || ($c_1 \in V'$ and $c_2 \in V$) then
18             expand ← true;
19     if stmt = intersection($c_1$, $c_2$) then
20         if ($c_1 \in V$ and $c_2 \in V'$) || ($c_1 \in V'$ and $c_2 \in V$) then
21             shrink ← true;
22     if stmt = add($c_1$, e) || addAll($c_1$, $c_2$) then
23         if $c_1 \in V$ then
24             gen ← true;
25     if stmt = remove($c_1$, e) || removeAll($c_1$, $c_2$) then
26         if $c_1 \in V$ then
27             kill ← true;
28 if (expand and gen) || (shrink and kill) then
29     return true;
30 else
31     return false;
```

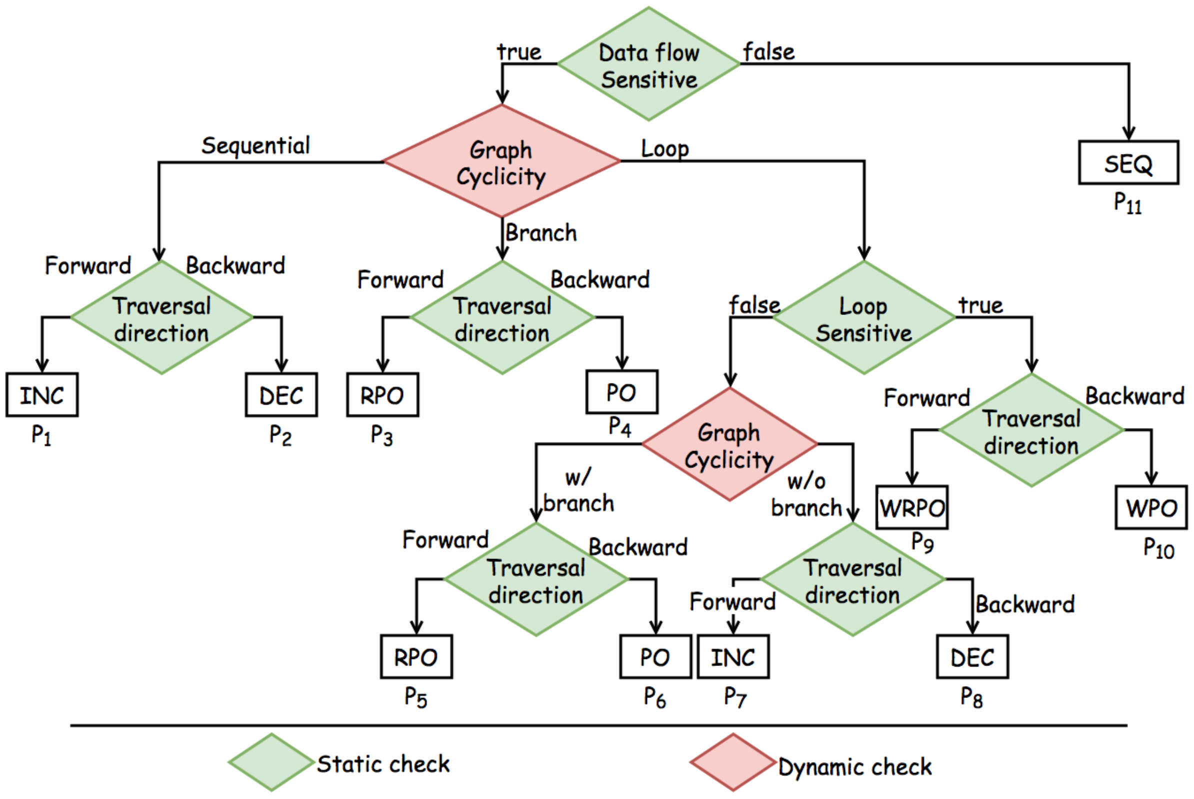


Figure 3.3: Traversal strategy selection decision tree. P_0 to P_{11} are markings for paths.

CHAPTER 4. Evaluation

We conducted an empirical evaluation on a set of 18 basic program analyses and 2 public massive code datasets to answer the following research questions about our hybrid approach for selecting and optimizing traversal strategies.

RQ1. *How much reduction in running time can be achieved using our hybrid approach?* The answer to this shows the benefit of using our optimized selected traversal over standard ones.

RQ2. *Are the analysis result correct using the hybrid approach?* This will show that the decision analyses and optimizations in our approach does not affect the correctness of the program analyses' results.

RQ3. *How often does the hybrid approach select the most time-efficient traversal?*

RQ4. *How do the different components in the approach and different kinds of static and dynamic properties impact the overall performance?* This is done by various insight analysis of our evaluation results.

4.1 Analyses, Datasets and Experiment Setting

Analyses. We collected program analyses that traverse control flow graphs from textbooks and program 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 18 program analyses as shown in Table 4.1. They include 10 basic ones (analyses 1, 2, 6, 7, 8, 9, 10, 12, 13 and 16) from textbooks Aho et al. (2006); Nielson et al. (2010) and 8 others for detecting program bugs, and code smells from the Soot framework Vallée-Rai et al. (1999) (analyses 3, 4, 5, 11 and 15), and FindBugs tool Ayewah et al. (2007) (analyses 14, 17 and 18). Table 4.1 also shows the number of traversals each analysis contains and their static

Table 4.1: List of program analyses and the properties of their involved traversals. Ts: total number of traversals. t_i: properties of traversal i-th. *Flw*: data-flow sensitive. *Lp*: loop sensitive. *Dir*: traversal direction where —, → and ← mean iterative, forward and backward, respectively.

	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	Available expression (AE)	2	✗	✗	—	✓	✓	→			
7	Dominator (DOM)	2	✗	✗	—	✓	✗	→			
8	Local may alias (LMA)	2	✗	✗	—	✓	✓	→			
9	Local must not alias (LMNA)	2	✗	✗	—	✓	✓	→			
10	Live variable (LV)	2	✗	✗	—	✓	✓	←			
11	Nullness analysis (NA)	2	✗	✗	—	✓	✓	→			
12	Post dominator (PDOM)	2	✗	✗	—	✓	✗	←			
13	Reaching definition (RD)	2	✗	✗	—	✓	✓	→			
14	Resource status (RS)	2	✗	✗	—	✓	✓	→			
15	Very busy expression (VBE)	2	✗	✗	—	✓	✓	←			
16	Used and defined variable (UDV)	1	✗	✗	—						
17	Useless increment in return (UIR)	1	✗	✗	—						
18	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%)

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 eighteen of these analysis using constructs described in §3.1.

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

Setting. We compared our *hybrid* approach against the six standard traversal strategies in §3.3: DFS, PO, RPO, WPO, WRPO and SEQ. 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 dynamic 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 of the analyses using our hybrid approach and then study the achieved reductions against standard traversal strategies.

4.2.0.1 Running Time

Table 4.3 shows the running times for 18 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. Columns **Static** and **Dynamic** 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 dynamic time, as it is performed only once per traversal. When compared to the average dynamic time, the static time is quite significant. However, the overhead introduced by static information collection

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

Analysis	Avg. Time		Static	Dynamic			
	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	0.006	1444K
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
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

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 BigCode analysis.

4.2.0.2 Time Reduction

To evaluate the efficiency in running time of the hybrid approach over other traversal strategies, we ran the 18 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 –.

Analysis	DaCapo						GitHub					
	DFS	PO	RPO	WPO	WRPO	SEQ	DFS	PO	RPO	WPO	WRPO	SEQ
CP	12%	76%	5%	55%	7%	62%	12%	86%	7%	77%	2%	81%
CSD	29%	90%	27%	63%	2%	82%	23%	–	17%	–	8%	–
DC	29%	19%	82%	3%	50%	72%	18%	15%	–	3%	–	–
LIC	11%	76%	5%	56%	3%	63%	13%	87%	12%	78%	13%	84%
USA	23%	87%	22%	59%	4%	78%	16%	–	12%	–	4%	–
AE	27%	8%	26%	60%	8%	79%	12%	–	11%	–	6%	–
DOM	43%	96%	37%	60%	3%	93%	23%	–	22%	–	4%	–
LMA	23%	32%	18%	60%	2%	31%	15%	–	8%	–	4%	–
LMNA	27%	37%	20%	65%	6%	40%	15%	–	9%	–	4%	–
LV	23%	18%	73%	5%	40%	60%	20%	16%	63%	5%	66%	70%
NA	14%	78%	17%	32%	4%	65%	8%	82%	8%	68%	4%	79%
PDOM	38%	29%	92%	5%	62%	85%	17%	14%	–	17%	–	–
RD	13%	78%	6%	58%	7%	65%	13%	88%	7%	77%	3%	82%
RS	27%	27%	26%	27%	23%	26%	10%	39%	6%	25%	3%	45%
VBE	25%	22%	79%	7%	61%	68%	23%	21%	–	6%	–	–
UDV	5%	3%	5%	6%	5%	3%	1%	2%	1%	4%	3%	-1%
UIR	2%	2%	1%	0%	0%	0%	-1%	3%	2%	4%	4%	-2%
WNIL	2%	3%	4%	3%	4%	-1%	-2%	4%	3%	3%	3%	-2%
Overall	23%	79%	60%	47%	25%	74%	–	–	–	–	–	–

(a) Time reduction for each analysis.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	SEQ
Data-flow	24%	80%	62%	49%	25%	76%
¬Data-flow	3%	3%	3%	4%	4%	1%

(b) Overall reduction over analysis properties.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	SEQ
Sequential	16%	71%	55%	49%	20%	66%
Branch	25%	77%	56%	51%	29%	85%
Loop	45%	83%	64%	54%	26%	88%

(c) Overall reduction over graph properties.

Figure 4.1: Reduction in running times. Background colors indicate the ranges of values: no reduction, (0%, 10%), [10%, 50%) and [50%, 100%].

The result in Figure 4.1a shows that the hybrid approach helps reduce the running times in almost all cases. 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 sequential 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 DFS, hybrid approach reduces the overall execution time by about 54 minutes (from 447 to 393 minutes) on GitHub dataset and about 4 minutes (from 16 to 12 minutes) on DaCapo 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,

Table 4.4: Traversal strategy prediction precision.

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

the reduction rates are high ranging from 24% to 80%. 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) and, in some cases, even no reductions at all (light red cells). We actually perform slightly worse than sequential order traversal strategy for two analyses in this category. This is because sequential order traversal strategy is the best strategy for all the CFGs in these analyses. Hybrid approach also chooses sequential order traversal strategy but we have the overhead of static and dynamic information collection phase.

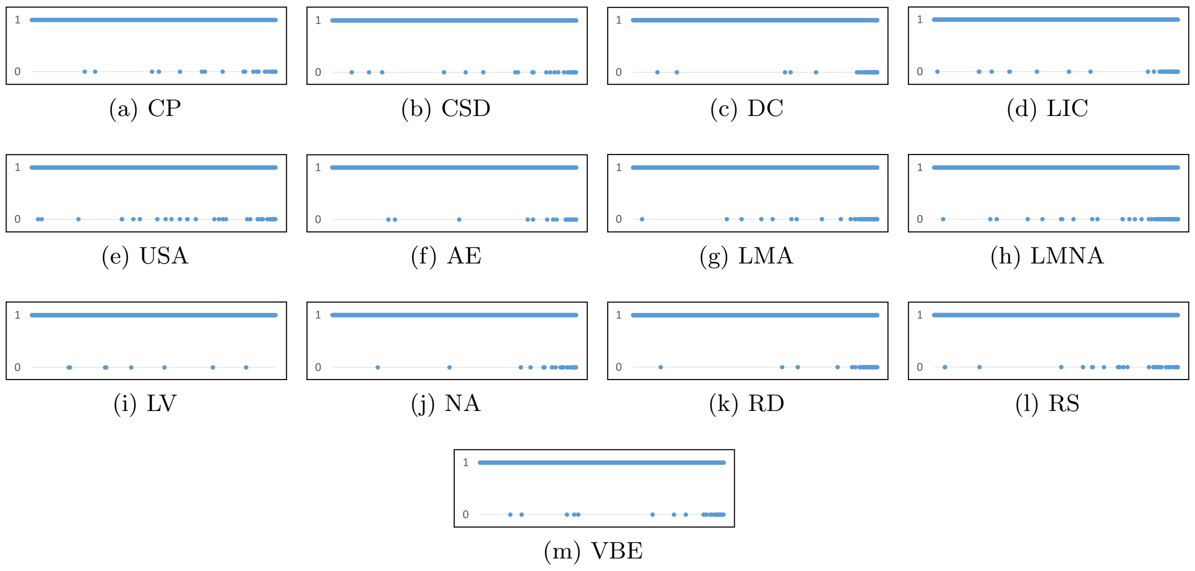
Figure 4.1c shows time reduction for different cyclicity types of input graphs, i.e., sequential, with branches and no loops, and with loops. We can see that reductions over graphs with loops is highest and those over sequential graphs is lowest.

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

In this experiment, we evaluated how well the hybrid approach picks the most time-efficient strategy. We ran the 18 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



(a) CP (b) CSD (c) DC (d) LIC (e) USA (f) AE (g) LMA (h) LMNA (i) LV (j) NA (k) RD (l) RS (m) VBE

Figure 4.2: Scatter charts for analyses that have loop sensitive traversals. On y-axis, 1 indicates a correct traversal strategy prediction and 0 indicates a mis-prediction.

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 4.2 shows scatter charts for the traversal selection results for 13 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.

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 4.3 show the result for 18 analyses. 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 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 18 analyses. Figure 4.4 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.

4.7 Threats to validity

Our first threat to validity is our selection of program analysis used in our evaluation. While there exists no source for a standard set of analysis, we relied mainly on text books and program 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 and backward analysis, data-flow sensitive and insensitive analysis, loop sensitive and insensitive analysis. As part of future work, we plan to extend the scope to inter-procedural analyses, which should enable more sophisticated choices like pointer and alias analysis.

Our next threat to validity is our selection of BigCode 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.

	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%
USA	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%
LIC	32%	0%	13%	0%	0%	0%	0%	0%	5%	0%	50%
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%
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	31.04%	11.29%	12.13%	4.41%	0.31%	0.31%	0.10%	0.10%	4.22%	1.26%	34.78%

(a) DaCapo

	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%
USA	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%
USA	35%	0%	10%	0%	0%	0%	0%	0%	5%	0%	50%
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%
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	33.03%	12.01%	9.80%	3.58%	0.31%	0.31%	0.13%	0.13%	4.47%	1.34%	34.78%

(b) GitHub

Figure 4.3: Distribution of decisions over the paths of the decision tree. Background colors indicate the ranges of values: 0%, (0%, 1%), [1%, 10%) and [10%, 100%].

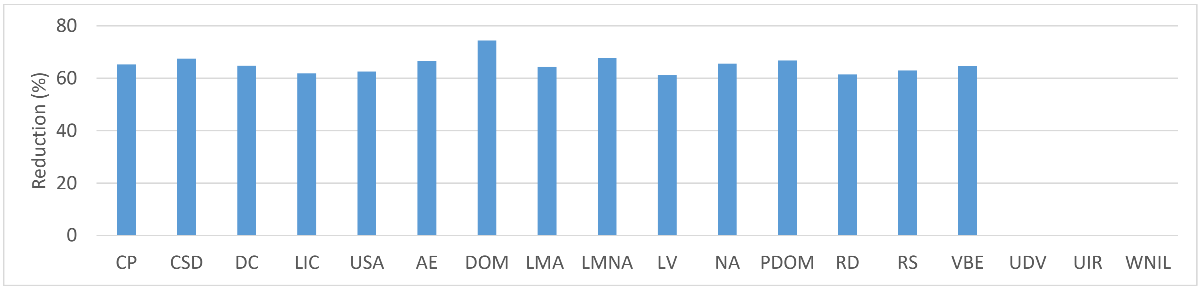


Figure 4.4: Reduction in execution time of the hybrid approach due to traversal optimization.

CHAPTER 5. Related Works

Our work on the hybrid technique to select and optimize traversal strategies for program analysis is related to the works that optimize program analysis.

Atkinson and Griswold Atkinson and Griswold (2001) discuss several implementation techniques for improving the efficiency of data-flow analysis, namely: factoring data-flow sets, visitation order of the statements, selective reclamation of the data-flow sets. They discuss two commonly used traversal strategies: iterative search and worklist, and propose a new algorithm that results in 20% fewer node visits. In their algorithm, a node is processed only if the data-flow information of any of its successors (or predecessors) has changed. Tok *et al.* Tok et al. (2006) proposed a new worklist algorithm for accelerating inter-procedural flow-sensitive data-flow analysis. They generate inter-procedural def-use chains on-the-fly to be used in their worklist algorithm to re-analyze only parts that are affected by the changes in the flow values. Hind and Pioli Hind and Pioli (1998) proposed an optimized priority-based worklist algorithm for pointer alias analysis, in which the nodes awaiting processing are placed on a worklist prioritized by the topological order of the CFG, such that nodes higher in the CFG are processed before nodes lower in the CFG. Kildall Kildall (1973) proposes combining several optimizing functions with flow analysis algorithms for solving global code optimization problems. For some classes of data-flow analysis problems, there exist techniques for efficient analysis. For example, demand interprocedural data-flow analysis Horwitz et al. (1995) can produce precise results in polynomial time for inter-procedural, finite, distributive, subset problems (IFDS), constant propagation Wegman and Zadeck (1991), etc. Compared to all these approaches that propose optimized traversal algorithms, our work proposes a technique to select the best traversal strategy from a list of candidate traversal strategies and optimize it, based on the static and dynamic properties of the analysis and graph.

Cobleigh *et al.* Cobleigh et al. (2001) study the effect of worklist algorithms in model checking. They identified four dimensions along which a worklist algorithm can be varied. Based on four dimensions, they evaluate 9 variations of worklist algorithm. They do not solve traversal strategy selection problem. Moreover, the properties that they consider do not take analysis into account, they are mainly variations of the worklist algorithm. For instance, using stack or queue to implement worklist (stack gives depth-first and queue gives breadth-first order of nodes). We consider both static properties of the analysis, such as data-flow sensitivity and loop sensitivity, and the cyclicity of the graph. Further, we also consider non-worklist based algorithms, such as post-order, reverse post-order, control flow order, sequential order, etc., as candidate strategies for selection.

Several infrastructures exist today for performing BigCode analysis Dyer et al. (2013); Bajracharya et al. (2014); Gousios (2013). Boa Dyer et al. (2013) is a language and infrastructure for analyzing open source projects from GitHub, SourceForge, etc. Sourcerer Bajracharya et al. (2014) is an infrastructure for large-scale collection and analysis of open source code. GHTorrent Gousios (2013) is a dataset and tool suite for analyzing GitHub projects. These frameworks currently support structural or abstract syntax tree (AST) level analysis and a parallel framework such as map-reduce is used to improve the performance of BigCode analysis. We believe, when these frameworks support graph level analysis, such as control-flow and data-flow analysis, our technique of selecting the best traversal strategy can improve the performance beyond parallelization.

There have been many works that targeted graph traversal optimization through various ways. Green-Marl Hong et al. (2012) is a domain specific language for expressing graph analysis. Green-Marl uses the domain specific knowledge in applying optimizations. It uses the high-level algorithmic description of the graph analysis written in Green-Marl to exploiting the exposed data level parallelism. In a way, Green-Marl' optimization is similar to ours, where both the approaches utilize the properties of the analysis description, however our technique also utilizes the properties of the graphs in selecting the best traversal strategy. Moreover, Green-Marl' optimization is through parallelism, ours is by selecting the suitable traversal strategy. Pregel Malewicz et al. (2010) is a map-reduce like framework that aims to bring distributed processing to graph algorithms. While Pregel' performance gain is through parallelism, our approach achieves performance gain by traversing the graph efficiently. 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 6. Conclusion

Improving the performance of program analyses that runs on massive code bases (aka BigCode) is an ongoing challenge. One way to improve the performance of program analysis expressed as traversals over graphs like CFGs, is by picking the right traversal strategy that defines the order of nodes visited. The selection of the best traversal strategy depends both on the functionality of the analysis and the properties of the graph on which the analysis is run. We proposed a hybrid technique for selecting and optimizing graph traversal strategies for program analysis expressed as traversals over graphs. Our solution includes a system for expressing program 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, the running times of the analyses on BigCode were considerably reduced by 23%–79%. The overhead imposed by collecting additional information for our hybrid 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

Soot Local May Alias Analysis. https://github.com/Sable/soot.

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.

Bajracharya, S., Ossher, J., and Lopes, C. (2014). Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259.

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.

Cobleigh, J. M., Clarke, L. A., and Osterweil, L. J. (2001). The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 37–46, Washington, DC, USA. IEEE Computer Society.

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.

Gousios, G. (2013). The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press.

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.

Hind, M. and Pioli, A. (1998). Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, SAS '98, pages 57–81, London, UK, UK. Springer-Verlag.

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.

Horwitz, S., Reps, T., and Sagiv, M. (1995). Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 104–115, 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.

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.

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.

Tok, T. B., Guyer, S. Z., and Lin, C. (2006). Efficient Flow-sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 17–31, Berlin, Heidelberg. Springer-Verlag.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press.

Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210.

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.