Dickinson College
Dickinson Scholar

Honors Theses By Year | Honors Theses

5-18-2014

AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem

Samuel Livingston Kelly
Dickinson College

Follow this and additional works at: http://scholar.dickinson.edu/student_honors

Part of the Software Engineering Commons

Recommended Citation
Kelly, Samuel Livingston, "AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem" (2014). *Dickinson College Honors Theses.* Paper 147.

This Honors Thesis is brought to you for free and open access by Dickinson Scholar. It has been accepted for inclusion by an authorized administrator. For more information, please contact scholar@dickinson.edu.

AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem

by

Sam Kelly

Submitted in partial fulfillment of Honors Requirements
for the Computer Science Major
Dickinson College, 2014

Associate Professor Tim Wahls, Supervisor
Associate Professor John MacCormick, Reader
Part-Time Instructor Rebecca Wells, Reader

April 22, 2014

The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by Sam Kelly, and awards departmental honors in Computer Science.

Tim Wahls (Advisor)	Date
John MacCormick (Committee Member)	Date
Rebecca Wells (Committee Member)	Date
Richard Forrester (Department Chair)	Date

Department of Mathematics and Computer Science
Dickinson College

April 2014

Abstract

AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem

by
Sam Kelly

In this paper we present two novel abstract syntax tree (AST) indexing algorithms that solve the get-descendants-by-type problem in near constant time. This work has been implemented in the U.S. Department of Energy's ROSE compiler framework with plans to be officially integrated as an optimization library called "NodeFinder". ROSE is an open source software analysis platform and source-to-source compiler suited for large-scale C/C++, UPC, Java, Python, Fortran, OpenCL, CUDA, and OpenMP applications that has been actively developed at Lawrence Livermore National Laboratory for the last sixteen years. The get-descendants-by-type problem is the problem of efficiently answering queries of the form "given an arbitrary AST node A and an arbitrary node type T, return all descendants of A that are of type T". Our algorithms are generic in that they can also be applied to any tree that has a meaningful notion of node "type", so we also explore some potential applications in the fields of file systems and databases.

Acknowledgments

This research is a partial continuation of work performed by Sam Kelly as a U.S. Department of Homeland Security HS-STEM summer research intern working with the ROSE compiler team at Lawrence Livermore National Laboratory (LLNL) in 2013 under contract DE-AC52-07NA27344. While the AST indexing algorithms were developed entirely independently of LLNL, some of the associated presentation materials were derived from material developed at LLNL now cleared for public release under LLNL-POST-641521. Special thanks go to the members of the ROSE development team, especially Justin Too, Markus Schordan, and Dan Quinlan, as well as to Professor Tim Wahls and the other members of our Honors Thesis committee at Dickinson College.

Table of Contents

Title Page………………………………………………………………….. i
Signature Page……………………………………………………………... ii
Abstract……………………………………………………………………..iii
Acknowledgments………………………………………………………… iv
Table of Contents…………………………………………………………...v

Introduction..1

1.1 Overview..1

1.2 Background...2

1.3 AST Traversals for Static Analysis...3

Related & Prior Work 7

2.1 AST Indexing & Searching...7

2.2 Grammar-Based AST Traversal Optimization 9

Two Novel AST Indexing Algorithms 11

3.1 A Naive Approach 11

3.2 AST Indexing Overview 13

3.3 Algorithm A: Overview 15

3.4 Algorithm A Implementation Details 17

3.5 Algorithm B Overview 19

3.6 Algorithm B Implementation Details..20

3.7 Algorithm B Memory & Time Complexity 23

3.8 Discussion & Theoretical Tradeoffs 25

Benchmarks & Results 26

4.1 Benchmark Methodology..26

4.2. Static Sized AST Results 29

4.3 Dynamically Sized AST Results...31

4.4 Discussion 35

Future Work..36

5.1 Dealing with DAGs...36

5.2 Performance Comparison with the ROSE AstQuery Library.............37

5.3 Parallelizing NodeFinder 37

Appendix A: Raw Dynamic Benchmark Results 39

References 41

Chapter 1

Introduction

1.1 Overview

In this paper we present two novel algorithms for indexing abstract syntax trees (ASTs) for the purpose of solving the get-descendants-by-type problem in near constant time within the ROSE compiler framework. ROSE is an open source software analysis platform and source-to-source compiler suited for large-scale C/C++, UPC, Java, Python, Fortran, OpenCL, CUDA, and OpenMP applications that has been actively developed at Lawrence Livermore National Laboratory for the last sixteen years[1]. We define the get-descendants-by-type problem as the problem of efficiently answering queries of the form "given an arbitrary AST node A and an arbitrary node type T, return all descendants of A that are of type T".

We first cover some of the motivating factors behind efficiently solving the get-descendants-by-type problem as well as some related and prior work. This includes a discussion of an alternate grammar-based optimization we considered that would have allowed us instead to make branch-cuts on standard AST traversals. We then cover the details of our AST indexing algorithms and derive their time and memory complexities. Next, we present implementation details and analyze benchmarks for our algorithms, which we implemented in a ROSE subproject called "NodeFinder". We also discuss some inherent tradeoffs that exist between the two algorithms in terms of indexing speed vs. query speed. Finally, we propose some future research that could be conducted on these algorithms, and outline a method by

which the indexing and querying process for both algorithms could be parallelized. As we show in our paper, the get-descendants-by-type problem is ubiquitous in the field of static analysis (either explicitly as with Markus Schordan's AST matching algorithm[2], or implicitly as part of a more standard visitor pattern traversal[1]). While we approach this problem from the perspective of indexing abstract syntax trees, our algorithms are applicable to any tree with a meaningful notion of node type, so we also discuss some potential applications of our algorithms outside the field of compilers. These potential applications include indexing file systems and databases for the purpose of searching these structures for particular classes of records or for particular file types.

1.2 Background

Abstract syntax trees (ASTs) are tree representations of the syntactic structure of computer program source code. Nodes in an AST correspond with individual programming constructs[3]. For example, in ROSE "SgFunctionDefinition" nodes correspond with function definitions, "SgIfStmt" nodes correspond with if-statements, and "SgVarRefExp" nodes correspond with variable references[4]. Variable references contained within the body of a function would thus be descendants of an SgFunctionDefinition node in a ROSE AST. In most compilers, ASTs are used as an intermediary representation – compilers will typically parse the various source files that make up a program and then generate an AST[3]. In the case of ROSE, once an AST has been generated researchers can then perform static analysis on this AST or initiate a translation routine that translates the program into one of the other supported

[1] For example, if one were to write a visitor pattern traversal that for each function in a source program generates a list of variable references that appear in that function, then this traversal would be *implicitly* solving the get-descendants-by-type problem(s) of the form: "get all function descendants of the root node; for each of these, get all variable reference descendants."

ROSE source languages[4].

A depth-first traversal is a common tree traversal that starts at the root node and explores all the way to the end of each branch before backtracking (as opposed to a breadth-first traversal, which explores all nodes at the current depth before moving on to the next level). The depth-first index of a node in a tree is simply the visitation index that node would have during a depth-first traversal[5]. Our AST indexing algorithms depend on properties we derive from the depth-first index of nodes in an AST.

1.3 AST Traversals for Static Analysis

Traditionally, most types of static analysis rely on visitor-pattern-style traversals, where programmers must write separate methods to handle each type of node involved in the traversal[3]. Visitor pattern code can be awkward to write, and ROSE has thousands of different node types[1], making this process even more difficult and error prone for implementers than in most typical environments. Another major downside of using visitor pattern traversals, particularly for static analysis, is the fact that visitor pattern traversals in the best case can only ignore nodes on a per-type basis. This means that even if a traversal is heavily optimized, the majority of the AST usually still has to be explored, even if only a small subset of nodes or node types are the target of the traversal. Without support from any additional analysis or data structures, a visitor cannot magically know that a particular branch does not contain any nodes that it is looking for, so oftentimes entire regions of ASTs are explored that end up being irrelevant to the overall search. This isn't a problem if the AST is only traversed once, however in practice programmers will often traverse the same AST multiple times in a row using different visitors, leading to compounded inefficiencies that

could be avoided with a more robust, index-based system.

Markus Schordan has written an "AST matching" algorithm for ROSE[2], which provides a regular-expression-like interface for making direct queries against arbitrary ROSE ASTs. This interface allows programmers to avoid having to write complex visitor pattern traversals, and allows them to specify advanced queries involving different node types and an impressive set of logical search operators. For example, a programmer could use the query "$v=SgVarRefExp | #SgFunctionDefinition" to get the list of all variable references that are not contained within function definitions. While the AST matching algorithm is extremely powerful and expressive from a programmer's perspective, it really just abstracts away from the programmer the complexity of writing visitor pattern traversals – internally, the algorithm analyzes match expressions and actually performs visitor pattern traversals based on these match expressions. Since the visitor pattern traversal code is abstracted away from the programmer, it is currently not possible for users to optimize these traversals so that, for example, unnecessary or irrelevant branches are not explored. Furthermore if multiple consecutive queries are performed on the same AST it is quite possible that the AST or sub-trees of the AST will be explored again and again needlessly.

This is a particularly relevant issue with Compass[4], which is a powerful extension for ROSE designed to automatically detect flaws and security vulnerabilities in existing software source codes. Specific flaws are detected by "checkers", which are defined on a per-flaw basis using Compass's plugin-based architecture. Currently each Compass checker is implemented independently either using the AST matching algorithm, or using a visitor pattern traversal designed to perform only the analysis specific to that checker. Many of the checkers within Compass will first use a match expression to search for all function *declarations*, another match

expression to search for all function *definitions*, and then will use an additional match expression for each function definition found in the previous search to search for all variable references that can be found as descendants of that function definition. Thus repeated global[2] searches and nested sub-searches (searching for nodes contained within already found nodes) are not only possible but seem to be common in practice[2]. Unfortunately, even when users take specific actions to avoid these situations (i.e. chaining all traversals of the same AST together into one incredibly complicated match expression), it is impossible to avoid repeatedly traversing the same parts of the AST numerous times in the case of nested traversals without some further optimization, because nested traversals depend on the results of previous traversals, and so must be performed sequentially. Furthermore, combining multiple queries together can result in some incredibly awkward and inefficient casting situations, such as having to iterate over a match result where the current node could be one of *many* possible node types.

Arguably, the main convenience provided by the AST matching algorithm is that it allows programmers to search the descendants of a specified node for all nodes of one or more specified types without using a visitor – that is, it allows programmers to solve the get-descendants-by-type problem using an elegant, imperative interface. That said, the performance costs of using AST matching can be significant, especially in the case of nested queries. Likewise it would be extremely difficult to write a single AST traversal that encapsulates all of the analysis performed by Compass's 70+ checkers.

To alleviate these issues, we have devised two AST indexing algorithms. Once an AST has been indexed using one of our algorithms, if a programmer specifies a node *A* in the AST,

---

[2] i.e. searches that begin at the root node of the AST

and a particular node type T to search for, all descendants of A that are of type T can be found in either average case constant time or logarithmic time (in terms of the total number of nodes of type T contained in the AST, rather than in terms of the total number of nodes in the AST) depending on which indexing algorithm is used.

As we discuss extensively in Chapter 3, we have successfully implemented our two AST indexing algorithms in a ROSE sub-project called NodeFinder. While NodeFinder in a very basic sense could be construed as a more efficient, drop-in replacement for some of the functionality provided by the AST matching algorithm, it is worth noting that NodeFinder only supports the ability to find descendants directly by type, whereas the AST matching algorithm is capable of performing complex queries with numerous conditionals and search operators. Thus the AST matching algorithm and NodeFinder really accomplish partially divergent goals. The AST matching algorithm provides as many advanced features and operators as possible while offering a predictable, roughly linear[3] runtime for each query. NodeFinder, on the other hand, offers some basic but very commonly used functionality in an extremely optimized fashion specifically designed for static analysis involving deeply nested or repeated searches on the same AST. While most of the advanced features of the AST matching algorithm are incompatible with NodeFinder, it is also worth noting that the AST matching algorithm could be optimized to make use of NodeFinder for basic queries that don't use any of its advanced features (we are actually in the process of working with Markus Schordan to determine whether the AST matching algorithm could be automatically optimized on certain queries using NodeFinder). That way, basic match expressions would be more efficient, but users would still have access to all the advanced features of the AST

[3] Linear in the number of descendants of the node being searched

matching algorithm at the cost of an increased runtime when these features are used.

Chapter 2

Related & Prior Work

2.1 AST Indexing & Searching

With the exception of the AST matching algorithm (about which Schordan has not yet published any papers) and our own AST indexing algorithms, we have not yet found any cases in the existing literature on static analysis where researchers have tried to create any sort of imperative, query based alternative to visitor-pattern-style AST traversals. Likewise we have found no existing cases where researchers have tried to "index" an AST in a way that allows for subsequent constant time (or even logarithmic time) queries for descendants of specific node types, as we are able to do with our algorithms (though in Section 3.1 we describe an obvious naive approach to this that ends up wasting an impractical amount of memory and whose index takes too long to generate). Thus at the time of writing we are confident that our algorithms represent a unique, novel addition to the field of compilers and also to the broader field of algorithms that operate on trees.

As mentioned before, our AST indexing algorithms are not limited just to ASTs, and can be applied to all trees in general so long as said trees have a meaningful notion of different node "types". In general, the existing literature on trees with different node types is rather sparse, though a tenuous analogue exists between our algorithms and some of the algorithms commonly used in file systems and databases.

Since the early 1980s, B-trees and variants such as B+-trees have commonly been used

to facilitate rapid searching of databases[6] and file systems[7]. Like binary search trees, B-trees can provide logarithmic time access to files in a file system or records in a database based on a search query, but unlike binary search trees, they are much easier to balance (though they tend to waste more memory)[7]. This situation is quite different from that faced by our AST indexing algorithms, however, since our algorithms only provide access to subsets of nodes arranged by type, whereas B-trees need to facilitate much more general string-based searches. Additionally, B-trees need to account for the fact that nodes in a file system or database are constantly being added, changed, or removed, whereas in our situation we can assume that the ASTs in question will remain static after indexing.

Notably, though, file systems *do* contain a meaningful notion of node type in the form of a file type (e.g. *.txt, *.jpg, *.pdf, etc.), and a frequent search operation in modern operating system is to search for all files of a particular type in some given directory or sub-directory. A NodeFinder-style index would be able to provide a much faster speed-up for these types of searches than would a standard B- or B+-tree, (specifically, Algorithm B would be able to carry out these types of searches in constant average case time) however this approach would be impractical in most cases because any changes to the tree in question would require full re-indexing. Thus while there are some analogues between our algorithms and B-trees in terms of the purposes they serve, and while there might even be some applications of our algorithms in the fields of databases and file systems, these applications would likely be rare due to the rarity[8] of predominantly read-only databases and file systems. On the other hand, future research might uncover more efficient ways of updating NodeFinder-style indexes without requiring full re-indexing as frequently.

2.2 Grammar-Based AST Traversal Optimization

Since one of the primary aims of this project is to optimize the ease and speed with which we can search portions of an AST for specific types of nodes, we originally considered a grammar-based optimization that would have pruned many unnecessary branches from an otherwise normal visitor pattern traversal. We abandoned this approach once we devised our AST indexing based approach because the latter is theoretically more efficient, and the former would have been rather awkward to use and would have had some of the same pitfalls as standard visitor pattern traversals. Namely, a grammar-based approach would not solve the problem faced by Compass developers of wanting to perform only a single AST traversal while running numerous checkers that each performs some independent static analysis without having to awkwardly merge individual checkers together.

Under the grammar-based approach, we would perform a reachability analysis at compile time on all node type pairs (i.e. an analysis that determines whether node type A is capable of containing node type B as a descendant) within the context free grammar being used by our target ASTs (in our case this would simply be the context free grammar for ROSE). This information would be stored in a generated function (consisting of a two-level nested switch statement) that could determine in constant time whether it is possible for an arbitrary node type to contain another arbitrary node type as a descendant according to the ROSE grammar. At runtime, the programmer would provide a list of the node types that his/her visitor will be visiting prior to performing the corresponding traversal. During the traversal, instead of blindly exploring all node types that the programmer hasn't written explicit code for, we would filter visits to inexplicitly handled node types by making calls to our constant time function to determine whether or not at least one of the node types the

programmer's visitor is searching for is reachable from the current node type. Thus this method would only visit node types that are potentially relevant to the current traversal, in that nodes that are grammatically incapable of containing node types that interest our visitor are never explored. In short, this method would have involved automatically pruning branches from AST visitor pattern traversals based on the context free grammar for the AST's underlying language.

While we haven't been able to find any directly related work for our AST indexing algorithms, we *were* able to find some material relevant to this grammar-based approach, and to the more general topic of optimizing standard AST traversals.

Lepper et al. tackle the more general problem of automatically pruning and optimizing visitor pattern traversals that operate on arbitrary data structures, resulting in a traversal that, like our grammar-based approach, only visits relevant nodes[9]. Since their method is geared towards arbitrary data structures instead of just ASTs, however, they are not able to make use of context free grammars to determine what node types can be ignored, and instead must rely on manual annotations in the source code and programmer-provided abstract representations of the structure being traversed in order to achieve a similar result.

Conveniently, Lepper et al. also perform an extensive survey of existing literature on the topic. Most of the sources they found do not immediately address the problem of automatic pruning, and of those that do, most use a similar approach to that of Lepper et al., and none of these makes use of context free grammars (with the exception of [10], which doesn't develop the idea of automatic pruning). In their discussion of [10], Lepper et al. actually seem to be stipulating that our grammar-based optimization would be viable when they remark that in [10], "all possible execution sequences of a certain visitor are translated

into a context-free grammar. In a second step, this approach would also allow automated pruning, but this is not addressed in the paper."[9] This is the only case in the existing literature we are aware of where the idea of automatic pruning visitor pattern traversals is connected with context free grammars as a performance optimization. Thus it would seem that there is ample research space at the moment for a grammar-based optimization of visitor pattern AST traversals, and the idea seems promising, so this could be an excellent topic for future research. That said, we believe that our AST indexing based approach is much more promising, so we do not explore grammar-based optimizations any further in this paper.

Chapter 3

Two Novel AST Indexing Algorithms

3.1 A Naive Approach

The problem our AST indexing algorithms are intended to solve (and what we believe is the most common traversal task of the visitors typically encountered in static analysis) is as follows: Given an AST node A and a node type T, efficiently return the list of all descendants of A that are of type T using some already generated index of the AST in question (with the further stipulation that the index can be generated with reasonable efficiency). In general, indexing an AST in such a way that programmers can have constant time access to lists of descendants of specified types is in a sense trivial because all possible outputs could simply be generated ahead of time and stored in an efficient data structure, such as a hash table. The naive, trivial approach to this sort of indexing, however, requires that each node A stores a

hash table mapping each descendant node type T to a unique list of all descendants of A that are of type T. It is possible to do this using different data structures, but in general, the naive approach requires that each node store some sort of reference to each of its descendants, which requires a substantial amount of memory and an unacceptably lengthy indexing process that is at least quadratic in the total number of nodes.

ASTs are trees, and the theoretical worst case tree for the naive approach in terms of memory overhead would be a linear chain of nodes with no offshoots (that way each node must reference every node that appears at a lower level, because all such nodes would be descendants of the nodes that appear above). In Equation 3.1 below, we derive the asymptotic memory complexity of this worst case scenario, where n is the number of nodes in an AST.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2) \tag{3.1}$$

Thus in the worst case, the naive approach requires a quadratic amount of memory, which is impractical especially in the case of an already large AST with millions of nodes (which we can expect to encounter frequently in practice). Furthermore, the time complexity of generating this index would also be quadratic because doing so entails generating at worst n^2 unique pieces of information. In practice, however, the worst case behavior almost never occurs, because it would be incredibly strange for there to be an AST consisting of a single linear chain of nodes with no offshoots.

Conversely, the best case behavior would consist of a tree with a root node and $n-1$ direct child nodes, with no additional nodes or connections whatsoever. In this best case scenario, only the root node would have children, so we would end up needing only a linear amount of memory (the time complexity would also be linear in this case because leaf nodes

would have empty hash tables, so we would only need to generate information for the root node, and there would only be $n - 1$ pieces of information to generate). That said, the best case would be just as rare (and odd) in practice as the worst case, though something loosely similar could occur, especially in the case of a large header file with hundreds of function prototypes without any function definitions, where we would expect to see a relatively flat AST with most nodes clustered at the root.

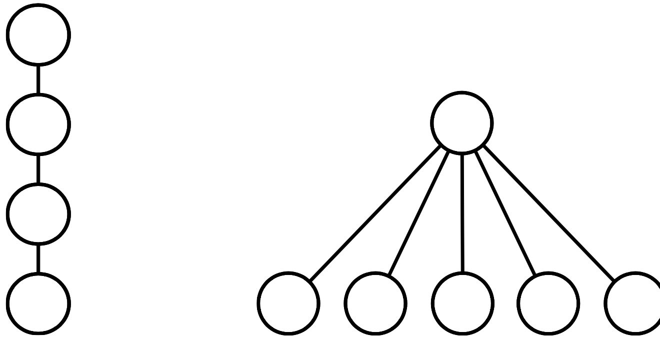

Figure 3.1: Worst case and best case tree configurations in terms of index memory overhead and the time complexity of the naïve indexing process. Worst case is shown on the left, best case is shown on the right.

In practice ASTs tend to be moderately deep, and roughly half way between the best and worst cases shown above, so our assumption is that in practice the naive approach will tend to require an impractical amount of memory, especially with large, complex ASTs (and thus the time complexity would also tend towards something closer to quadratic than linear). We have analyzed a number of real world ASTs, including ASTs generated from popular open source projects such as FFMPEG and gzip, and found that the ASTs generated by these projects so far has been consistent with this hypothesis.

3.2 AST Indexing Overview

Both AST indexing algorithms first perform a heavily modified recursive depth-first

traversal of the AST starting on the root node. They can also run starting on any descendant of the root node using exactly the same process, and this is supported by NodeFinder, however here we only discuss starting on the root node in this paper so as to simplify our description and because one would rarely want to index only part of an AST.

One of the primary intuitions our algorithms rely on is the fact that if the nodes of a tree are added to an array in depth-first order, certain sequential runs of elements within this array will correspond with the sets of descendants for particular nodes in the tree. During a depth-first traversal of an AST (or any tree, for that matter), the visitation order of a node A and its surrounding nodes and descendants can be broken down as follows. All non-descendants of node A to the left of node A in the AST will be visited before we visit node A, because all branches and sub-branches are visited in a depth-first traversal before moving on to any sibling nodes (this is a basic property of depth-first traversals). For this same reason, once we visit node A, all descendants of node A will be visited before any further nodes in the AST can be visited. Thus if we wanted to get the list of all nodes that are the descendants of node A, we could simply take node A's depth-first index as the start index, and the depth-first index of node A's non-child depth-first successor as the end index, and the region encapsulated by these two indices would exactly correspond with the list of descendants of node A (if there are no non-child depth-first successors of A, then the end index would simply be $n - 1$). And so, if we store all nodes of particular types in independent arrays in depth-first order, a node's descendants that are of a particular type can be represented merely as a start and end index into one of these already-generated arrays. This principle, which as we argued above, flows naturally from the basic properties of depth-first tree traversals, can be represented by the following theorem:

> **Theorem 3.1:** In a tree, nodes that are descendants of any node A will have depth first indices greater than the depth first index of A and less than the depth first index of A's non-child depth first successor. If A is the last node in the depth first traversal of the tree, then this upper bound is simply $n-1$ *inclusive* where n is the total number of nodes in the tree.

Thus even though there are a quadratic number of possible outputs that our index has to account for, we can both represent and generate all of this information using sub-quadratic memory and time by relying on overlapping sets of indices (we refer to these as "regions" in the code) into these node arrays. As a result, both of our algorithms have the intriguing property that they are able to generate roughly n^2 pieces of information in better than quadratic time (in the average case) simply because of the overlapping nature of these AST "regions".

3.3 Algorithm A: Overview

Algorithm A is the simpler of the two AST indexing algorithms, has a fast indexing process, but also suffers from slightly slower query speed once an index has been generated. In Algorithm A, the initial depth-first traversal used to generate the AST index runs in $O(n)$ time where n is the total number of nodes in the AST. Thus Algorithm A generates its index in the same asymptotic time taken to perform a standard visitor pattern traversal. Once an index has been generated using Algorithm A, each get-descendants-by-type query is performed using binary search on an array containing all nodes of the specified search type. Thus Algorithm A returns query results in $O(\log t_n)$ average case time where t_n is the total number of nodes of type T in the portion of the AST that has been indexed. Technically, the worst case time would be $O(t_{total} + \ log t_n)$ where t_{total} is the total number of node types in the AST, but this would only occur in the event of an improbable number of hash collisions (as discussed below, the index structure contains a hash table mapping node types to dynamic arrays containing all

nodes of those types). Since on average Algorithm A is logarithmic in terms of the t_n rather than n, it should be substantially faster than using, say, a binary search tree to store the same information, because a binary search tree would be logarithmic in the total number of nodes, rather than in just a small subset as is the case here.

During the initial traversal, an index of the AST is generated and stored as a hash table mapping node types to dynamic arrays, where each array contains all of the nodes of the corresponding type that are found in the AST, listed in depth-first order. Hereafter, we will refer to this structure as "*nodes*". Once *nodes* is fully populated by the indexing process, if one wants to get the list of all nodes of type T that are descendants of node A, one can get this list in $O(\log t_n)$ time merely by performing two binary searches on *nodes*[T] – one binary search for the depth first index of the search root node, and another binary search for the depth first index of the depth first successor of the search root node (or $n - 1$ if no such node exists), as per Theorem 3.1.

It is important to note that *finding* the depth first successor of a node is not always a trivial operation. In the event that the node whose depth first successor we need to find is at the bottom of a long linear chain of nodes, we would have to move all the way up this chain before we could find the depth first successor. To avoid this worst case scenario, Algorithm A also stores the number of descendants of each node as an attribute on each node during the indexing process, since this can be calculated during indexing without impacting time complexity. With this information and given a node A, the depth first index of node A's depth first successor is given by $A.depthFirstIndex$ + $A.numDescendants + 1$ and can thus be calculated in constant time. In the event that node A's depth first successor does not exist (e.g. node A would be the the last node in a depth first traversal), this will be obvious since the

returned index will be equal to the total number of nodes in the AST.

3.4 Algorithm A Implementation Details

Algorithm A begins by initializing the *nodes* composite data structure, as demonstrated below. We also create the integer variable $currentDepthFirstIndex$ to keep track of the current depth first index of the node we are processing. We need this information because Algorithm A labels nodes with their depth first index and number of descendants so that the depth first index of the depth first successor can be found in constant time during queries (note that the depth first index of the depth first successor of node A is merely $A.depthFirstIndex + A.numDescendants + 1$).

```
var nodes ← HashTable<NodeType, DynamicArray<Node>>
var currentDepthFirstIndex ← 0
```

Below is some detailed pseudo-code for the recursive indexing algorithm that must be run on the root of the AST prior to performing any queries using Algorithm A (this is the algorithm that actually builds the index of an AST). Note that the indexing routine for Algorithm A is substantially simpler than that of Algorithm B.

```
function buildIndexA(Node currentNode)
   currentNode.depthFirstIndex ← currentDepthFirstIndex
   currentDepthFirstIndex ← currentDepthFirstIndex + 1
   currentNode.numDescendants ← 0

   if nodes[currentNode.type] is null then
      nodes[currentNode.type] ← DynamicArray<Node>
   end if
   var currentNodes ← nodes[currentNode.type]
   currentNodes.add(currentNode)

   for each Node child in currentNode.children
      buildIndexA(child)
   next
```

```
   if currentNode.parent is not NULL then
      parent.numDescendants ← parent.numDescendants + 1
   end if
end function
```

Once an index has been built, we can use the following function to obtain the list of all descendants of a particular node that are of a particular type in $O(\log t_n)$ time. Note that $binarySearch(index, start, finish, array)$ will search $array$ for the closest match to $index$ using binary search over the inclusive range $[start, finish]$.

```
function getDescendantsByTypeA(Node searchRoot, NodeType T)
   if nodes.containsKey(T) is false then
      return empty result
   end if
   int A ← binarySearch(searchRoot.depthFirstIndex,
           0, nodes[T].length, nodes[T])
   int B ← binarySearch(searchRoot.depthFirstIndex +
           searchRoot.numDescendants, A + 1, nodes[T].length, nodes[T])
   return reference to part of nodes[T] specified by [A, B]
end function
```

Thus using an index generated by Algorithm A is as simple as performing a single hash code calculation on the $nodes$ structure, and then performing two consecutive binary searches on the array returned by $nodes$. In terms of time complexity, there is nothing asymptotically significant that occurs during the initial depth-first traversal, and so Algorithm A indexes an AST in $O(n)$ time where n is the number of nodes in the AST in question. As we have discussed previously, Algorithm A is able to perform get-descendants-by-type queries in $O(\log t_n)$ in the average case once an index has been generated. In terms of memory complexity, the index for Algorithm A is just a hash table of dynamic arrays where the total amount of data stored across the dynamic arrays is n. Thus Algorithm A uses $O(n)$ memory to store its index.

3.5 Algorithm B Overview

Algorithm B is by far the more complex of the two AST indexing algorithms, has a slower indexing process, but offers average case constant time query speed once an index has been generated. In Algorithm B, the initial depth-first traversal has $O(n \cdot t_{avg} \cdot c_{avg})$ average case runtime, where n is the total number of nodes in the AST, t_{avg} is the number of unique node types represented in the descendants of a typical/average AST node[4], and c_{avg} is the number of children of a typical/average AST node (see section 3.7 for a derivation and discussion of the time and memory complexity). During this traversal, an index of the AST is created using two hash-based composite data structures which we will refer to as *nodes*, and *regions* respectively. The *nodes* structure, as it was in Algorithm A, is a hash table mapping node types to dynamic arrays, where each array contains all of the nodes of the corresponding type that are found in the AST, listed in depth-first order. The *regions* structure is a hash table that maps each AST node to its own hash table, which in turn maps particular node types to pairs of start/end indices. These indices indicate the start and end index of the region of the *nodes* array for that type containing the list of nodes of that type that are descendants of that node. Essentially, these pairs are cached results of the binary searches that would normally be performed by Algorithm A. Once these data structures are fully populated by the indexing process, if one wants to get the list of all nodes of type T that are descendants of node A, one can get this list in constant average case time (in the worst case this is linear if there are an improbably high number of hash collisions) merely by looking up *regions*[A][T] and jumping

---

[4] i.e. a hypothetical AST node A where the distribution of node types and number of descendants of A is typical of most nodes in a particular AST. Specifically for a given AST G, $t_{avg} = \frac{\sum_{i=1}^{n} \text{numTypes}(x_i)}{n}$ and $c_{avg} = \frac{\sum_{i=1}^{n} \text{numChildren}(x_i)}{n}$ where $\text{numTypes}(x)$ denotes the number of unique node types represented in the descendants of x, $\text{numChildren}(x)$ denotes the number of nodes that share x as a direct parent, x_i denotes the ith node in G and n denotes the total number of nodes in G.

to the returned start and end indexes in *nodes*[T]. Doing so only requires three $O(1)$ average case hash calculations, and so the overall process runs in $O(1)$ time on average.

3.6 Algorithm B Implementation Details

Algorithm B begins by initializing the *regions* and *nodes* composite data structures. Note that we can represent index pairs or "regions" as 2-tuples of integers. As a general convention, if we have a region 2-tuple, we refer to the first number in the 2-tuple as the *beginIndex*, and the second number in the 2-tuple as the *endIndex*.

```
var regions ← HashTable<Node, HashTable<NodeType, Tuple<int, int>>>
var nodes ← HashTable<NodeType, DynamicArray<Node>>
```

Below is some detailed pseudo-code for the recursive indexing algorithm that must be run on the root of the AST prior to performing any searches (this is the algorithm that actually builds the index of an AST).

```
function buildIndexB(Node currentNode)
   if nodes[currentNode.type] is null then
      nodes[currentNode.type] ← DynamicArray<Node>
   end if
   var currentNodes ← nodes[currentNode.type]
   currentNodes.add(currentNode)

   var currentRegionMap ← HashTable<NodeType, Tuple<int, int>>
   regions[currentNode] ← currentRegionMap

   currentRegion ← Tuple<int, int>
   currentRegion.beginIndex ← currentNodes.length - 1
   currentRegion.endIndex ← currentNodes.length + 1
   currentRegionMap[currentNode.type] ← currentRegion

   for each Node child in currentNode.children
      buildIndexB(child)
      var childRegionMap ← regions[child]
      for each NodeType T in childRegionMap.keys
         var childRegion ← childRegionMap[T]
         var parentRegion ← Tuple<int, int>
```

```
         // if currentNode doesn't have a type entry for this child
         // then we bubble this up directly from the child
         if currentRegionMap[child.type] is null then
            parentRegion.beginIndex ← childRegion.beginIndex
            parentRegion.endIndex ← childRegion.endIndex
         else
            parentRegion ← currentRegionMap[T]
            // if parent region is empty then we overwrite it
            // completely with the child region
            if parentRegion.beginIndex == parentRegion.endIndex then
               parentRegion ← childRegion
            else
               if childRegion.beginIndex < parentRegion.beginIndex then
                  parentRegion.beginIndex ← childRegion.beginIndex
               end if
               if childRegion.endIndex > parentRegion.endIndex then
                  parentRegion.endIndex ← childRegion.endIndex
               end if
            end if
         end if
         currentRegionMap[T] ← parentRegion
      next
   next
end function
```

The indexing function works by performing a recursive depth-first traversal of the AST. The base case creates a "region" 2-tuple containing index information for where the current node is located within the *nodes*[currentNode.type] dynamic array. For each child node, after the recursive call, all of the regions should have been already created for all descendants of the current node, so we proceed to bubble these results up recursively and alter or create, if necessary, the regions associated with the current node. This bubbling up process consists of merging child regions with their parent region. When merging a region A with another region B, we simply set $startIndex = \min(B.startIndex,\ A.startIndex)$ and $endIndex = \max(B.endIndex,\ A.endIndex)$. This way, merged regions represent the *union* of the two regions being merged. Since nodes are added to the *nodes*[type] dynamic arrays in depth-first order, by Theorem 3.1 each of these regions then represents a list of the of descendants of a particular type for a particular node, or rather, a particular solution to the get-descendants-by-

type problem.

Since for each node, we loop through each child node, and for each child node, we loop through each type that is represented in the descendants of that child node, the time complexity of the indexing algorithm becomes $O(n \cdot t_{avg} \cdot c_{avg})$ where t_{avg} is the average number of descendant types for an arbitrary AST node, and c_{avg} is the average number of children for an arbitrary AST node. It is worth noting that theoretically t_{avg} and c_{avg} depend on the layout of the AST in question, rather than on the number of nodes, so these are really both constant in terms of n.

Once an index has been built, we can use the following function to obtain the list of all descendants of a particular node that are of a particular type in constant average case time.

```
function getDescendantsByTypeB(Node searchRoot, NodeType T)
   var relevantRegionMap ← regions[searchRoot]
   if relevantRegionMap is null then
      return empty result
   else
      var resultRegion ← Tuple<int, int>
      relevantRegion ← relevantRegionMap[T]
      if relevantRegion is null then
         return empty result
      else
         if searchRoot.type == T then
            resultRegion.beginIndex ← relevantRegion.beginIndex + 1
         else
            resultRegion.beginIndex ← relevantRegion.beginIndex
         end if
         resultRegion.endIndex ← Max(0, relevantRegion.endIndex - 1)
         if resultRegion.beginIndex == resultRegion.endIndex then
            return empty result
         end if
      end if
      return reference to part of nodes[T] specified by resultRegion
   end if
end function
```

The extra processing on *relevantRegion* is required because as it stands, the indexing algorithm will record nodes as containing themselves as a descendant (writing the indexing

algorithm this way makes the recursion much less confusing, and has no effect on time complexity). To avoid this affecting our results, we check to see if the type we are searching for matches the type of the node whose descendants we are searching, and if this is the case, we offset the region indexes slightly to make sure the node whose descendants we are searching is not included in the result. Since the return value is simply an index into an already-existing array, we technically return this result in constant time in the average case (the routine only costs three hash code calculations and several arithmetic operations and comparisons) rather than in $O(m)$ time where m is the number of nodes in the result (because we are not required to actually iterate over the result being returned). This fact in itself provides us with an even greater advantage over the AST matching algorithm, because it means programmers using our algorithm can avoid ever having to completely iterate over the list of results, even indirectly, if they wish to break early or simply obtain a count (which can also be done in $O(1)$ by examining the size parameter in the result returned by NodeFinder). Thus one could, for example, find the number of variable references contained within a specified function definition in an already indexed AST in constant average case time, without having to explicitly visit or traverse any of these variable references supposing that the AST in question has already been indexed using Algorithm B. In the event of an improbably high number of hash collisions, the worst case time for this lookup would technically be $n + t_{total} + t_{avg} \in O(n)$ where t_{total} is the total number of unique node types in the AST, however this is incredibly unlikely to occur in practice so long as appropriate hashing functions are used.

3.7 Algorithm B Memory & Time Complexity

The primary motivation for using our AST indexing algorithms over the naive approach

is the fact that both of our algorithms have substantially better average case asymptotic memory bounds and time complexity. We have already derived these bounds for Algorithm A, so let us proceed by deriving them for Algorithm B.

Suppose we are working with an AST with n total nodes. By the time Algorithm B has indexed this AST, the *regions* hash table will contain n sub hash tables, one for each node in the AST (as is the case with the naive approach). The difference between our approach and the naive approach is that in our approach, these sub hash tables only contain t_i items where t_i denotes the total number of unique types that are represented within node i's descendants (for leaf nodes, this is zero). Thus if we had a basic block node containing 20 variable declarations and no other descendants, we would only have to store a single entry in the sub hash table for that node rather than 20. Likewise if we had a basic block node containing 20 variable declarations, 20 integer literals, and 20 assignment statements as its only descendants, we would only have to store three entries in the sub hash table for that node rather than 60. The only additional asymptotically significant source of memory usage in Algorithm B comes from the fact that the *nodes* structure must contain n total sub-entries, and thus costs $O(n)$ in terms of memory. Thus the overall memory usage of the index structures for Algorithm B can be represented as follows.

$$n + \sum_{i=1}^{n} t_i \tag{3.2}$$

We can replace the summation of t_i with $n \cdot t_{avg}$ where t_{avg} is the number of unique node types that are represented in the list of descendants for an *average* AST node (which we defined previously). Thus we get the following.

$$n + \sum_{i=1}^{n} t_i = n + n \cdot t_{avg} \in O(n \cdot t_{avg}) \quad (3.3)$$

Thus our algorithm only needs $O(n \cdot t_{avg})$ memory to represent its index in the average case. That said, the average case *time* complexity for generating the index is actually $O(n \cdot t_{avg} \cdot c_{avg})$, where c_{avg} is the number of children of an average AST node. This is because the indexing traversal must visit each node once, and then do some additional processing for each unique node type in the list of descendants for each child node. In practice t_{avg} should be substantially less than n, because ASTs tend to have a large number of leaves, all of which by definition have zero descendants, and so we get $n \cdot t_{avg} \leq n^2$. There is a freak case where $n \approx t_{avg}$, and thus $n \cdot t_{avg} \approx n^2$, however this would be extremely rare in practice because it would involve an AST where there are no two nodes of the same type. Since in practice ASTs tend to have multiple nodes for most if not all of their contained node types, then it should be safe to assume that $t_{avg} \ll n$. Since in practice most nodes in an AST are going to be leaf nodes, n should also dominate c_{avg} such that $c_{avg} \ll n$. Thus it is safe to assume that in practice, even the quantity $t_{avg} \cdot c_{avg}$ would still be dominated by n, meaning that the algorithm should still run in much better than quadratic time. Specifically, it is doubtful that t_{avg} or c_{avg} would increase with respect to n beyond a certain point in practice. Since the naive indexing approach requires $O(n^2)$ memory and time complexity, using our approach, the index should thus use substantially less memory and take less time to generate than the naïve approach in practice.

3.8 Discussion & Theoretical Tradeoffs

The main bottleneck faced by static analysis suites such as Compass is already solved

by both versions of our algorithm since both allow multiple checkers to be run using the same index, and thus require only a single AST traversal. Although Algorithm B has a theoretically slower indexing routine than does Algorithm A, in practice the speed of get-descendants-by-type queries should be much more important. This is especially true in the case of Compass, where potentially hundreds of thousands of these queries will be performed consecutively when the full battery of checkers is run on an AST. Thus our preliminary recommendation is that Algorithm B be used in practice since it is able to perform these queries in constant average case time.

Chapter 4

Benchmarks & Results

4.1 Benchmark Methodology

As part of the NodeFinder implementation, we added a “make benchmark” make action which runs a number of benchmarks comparing performance between Algorithm A, Algorithm B, and Markus’s AST matching algorithm on a variety of queries and indexing tasks. NodeFinder was written in C++, as this is the language ROSE is written in. We made use of the Boost C++ library for the hash table and hash set objects required by our algorithms (specifically, we made use of “unordered_set” and “unordered_map”). Hardware-wise, we ran our benchmarks on a Lenovo ThinkPad X-230 laptop equipped with 8 gigabytes of memory, a 160 gigabyte Intel solid state hard drive, and a quad-core Intel Core i7-3520M CPU, which runs at 2.90 GHz. We ran the actual benchmarks within a Virtual Box virtual

machine running CentOS 6.5 x64 with full CPU virtualization enabled (giving the VM access to all four CPU cores) and 6 gigabytes of dedicated RAM for the VM. Our host OS was Windows 8.1 Professional x64. By default, our benchmark routine automatically disables ROSE asserts, which slightly increases performance.

For our benchmarks, we used an AST generated from the source code of the popular open source gzip compression library, since this should be indicative of performance on a complex, "real world" AST. While this is a large AST, it still only takes up a tiny portion of the nearly six gigabytes of available system memory, so once the source files are read from disk, there shouldn't be any additional IO-related performance bottlenecks. Because of the large asymptotic differences between some of the algorithms being tested (e.g. the difference between constant time and logarithmic time in the case of Algorithm B's query speed vs. Algorithm A), instead of having each algorithm run a specified number of iterations and recording how long this takes, we instead have each algorithm run for a specified amount of time, and record how many iterations it is able to perform over this period.

Originally our benchmarks were limited to the size of the input AST, however we have since developed a routine that automatically partitions an AST into successively smaller sub-trees that can be benchmarked separately. Our routine takes the total size of the AST, and divides it into k successively smaller sizes, shrinking by a factor of $\frac{n}{k}$ with each step where n is the total number of nodes in the AST and k is the number of data points we wish to generate. We then use binary search to search for closest-possible matches to these sizes (in terms of total number of descendants) among the list of all sub-trees of the original AST, and perform benchmarks on the resulting sub-trees. The result is a roughly linear progression of increasingly large ASTs based on the original master AST. This provides us with

graphable data on how our algorithms perform on increasingly large ASTs, thereby allowing us to verify our asymptotic claims about the time complexity of our algorithms.

Test 1 compares the index building speed between algorithms A and B, and also measures the basic traversal speed of the AST matching algorithm performing a basic query[5], since AST matching does not generate an index. This gives us a useful reference, since internally the AST matching algorithm is also performing an AST traversal. Test 2 uses these already generated indexes to compare the speed of basic root node level queries searching for all variable references that are descendants of the root node (the AST matching algorithm simply performs this basic query over and over again as a reference, once again). Test 3 performs a "nested" query. This entails finding all function definitions that are descendants of the root node and then finding all variable reference descendants of each of these function definitions. Thus if there are k function definitions in an AST, Test 3 would end up running each algorithm $k + 1$ times: one time for the top level query searching for function definitions, and an additional time for each of the k function definitions that are found. Here the AST matching algorithm and our algorithms perform the same query, however our algorithms have the added advantage of a pre-computed index. This sort of nested query is typical of most of the simpler varieties of queries that are commonly encountered within Compass checkers[2]. Finally, Test 4 performs a "triply nested" query, which entails finding all function definitions that are descendants of the root node, then finding all if-statement descendants of each of these function definitions, and finally finding all variable reference descendants for each of these if-statements. Thus there are three levels of get-descendants-by-type queries. This set of queries is typical of some of the more complex Compass checkers, particularly the 600+ line memory

[5] The query is "$v=SgVarRefExp" performed on the root node, so this will return all variable references that are descendants of the root node of the AST.

leak detection checker created as part of Compass2[2].

We first ran these tests on timing intervals of 10 and 60 seconds so we could verify that the speed of each algorithm stays proportionally consistent regardless of how long we run the test. We then used our AST partitioning routine to run benchmarks on increasingly larger ASTs, and we generated graphs of this data. We used the POSIX clock() call to measure elapsed time, because this will return elapsed CPU time *only* for the current process, meaning we will not have any interference from other unrelated processes taking up CPU time[11]. This will give more accurate results than if we had used a more typical time function.

4.2. Static Sized AST Results

Table 4.1 below depicts the raw data that was generated when we ran our suite of benchmarks on the 10 CPU second timing interval. Table 4.2 shows the data that was generated for the 60 CPU second timing interval. As predicted, both sets of data were proportionally consistent in that the 60 second test was able to perform roughly 6 times the number of iterations and the 10 second test for each algorithm.

Table 4.1: The data from benchmark tests 1-4 is shown for the 10 CPU second timing interval for the AST matching algorithm, Algorithm A, and Algorithm B. Data is expressed as the number of iterations the corresponding algorithm was able to run within 10 CPU seconds. Thus a higher number means the corresponding algorithm was faster.

Test #	AST Matching	Algorithm A	Algorithm B
1	437	855	211
2	675	1,221,726	47,427,057
3	495	55,384	5,935,149
4	369	6816	551,296

Table 4.2: The data from benchmark tests 1-4 is shown for the 60 CPU second timing interval for the AST matching algorithm, Algorithm A, and Algorithm B. Data is expressed as the number of iterations the corresponding algorithm was able to run within 60 CPU seconds. Thus a higher number means the

corresponding algorithm was faster.

Test #	AST Matching	Algorithm A	Algorithm B
1	2,869	4,329	1,133
2	3,444	6,219,825	271,204,769
3	2,396	300,368	32,406,705
4	1,891	41,570	3,122,255

In Test 1 (the basic indexing test), we see as expected that Algorithm A is substantially faster than Algorithm B since its indexing traversal is linear in the number of nodes being indexed (in the average case), though Algorithm B's indexing method only seems to be slower than that of Algorithm A by roughly a factor of 4 (note that this factor is likely specific to the AST in question).Algorithm A's indexing algorithm managed to be roughly 1.5 times faster even than Markus's AST matching traversal. This is likely due to the fact that the AST matching algorithm has some additional work it must perform to facilitate some of its more advanced queries, especially the "where" clause[2].

In Test 2 (the root level query test), we see as expected that both Algorithm A and B are substantially faster than the AST matching algorithm, since A and B both have the asymptotic advantage of a pre-computed index, whereas the AST matching algorithm must re-traverse the entire AST for each query. Also note that as expected, Algorithm B is substantially faster than Algorithm A (by a factor of roughly 44), since Algorithm B has constant average case access to query results, whereas Algorithm A takes $O(\log t_n)$ time to look up a result in its index.

In Test 3 (the nested query test), the same relationship we saw in Test 2 is stretched even further – the AST matching algorithm runs roughly twice as slow because the use of nesting, and the difference between Algorithm A and B becomes even more drastic in that B runs faster than A roughly by a factor of 108. This suggests that the more levels of nesting

we add, the greater the degree to which Algorithm B outperforms Algorithm A in terms of query speed.

Finally in Test 4 (the triple nested query test) we see the AST matching algorithm's performance decreases again, this time by a factor of roughly 1.3 (recall that in the move from Test 2 to Test 3 this decreased by a factor of 1.5). Likewise, Algorithm B remains roughly 75 times faster than Algorithm A.

4.3 Dynamically Sized AST Results

The graphs below depict the results of running benchmarks on 16 AST sub-trees ranging from size 45 to size 929 that were found using our AST partitioning routine within the main gzip AST. We used a timeframe of 5 CPU seconds for each algorithm. Since we measure the number of iterations that were able to complete within the given timeframe rather than the actual time taken per iteration (which is the information that we want to graph), we converted this data by dividing $5000/x$ for each data point where x is the original data point (i.e. x is the number of iterations that completed within 5 CPU seconds for that algorithm). This gives us graphable data telling us the average number of milliseconds it takes for a single iteration to run. The raw data from these benchmarks can be found in Appendix A.

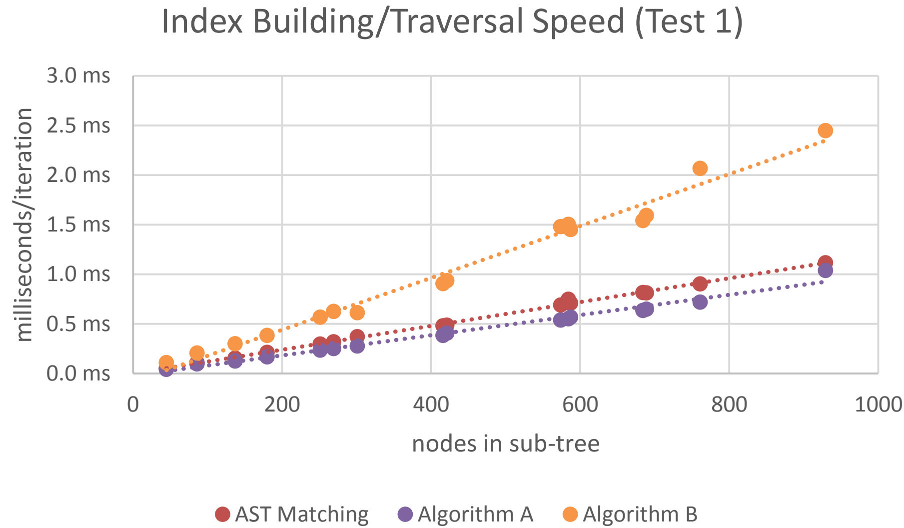

Figure 4.1: Comparison of the index building speed (in the case of Algorithm A and B) and traversal speed (in the case of the AST matching algorithm) on ASTs of increasing sizes.

In Figure 4.1, Algorithm A and AST matching both have an obviously linear runtime, and this is consistent with our previous predictions. We also see again that the AST matching algorithm is slightly slower than Algorithm A's indexing routine by a constant factor, likely because some of the extra work the AST matching algorithm has to perform to facilitate advanced queries. Theoretically, Algorithm B's indexing routine runs in $O(n \cdot t_{avg} \cdot c_{avg})$ time, but we predicted that in practice t_{avg} and c_{avg} would not increase with respect to n beyond a certain point in practice, thus functioning merely as an extra constant factor and resulting in an $O(n)$ runtime. In the graph, we see that Algorithm B's indexing routine is clearly linear in practice, albeit with a significantly worse constant factor than that of Algorithm A or the AST matching algorithm, so this is consistent with our hypothesis. In practice, it seems that Algorithm B's indexing routine does indeed run in linear time, as does Algorithm A.

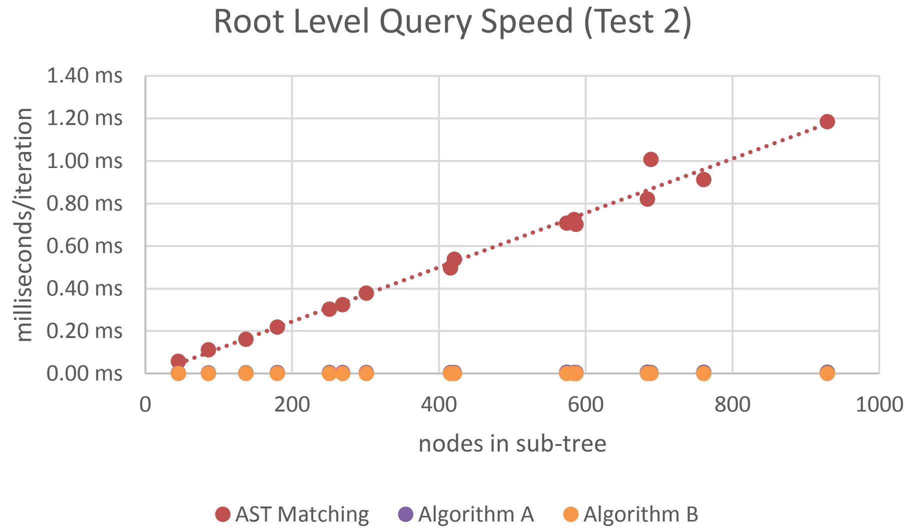

Figure 4.2: Comparison of root level query speed (Test 2) for all three algorithms.

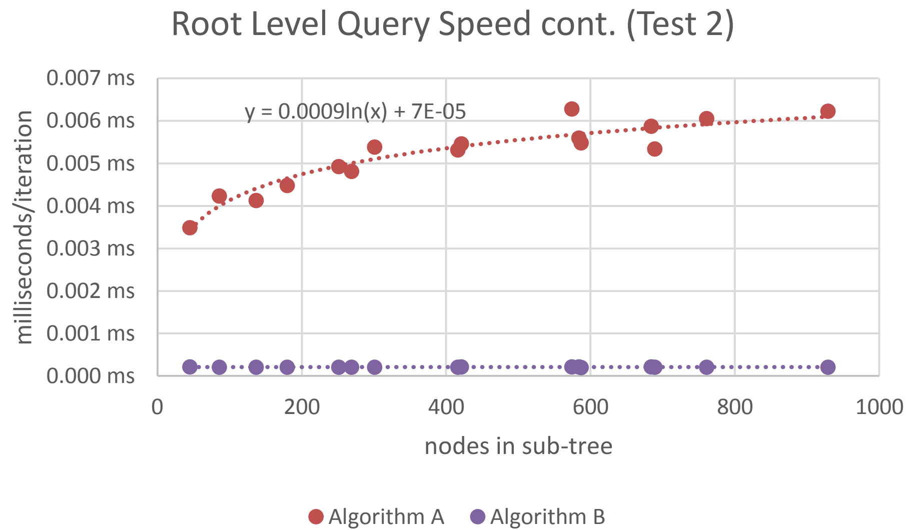

Figure 4.3: Comparison of root level query speed (Test 2) just for algorithms A and B.

As we can see clearly in Figure 4.2, the AST matching algorithm takes linear time to perform root level queries whereas algorithms A and B seem to be running in constant or near-constant time. In Figure 4.3, we compare algorithms A and B directly and find that Algorithm B runs in constant time, whereas Algorithm A runs in very slightly logarithmic

time with respect to n. Recall that the average case query speed for Algorithm A is $O(\log t_n)$ where t_n is the number of nodes in the AST that are of the designated search type. This does increase with respect to n, but $O(\log t_n)$ is still substantially faster by a constant factor than $O(\log n)$, and in practice still seems quite close to constant time. That said, while A and B are close, B has an asymptotic advantage, and this becomes significant if millions of consecutive queries are performed.

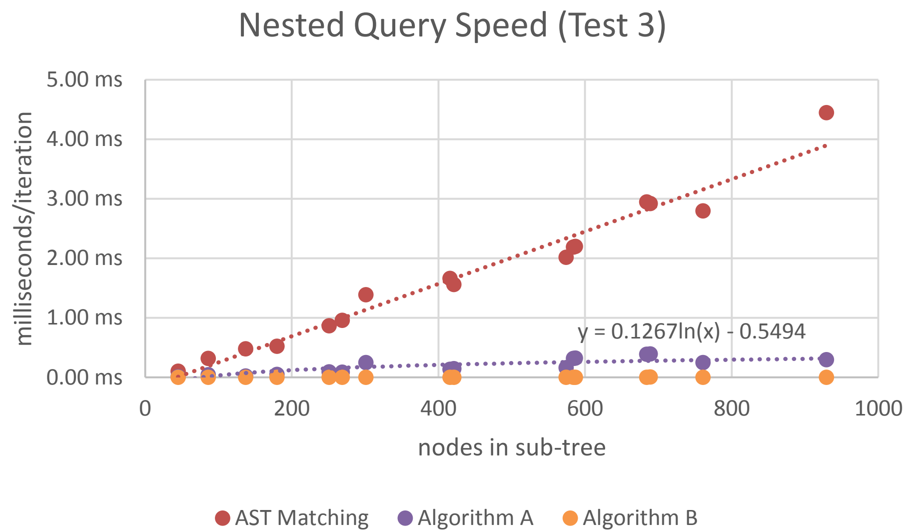

Figure 4.4: Comparison of nested queries for all three algorithms (Test 3).

The results in Figure 4.4 are essentially the same as they were for root level query speed (in particular, Algorithm A still takes logarithmic time), except the algorithms whose runtime depends on n (i.e. the AST matching algorithm and Algorithm A) are slower by a constant factor since many more queries are being run per iteration than with Test 2. Note once again that Algorithm B is clearly the best choice here, with its extremely efficient constant time query speed.

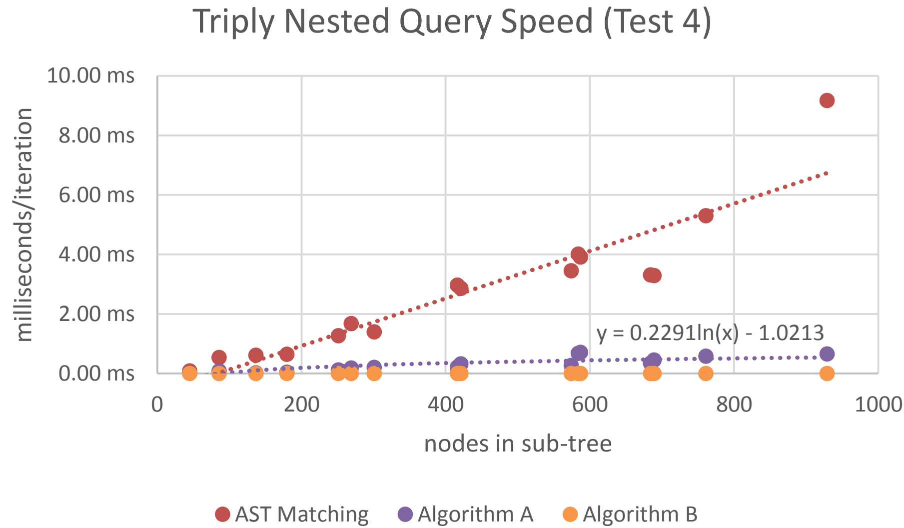

Figure 4.5: Comparison of triply nested queries for all three algorithms (Test 4).

In Figure 4.5, we once again, see the same trends from Test 3 continued. Algorithm B still has constant time performance, whereas Algorithm A and the AST matching algorithm are slower than they were in Test 3 by a constant factor due to the increased number of queries per iteration that must be performed to achieve this level of nesting.

4.4 Discussion

While our results are limited at the moment in that they are merely a case study of the AST generated by ROSE for the gzip source code, the results we have so far corroborate our theoretical assumptions about the time complexities of the various algorithms being tested. As predicted, Algorithm A is very fast at indexing ASTs – even faster than the AST matching algorithm is at traversing ASTs – but is relatively slow at performing get-descendants-by-type queries when compared with Algorithm B, which on the other hand has a slightly slower indexing/traversal process than Algorithm A. Algorithm B's indexing traversal was also slightly slower than the traversals performed by the AST matching

algorithm. That said, in Chapter 3 we argued that at least for applications such as Compass, get-descendants-by-type query performance is vastly more important than traversal/indexing performance since indexes only have to be generated once for any given AST, whereas queries might be performed millions of times consecutively, and furthermore might be nested down several levels (which, as we have demonstrated, only compounds the relative advantage of having a pre-computed index over performing pure traversals as the AST matching algorithm does). Based on these considerations and our data, it follows that Algorithm B should be vastly preferable to Algorithm A in practice – its indexing process seems to be only slightly slower than that of A in practice, and the advantage of Algorithm B's average case $O(1)$ time query results over the $O(\log t_n)$ time offered by Algorithm A is profound for large n. Thus the checkers in Compass would benefit greatly from being rewritten using Algorithm B and/or from the AST matching algorithm having NodeFinder included on basic queries (queries that just solve the get-descendants-by-type problem and don't make use of any of the advanced AST matching features) as an extra optimization.

Chapter 5

Future Work

5.1 Dealing with DAGs

By default, ASTs created by ROSE are trees and thus cannot contain any directed acyclic graphs[4] (in a tree, for all nodes A, there can only be one unique path leading from the root node to A). Some projects in ROSE, however, make use of ROSE's built-in AST merging capability, which can result in ASTs that *do* contain DAGs. If NodeFinder is to be useful to

ROSE users in general, it thus should work in scenarios where DAGs might be present in the AST being indexed. After a preliminary investigation, we suspect that NodeFinder already works with ASTs containing DAGs with the exception that repeated nodes may or may not be repeated in the results depending on the structure of the AST in question. Thus some future work might include verifying conclusively how NodeFinder performs when asked to index ASTs containing DAGs. AGs.

5.2 Performance Comparison with the ROSE AstQuery Library

ROSE's built-in AstQuery library is similar to NodeFinder, and therefore would be an ideal subject to test against NodeFinder. Like the AST matching library, the AstQuery library looks for particular types of nodes while it traverses an AST and returns these nodes, thus solving the get-descendants-by-type problem. That said, there are a number of key differences between AST matching and the AstQuery library that are beyond the scope of this paper. While the AstQuery library does not offer the same asymptotic benefits NodeFinder can offer for solving the get-descendants-by-type problem (we currently have no reasons to believe that the AstQuery approach will be faster than NodeFinder asymptotically) it will be important moving forward to perform some benchmarks that compare AstQuery performance with that of NodeFinder given that it is heavily used within ROSE[2].

5.3 Parallelizing NodeFinder

We have also devised a potential method by which NodeFinder could be parallelized in the future. Parallelizing the indexing process would entail somehow partitioning the AST in question into k approximately equal sub-trees where k is the number of cores available for

parallelism. In this way, the indexing process can be made to be embarrassingly parallel, though the problem remains of how to partition the AST equally without traversing it beforehand. Heuristics based on average-case properties of real-world ASTs might prove useful here. Another method might be to cache statistical information about the distribution of nodes in particular ASTs in a file, so that our algorithms will be aware of how to efficiently partition an AST they have seen in a previous run, though this might be of limited use, since it would probably be rare in practice that one would run the same analysis suite multiple times on the same AST. That said, doing so would be of little consequence because as we have argued previously, get-descendants-by-type query performance is vastly more important than indexing performance, (especially for applications such as Compass), so parallelizing the latter is probably much more important. This would also require more complicated query result representations – results would need to span across multiple arrays that exist in multiple indexes, though this shouldn't affect things asymptotically. In NodeFinder, random access reads from the results arrays are all intercepted on an individual basis already, so this process could easily be extended to support results that span separate arrays.

Since with both algorithms, the index data structures can be considered read-only once an index has been generated, get-descendants-by-type queries are already embarrassingly parallel. Thus multiple cores could easily perform simultaneous unrelated get-descendants-by-type queries if a shared memory system were employed, so it would be relatively easy to parallelize all sequential, unrelated get-descendants-by-type queries (related queries would still have to be executed sequentially – i.e. queries where the result of query B depends on the result of query A, as is the case with a nested query).

Appendix A: Raw Dynamic Benchmark Results

Below we list the raw benchmark data we used to generate the graphs for the dynamically sized AST benchmarks shown in section 4.3. This data is expressed in terms of the number of iterations each algorithm was able to run for each AST sub-tree size within a 5 CPU second time window on the specified benchmark test. Thus higher numbers in these tables are indicative of an algorithm being faster.

Index Building / Traversal Data

Nodes	AST Matching	Algorithm A	Algorithm B
45	87259	115110	44507
86	45091	51553	23961
137	32266	39400	16684
180	23510	29551	12944
251	16742	21207	8792
269	15600	19906	7963
301	13417	17911	8140
416	10356	12951	5515
421	10264	12290	5335
574	7247	9276	3373
584	6659	9062	3322
587	7052	8768	3445
684	6131	7851	3241
689	6138	7701	3138
761	5536	6943	2418
929	4479	4816	2042

Root Level Query Data

Nodes	AST Matching	Algorithm A	Algorithm B
45	86350	1435556	23438175
86	44908	1180858	24456730
137	31076	1211262	24509986
180	22928	1116399	24231793
251	16520	1015235	24191976
269	15414	1039306	24644844
301	13196	928425	24354822

416	10058	941226	24692447
421	9288	916679	23162954
574	7069	796713	23347066
584	6903	893679	23152975
587	7121	911745	24798067
684	6095	851993	23109009
689	4966	936288	24733295
761	5477	825940	23926605
929	4219	803356	24416672

Nested Query Data

Nodes	AST Matching	Algorithm A	Algorithm B
45	47410	354881	18255180
86	15613	115214	9286579
137	10474	216149	15919845
180	9525	97235	9348036
251	5775	54455	5338800
269	5214	55878	4778463
301	3598	20115	1950826
416	3005	36351	3651609
421	3211	34724	3289580
574	2481	30422	2999956
584	2287	15717	1695737
587	2276	15392	1688552
684	1698	12930	1336418
689	1714	12675	1277045
761	1787	20203	2065541
929	1124	16735	1669345

Triply Nested Query Data

Nodes	AST Matching	Algorithm A	Algorithm B
45	55224	384354	16526104
86	9193	55155	4371643
137	8089	211398	12834702
180	7759	103005	7681921
251	3936	41880	3446946
269	2974	27002	2434008
301	3579	23546	1906380
416	1686	23027	2125954
421	1746	15222	1488232
574	1447	18481	1843928

584	1246	7349	691364
587	1276	7043	726843
684	1508	14543	1239697
689	1518	11090	972105
761	943	8544	819455
929	545	7529	741005

References

[1] D. Quinlan, C. Liao, J. Too, R. Matzke, and M. Schordan, "ROSE Compiler Infrastructure," 2013. [Online]. Available: http://www.rosecompiler.org.

[2] D. Quinlan, C. Liao, M. Schordan, and J. Too, "ROSE Compiler Source Code (EDG4)," *GitHub*, 2013. [Online]. Available: https://github.com/rose-compiler/edg4x-rose.

[3] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, & Tools (2nd Edition)*. Boston: Pearson - Addison Wesley, 2007.

[4] D. Quinlan, C. Liao, M. Schordan, T. Panas, and R. Matzke, "ROSE User Manual: A Tool for Building Source-to-Source Translators Draft User Manual (version 0.9.5a)," 2013.

[5] A. Gibbons, *Algorithmic Graph Theory*. Cambridge University Press, 1985, p. 259.

[6] G. Graefe, *Modern B-Tree Techniques*. Now Publishers Inc, 2011, p. 208.

[7] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[8] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts with Java, 8th Edition*. Wiley, 2009, p. 1040.

[9] M. Lepper and B. Trancón y Widemann, "Optimization of Visitor Performance by Reflection-Based Analysis," in *Theory and Practice of Model Transformations*, vol. 6707, J. Cabot and E. Visser, Eds. Springer Berlin Heidelberg, 2011, pp. 15–30.

[10] M. Schordan, "The Language of the Visitor Design Pattern," *J. Univers. Comput. Sci.*, vol. 12, no. 7, pp. 849–867, Jul. 2006.

[11] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, vol. 2005. "O'Reilly Media, Inc.," 2005, p. 944.