On the Study of Tree Pattern Matching Algorithms and Applications

by

Fei Ma

B.Sc., Simon Fraser University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

August, 2006

Abstract

Solving Tree Pattern Query problem is the central part of XML database query. This thesis presents several new algorithms to solve Tree Pattern Query problem and its variations. All of them have better time complexity than any existing algorithms.

This thesis also describes the design, implementation, and application of two new algorithms to detect cloned code. The algorithms operate on the abstract syntax trees formed by many compilers as an intermediate representation. They extend prior work by identifying clones even when arbitrary subtrees have been changed. On a 440,000-line code corpus, 20-50% of the clones found eluded previous methods.

Table of Contents

List of Figures

List of Algorithms

Acknowledgements

I would like to thank Dr. Will Evans for his guidance, patience and dedication to teaching. Thanks also to Dr. Rachel Pottinger for her insightful advice on this work. Last but not least, thanks to my wife, Diane, for you always being the first audience of my ideas.

To my parents.

Chapter 1

Introduction

1.1 Background

We first introduce some common definitions and notations associated with trees. This will allow us to define the problems addressed in this thesis in Section 1.2.

1.1.1 Tree

In graph theory, a *tree* is a connected, acyclic, undirected graph. Trees, especially *rooted trees*, for which one node is designated the *root*, are important data structures used in computer science.

The relationships between nodes of a rooted tree are named analogous to family relations.

- Node n_1 is a *ancestor* of node n_2 if it is on the path of node n_2 to the root. Node n_2 is called a *descendant* of node n_1.
- Node n_1 is the parent of node n_2 if it is a ancestor of node n_2 and is connected to node n_2. Node n_2 is called a child of node n_1.
- Nodes having the same parent are called *siblings*.

Nodes with no children are called *leaf nodes*, and the other nodes are called *internal nodes*.

An *ordered tree* is a rooted tree in which the children of each node are ordered. In other words, if a node has k children, we can refer to them as the first child, the second child, and so on. For *unordered trees*, there is no such distinction between the children of a node.

A *labeled tree* is a tree in which each node is given a label.

The *level* of a node is the length of the path from the root to the node. The *height* of a tree is the length of the path from the root to its furthest leaf.

Figure 1.1 shows an example of a labeled rooted tree, where nodes are labeled with characters inside them and node A is the root. The height of the tree is 2. Node A is at level 0, node B and C are at level 1, and node D and E are at level 2.

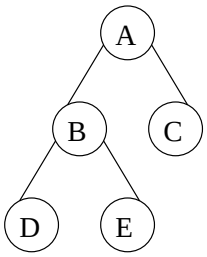

Figure 1.1: A labeled rooted tree

1.1.2 Tree Traversal

Tree traversal is the process of visiting each node in a tree. *Pre-order traversal* is most commonly used, which can be done by calling the procedure VISIT defined in Algorithm 1.1 on the root of a tree.

For ordered trees, the children of each node are visited in their predefined order; for unordered trees, an arbitrary order (e.g., alphabetical order) can

Algorithm 1.1 Pre-order traversal of a tree

```
VISIT(node)
    PROCESS(node)
    for each child of node
    VISIT(child)
```

be used. For the tree in Figure 1.1, pre-order traversal will visit the nodes in the order of A, B, D, E, C.

If instead we visit the child subtrees first, then the current node, this produces *post-order traversal*.

1.1.3 Positional Notation for Tree Nodes

Different nodes of a tree can use the same label, so it is more convenient to have a unique notation to refer to the nodes of the tree than to use the labels directly. In most cases, pre-order numbering or post-order numbering is used, which can be done by assigning the next available number to each tree node along the way of pre-order or post-order traversal of a tree.

In this thesis, we will use a different positional notation for tree nodes, which is first introduced and used in [11, 12, 22]. As shown later, this positional notation provides more benefits when determining the relationship of two tree nodes.

The positional notation of each tree node is a 3-tuple of integers: $< start, end, level >$, which can be computed by calling the procedure NUMBERING in Algorithm 1.2 on the root of the tree.

Figure 1.2 shows the same tree from Figure 1.1 with the positional no-

Algorithm 1.2 Computing positional notations for a tree

```
NUMBERING(node)
    ▷ N and L are global variables, initially all −1.
    N ← N + 1
    L ← L + 1
    node.start ← N
    node.level ← L
    for each child of node
        NUMBERING(child)
    N ← N + 1
    L ← L − 1
    node.end ← N
```

tation for each node of the tree annotated. [1]

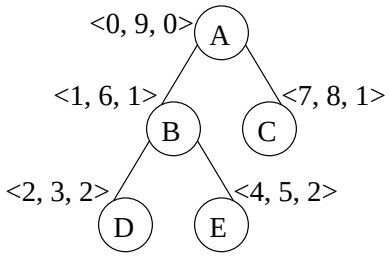

Figure 1.2: Positional notation of a tree

The relationship of two tree nodes can be determined in constant time by Proposition 1.1.

Proposition 1.1. *Given the positional notations of node n_1 and n_2,*

1. *n_1 is an ancestor of n_2 iff $n_1.start < n_2.start$ and $n_1.end > n_2.end$.*
2. *n_1 is the parent of n_2 iff n_1 is an ancestor of n_2 and $n_2.level = n_1.level + 1$.*

[1] Some authors use the same number as both start and end position for leaf nodes.

1.1.4 Tree Pattern Matching

Let pattern P and target T be ordered labeled trees, P matches T at node v if there exists a mapping from the nodes of P into the nodes of T such that

1. the root of P maps to v,
2. if x maps to y, then x and y have the same labels, and
3. if x maps to y and x is not a leaf, then the i^{th} child of x maps to the i^{th} child of y. [2]

If P and T are unordered labeled trees, then the mapping from nodes of P into nodes of T is defined as:

1. the root of P maps to v,
2. if x maps to y, then x and y have the same labels, and
3. if x maps to y and x is not a leaf, then each child of x maps to some child of y.

Figure 1.3 shows an example of tree pattern matching between pattern tree P and target tree T, where the one-to-one mappings are illustrated by dotted lines connecting corresponding pairs of matching nodes.

[2] Node y may have more children than node x.

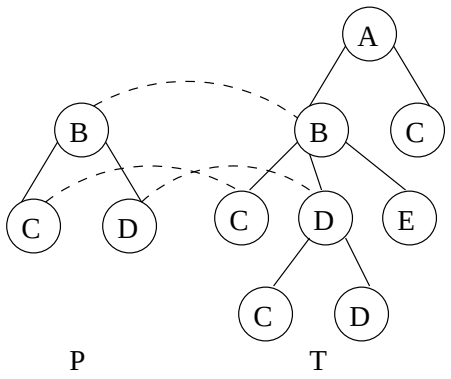

Figure 1.3: An example of tree pattern matching

1.2 Problems Studied

In this thesis we consider two problems related to pattern discovery in trees. The first problem, *Tree Pattern Query*, is to find all copies of a given tree pattern in a large target tree. The second problem, *Clone Detection*, is to find sets of identical or near-identical subtrees in a given tree. We introduce these problems in the next two sections. Studying and solving these problems is the major task of this thesis.

1.2.1 Tree Pattern Query

The tree is a widely-used data structure in computer science, and there are many applications requiring comparison between trees. The problem of tree pattern matching has been well studied since the early 1980's. However, most algorithms (see [2] for a complete review of these algorithms) rely on the assumption that the target tree can be held entirely in main memory, which is not valid for many modern applications using tree data structures, such as an XML database.

Motivated by the classical inverted list data structure in information retrieval [19], researchers model very large trees as streams of positional notations of tree nodes clustered by their labels [1, 7, 22]. Although in the streams there is no pointer available for any node to its parent or children, the relationship between any pair of nodes can still be easily determined using their positional notations in the way explained in Section 1.1.3. Data streams are normally stored on secondary storage, e.g., disk. Figure 1.4 shows an example of a target tree and its corresponding data streams.

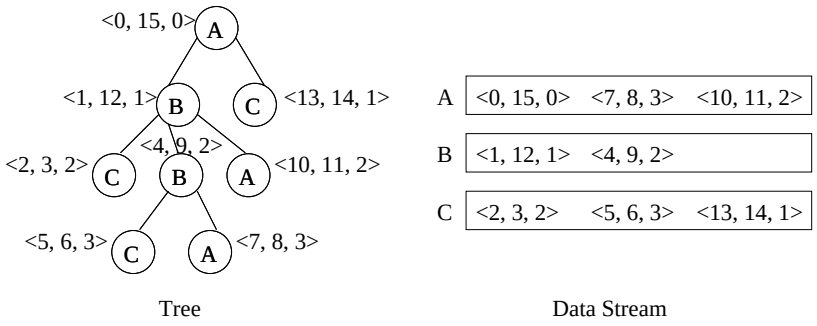

Figure 1.4: A tree and its stream format

The change of the input format has great impact on the design of an efficient algorithm. For example, sequential scan of the input streams is preferable to random access, and the number of scans becomes an important factor in the evaluation of algorithms since it directly affects I/O complexity. In this sense, most of the existing algorithms are not suitable or not efficient any more, even with modifications.

In most modern applications pattern trees are still considered small enough to be held in main memory and accessed using traditional tree traversal methods, but pattern trees become more complex with the use of wildcard nodes and ancestor-descendant edges [8, 13, 18]. A wildcard

node, usually labeled with "?" in the pattern tree, can be mapped to a node with arbitrary label in the target tree; an ancestor-descendant edge, usually denoted by a doubled line in the pattern tree, allows the pair of nodes connected by it in the pattern tree to be mapped to a pair of nodes with ancestor-descendant relationship in the target tree even if the two nodes are not in a parent-child relationship.

Furthermore, instead of finding one matching between the pattern tree and target tree, most modern applications require the output of all possible matchings, and the output of each matching includes not only the target tree nodes matching the pattern tree root, but also other target tree nodes matching the non-root nodes of the pattern tree.

Different requirements generate various problems different than the traditional tree pattern matching problem defined in Section 1.1.4, and bring the challenge of designing new efficient algorithms.

The *Tree Pattern Query* (TPQ) problem is to search for all matchings of a relatively small pattern tree over a large target tree. [3] The pattern tree P contains both ancestor-descendant edges and parent-child edges. The target tree T is very large and is stored in the format of data streams as explained in Section 1.2.1. The Tree Pattern Query problem has application to querying XML databases [8, 13].

Definition 1.1 (*TPQ Matching*)**.** A *matching* in Tree Pattern Query problem is a function M that maps each node of the pattern tree P to a node of the target tree T such that:

1. x and $M(x)$ have the same label, and

[3] The target could also be a forest of trees, In which case, we assume there is a virtual root whose children are the roots of the individual trees.

2. whenever (x, y) is a parent-child edge (resp., ancestor-descendant edge) in P, $M(x)$ is a child (resp., descendant) of $M(y)$ in T.

Note that matchings in Tree Pattern Query problem are unordered. It is different from the unordered matching defined in Section 1.1.4 due to the existence of ancestor-descendant edges in the pattern tree. Figure 1.5 shows an example of a TPQ matching from nodes of a pattern tree to nodes of a target tree. In the target tree in Figure 1.5 nodes having the same labels are annotated with subscripts for distinction.

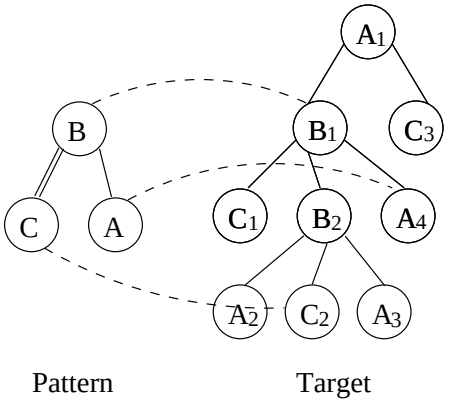

Figure 1.5: An example of Tree Pattern Query problem

The output of the Tree Pattern Query problem is the set of all possible mapping functions. For example, given the pattern tree and target tree in Figure 1.5 the output of the Tree Pattern Query problem is a set $\{(B \to B_1, C \to C_1, A \to A_4), (B \to B_1, C \to C_2, A \to A_4), (B \to B_2, C \to C_2, A \to A_2), (B \to B_2, C \to C_2, A \to A_3)\}$.

Definition 1.2 (*root-match*)**.** Given a pattern tree P and a target tree T, a root-match is a node of the target tree, to which is mapped the root of the pattern tree in some matching(s) from P to T.

In some cases we are only interested in outputting all root-matches instead of all matchings, we call this type of problem the *Simple Tree Pattern Query* problem. For example, given the same pattern tree and target tree in Figure 1.5, the output of the Simple Tree Pattern Query problem is $\{B_1, B_2\}$.

If the pattern tree contains only ancestor-descendant (resp. parent-child) edges, we call this type of problem the *AD-edge-only (resp. PC-edge-only) Tree Pattern Query* problem.

If wildcard nodes are allowed in the pattern tree, we call this type of problem *Tree Pattern Query with Wildcards*. A wildcard node in the pattern tree can match a node with arbitrary label in the target tree. For example, if node "A" of the pattern tree in Figure 1.5 is replaced with a wildcard node, then the function $(B \to B_1, C \to C_1, ? \to B_2)$ is also a valid matching.

1.2.2 Clone Detection Using Abstract Syntax Tree

Given a piece of source code, the problem of clone detection is to find all clones inside it. Clones can be copied code fragments or common programming concepts.

There are various reasons why clones occur.

1. In legacy code, which is constructed by less structured design processes and formal reuse methods, a considerable amount of code is created by reusing existing code. To be more specific, programmers search for and copy existing code that has functionality similar to the desired one, and then modify the copy to implement the exact functionality.

2. Clones occur when programmers break the software engineering principal of encapsulation. For example, operations on a certain data type

may be copied at various locations in a program instead of being placed in a library.

3. Some code fragments, such as user interface styles, may need to be duplicated.

4. Clones may occur, without copying and pasting, when a computation is simple enough so that a programmer may use a mental macro to write essentially the same code each time the computation needs to be carried out.

Cloning makes it harder to maintain, update, or otherwise change the program. For example, when an error is identified in one copy, then the programmer must find all of the other copies and make parallel changes or refactor the code. A tool that automatically detects clones can aid the programmers in detecting these potential sources of error. Once clones are detected, the programmer might create a new procedure that abstracts the clones. Any differences between the clones would then become arguments passed to the procedure. After that, software maintenance costs would be decreased correspondingly due to the reduction in code size.

An abstract syntax tree (AST) is an ordered labeled tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators, e.g., variables or constants. Figure 1.6 shows an example of a code fragment and its abstract syntax tree. Note that there is no canonical way to generate an AST for a given source code, but we assume in this thesis that we have chosen a fixed method for AST generation.

Given an AST the problem of *Clone Detection using Abstract Syntax Tree* is to find all large patterns in the AST that occur more than once. A pattern

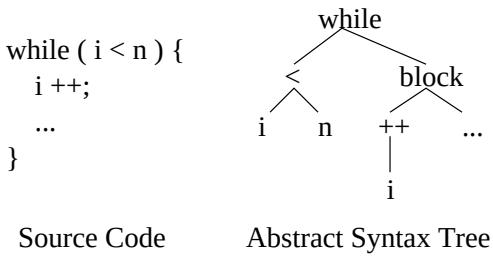

Figure 1.6: Abstract Syntax Tree Example

is a subtree that matches a subtree of an AST (and occurs at that point) if they match in the sense of an ordered Tree Pattern Query match, and every pair of matching nodes has the same degree. Defining clones based on the AST, rather than on a more typical lexical approach, allows the recognition of clones that differ in more than just variable names. For example, the AST approach would detect that `while( i != j) { i++; ... }` matches the source given in Figure 1.6 except for the while-condition.

In Chapter 3 we will describe algorithms for performing Clone Detection using Abstract Syntax Tree and investigate their performance in relation to lexical approaches. Our results indicate that AST-based methods do find more interesting clones than lexical methods, and that the AST-based methods are efficient in practice.

1.3 Chapters Overview

In Chapter 2 we will present several new algorithm for TPQ problem and its variations. All of them are both sound and complete, and have better time complexity than any existing algorithms.

In Chapter 3 we will describe the design, implementation, and application of two new algorithms to detect cloned code using Abstract Syntax Tree. They extend prior work by identifying clones even when arbitrary subtrees have been changed.

Chapter 2

Algorithms for Tree Pattern Query Problem and Its Variations

2.1 Review of the Problems and Prior Work

2.1.1 Review of the Problems

The input of Tree Pattern Query Problem is a pattern tree P containing both ancestor-descendant edges and parent-child edges, and a target tree T stored as data streams. The output of the Tree Pattern Query Problem is a set of all possible matchings from P to T. For example, the output of the Tree Pattern Query in Figure 2.1 is $\{(B \to B_1, C \to C_2, A \to A_4, D \to D_1), (B \to B_2, C \to C_2, A \to A_2, D \to D_1), (B \to B_2, C \to C_2, A \to A_3, D \to D_1)\}$.

The output of the Simple Tree Pattern Query Problem is a set of all root-matches instead of matchings. For example, the output of all root-matches for the problem in Figure 2.1 is $\{B_1, B_2\}$.

If the pattern tree only contains ancestor-descendant (resp. parent-child) edges, we call the problem AD-edge-only (resp. PC-edge-only) Tree Pattern Query problem.

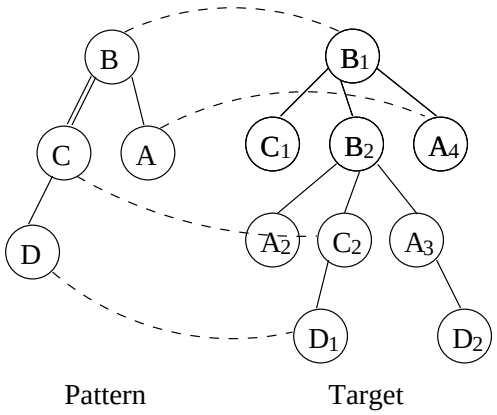

Figure 2.1: An example of a Tree Pattern Query

2.1.2 Review of Prior Work

Early work (e.g., [1, 22]) attacks the Tree Pattern Query Problem by

1. decomposing the pattern tree into a set of (ancestor-descendant and parent-child) edges,
2. matching each of these edges against the target tree individually, and
3. joining the results of edge matchings together.

For example, the pattern tree in Figure 2.1 will be decomposed into a set of three edges $\{B//C, B/A, C/D\}$.

It seems trivial to solve the second sub-problem of matching edge patterns. An intuitive solution would be to iterate through the stream of the target tree corresponding to one node of the edge pattern and for each element of the stream, scan the stream corresponding to the other node of the edge pattern and output solutions. However, this intuitive method may cause unnecessary multiple scans of the second stream as indicated in [1].

We illustrate the problem of unnecessary multiple scans by using the following example. The edge pattern to be matched is A/B. The target tree is shown in Figure 2.2. The two streams would be $T_A = \{A_1, A_2, \ldots, A_n\}$ and $T_B = \{B_1, B_2, \ldots, B_{2n}\}$, assuming the elements of each stream are sorted by their *start* value. Recall that each element of a stream is a 3-tuple $< start, end, level >$, which uniquely represents a node and its position in the target tree.

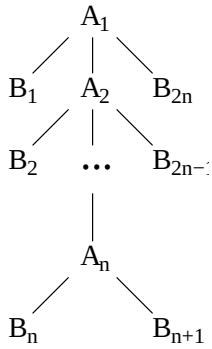

Figure 2.2: Example showing the problem of simple scan algorithm

The first element of T_A is A_1 and it has two children: B_1 at the beginning of T_B and B_{2n} at the very end of T_B. As a result, we need to scan the whole stream T_B once in order to output all solutions corresponding to A_1. Similarly, it is easy to see that for any element in T_A at least half of the elements in T_B need to be scanned. Therefore, the I/O complexity is $O(|T_A| * |T_B|)$, although the output is only of size $O(|T_A| + |T_B|)$.

Recently, by using a stack of size as large as the height of the target tree, Al-Khalifa et al. [1] proposed a family of algorithms which solves the second sub-problem (i.e., matching an edge) optimally. An optimal algorithm for the second sub-problem has both I/O and time complexity

of $O(|\ inputStreams\ | + |\ output\ |)$, where *output* is a set of all matchings corresponding to the edge.

However, as indicated by Bruno et al. [7], the above "decomposing and joining" approach has the disadvantage of producing too many intermediate results.

We illustrate the problem by using the following example. The pattern tree to be matched is $B//C//D$, a path consisting of two ancestor-descendant edges $B//C$ and $C//D$. The target tree is shown in Figure 2.3. It is easy to see that both edge $B//C$ and edge $C//D$ have n^2+1 matchings in the target tree, so the second step of "decomposing and joining" approach will produce intermediate results of $O(|\ inputStream\ |^2)$, but the final output has only one matching $(B \rightarrow B_0, C \rightarrow C_0, D \rightarrow D_0)$.

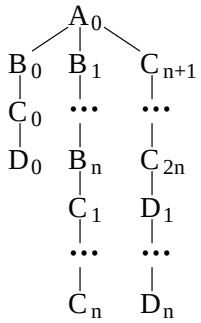

Figure 2.3: Example for decomposing and joining algorithm

Instead of decomposing and joining, Bruno et al. [7] proposed two algorithms *PathStack* and *TwigStack* which try to match a whole path or the whole pattern tree at once by using a chain of stacks. They claim that their PathStack algorithm can solve path queries with worst case I/O and time complexities $O(|\ inputStream\ | + |\ output\ |)$, and their TwigStack algorithm can solve tree pattern queries with only ancestor-descendant edges with the

same worst case complexities.

However, the cost of processing each element of the input streams using PathStack algorithm is $O(\log(|P|))$, and the worst case cost of processing each element of the input streams using TwigStack algorithm is $O(|P|)$, where $|P|$ is the size of the pattern tree, so the time complexity of the PathStack algorithm is actually $O(|\, inputStream\,| \cdot \log(|P|) + |\, output\,|)$, and the time complexity of the TwigStack algorithm is actually $O(|\, inputStream\,| \cdot |P| + |\, output\,|)$. Figure 2.4 shows an example of AD-edge-only Tree Pattern Query. Algorithm *TwigStack* needs to find the minimum of set $\{C_i, D^1, \ldots, D^m\}$ for all $i \in [1, n]$, which will need at least $n \cdot \log m$ comparisons, and hence for this example its time complexity is $O(\log |P| \cdot |\, input\,| + |\, output\,|)$ where P is the pattern tree.

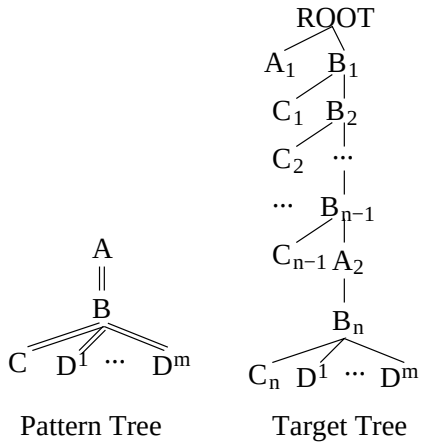

Figure 2.4: Tree Pattern Query Example

Ideally, we would like the time complexity be $O(|\, inputStream\,| + |\, output\,| + |P|)$. In this chapter, we will propose new algorithms which has constant cost of processing each element of the input streams. Further, to the best of our knowledge, neither their algorithms nor any existing algorithm is

optimal for tree pattern queries with both ancestor-descendant edges and parent-child edges, and no existing algorithm is optimal even for tree pattern queries with only parent-child edges. In this chapter, we will also propose new algorithms which are designed to deal with pattern trees with parent-child edges.

2.1.3 Chapter Overview

In this chapter we will study six tree pattern matching related problems, including:

1. Simple AD-edge-only Tree Pattern Query;
2. AD-edge-only Tree Pattern Query;
3. Simple PC-edge-only Tree Pattern Query;
4. PC-edge-only Tree Pattern Query;
5. Simple Tree Pattern Query;
6. Tree Pattern Query.

We will give one algorithm for each of the above problems. Figure 2.5 lists all the algorithms which will be presented in this chapter.

To the best of our knowledge, there is no existing algorithm to solve Tree Pattern Query with parent-child edges, even if the query is parent-child edge only. Although Algorithm *TwigStack* [7] could be used to solve Tree Pattern Query with parent-child edges, it is not *sound*, i.e., it will generate path-matchings which are not a subset of any matching. On the contrary, all the algorithms presented in this chapter are both sound and complete.

Problem	Algorithm	Section
Simple AD-edge-only Tree Pattern Query	Q-MATCH	Sec. 2.2
AD-edge-only Tree Pattern Query	INTEGRAL-MATCH	Sec. 2.3
Simple PC-edge-only Tree Pattern Query	LEVEL-MATCH	Sec. 2.4
PC-edge-only Tree Pattern Query	EXTEND-MATCH	Sec. 2.5
Simple Tree Pattern Query	VERIFY-MATCH	Sec. 2.6
Tree Pattern Query	BUFFER-MATCH	Sec. 2.7

Figure 2.5: Summary of TPQ Algorithms

Both Algorithm *TwigStack* and our Algorithm INTEGRAL-MATCH can solve the AD-edge-only Tree Pattern Query problems. However, our Algorithm INTEGRAL-MATCH is better than Algorithm *TwigStack* in terms of time complexity, and they have the same space and I/O complexity.

To the best of our knowledge, no existing algorithms are designed to solve Simple Tree Pattern Query problem and its variations.

2.2 Simple AD-edge-only Tree Pattern Query

In this section we will present an algorithm Q-MATCH which solves the Simple AD-edge-only Tree Pattern Query problem optimally. We will extend it to solve the AD-edge-only Tree Pattern Query problem in Section 2.3.

2.2.1 Notation and Data Structures

The pattern tree P is a labeled unordered tree with root ROOT. Each node of P is called a query node. For any query node q, function ISLEAF(q) (resp. ISROOT(q)) checks if q is a leaf node (resp. the root), and function CHILDREN(q) (resp. PARENT(q)) returns the children (resp. parent) of q.

The target tree T is stored as data streams of positional notations of target tree nodes. Each positional notation is called an element. The streams are clustered by node labels, and elements of each stream are sorted by their *start* value in ascending order. Each query node q is associated with a virtual stream T_q, which is an image of the data stream with the same label as q. Each virtual stream T_q has a cursor pointing to the current element and initially the cursor is pointing to the first element of the stream. Function CURRENT(T_q) returns the current element of the stream T_q, and function ADVANCE(T_q) moves the cursor of T_q forward by one element. Function EOF(T_q) checks if the cursor is at the end of stream T_q. Note that if two query nodes have the same label, there will be two corresponding cursors and virtual streams.

Recall that each element e of the stream is a 3-tuple of integers $< start, end, level >$. We use $e.start$ (resp. $e.end$ or $e.level$) to refer to the value of *start* (resp. *end* or *level*) of the element e. Elements of a stream correspond to nodes of the target tree, and we will use the same notation to

refer to both.

2.2.2 Algorithm Q-MATCH

Definition 2.1 (*q-match*)**.** Given a pattern tree P, a target tree T, and a query node q in P, let the subtree rooted at q be P_q. A q-match is a node of the target tree, which is mapped to q in some matching(s) from P_q to T.

Theorem 2.1. *Given a pattern tree P, a target tree T, and a query node q in P,*

1. *if q is a leaf node, any element of stream T_q is a q-match;*
2. *if q is an internal node, then an element e_q of T_q is a q-match if and only if for each child node n of q there is an distinct element e_n, which is a n-match and is a descendant (resp. child) of e_q if the edge (q, n) in P is an ancestor-descendant (resp. parent-child) edge.*

Proof. By definition of matching (Def. 1.1) and q-match (Def. 2.1). □

Definition 2.2 (*useful element*)**.** Let q be a non-root query node and p be its parent. Let e be an element of stream T_p. An element e' of stream T_q is useful with respect to e if

1. e' is a q-match, and
2. e' is a descendant (resp. child) of e, if edge (p, q) is an ancestor-descendant (resp. parent-child) edge.

Corollary 2.2. *An element e in T_p is a p-match if and only if for each child q of p, there is an element in T_q which is useful with respect to e.*

Proof. By Theorem 2.1 and definition of useful element (Def. 2.2). □

Based on Theorem 2.1 and Corollary 2.2, we define a recursive function FIND-USEFUL(q) in Algorithm 2.1, which keeps advancing stream T_q until it finds an element which is useful with respect to the current element of stream T_p where p is the parent of q. To check if the current element of T_q is a q-match, by Corollary 2.2 we can just call function FIND-USEFUL(n) on each child n of q, as is in function IS-MATCH(q). Obviously, if p is the root of P, then p-match is exactly root-match, therefore to find all root-matches we can just scan stream T_{ROOT} and for each element e check if it is a root-match, as is in function ROOT-MATCHES. For convenience of analyzing the algorithm we assume all the streams are infinitely long so that we don't need to check EOF whenever we advance streams.

2.2.3 Correctness

Lemma 2.3. *Let q be a non-root query node and p be its parent. During any call of function* FIND-USEFUL*(q), the current element of stream T_p stays the same.*

Proof. Operation ADVANCE(T_q) is called directly only in function FIND-USEFUL(q). Function FIND-USEFUL(q) will recursively call function FIND-USEFUL(n) if q has a child node n, but it won't call function FIND-USEFUL(p) where p is the parent of q, so the streams corresponding to the descendants of q might be advanced indirectly during the call of FIND-USEFUL(q), but the streams corresponding to the ancestors of q won't be advanced. Therefore, the current element of T_p stays the same. □

Let C_q be the set of all calls of function FIND-USEFUL(q). Let $I_q : C_q \rightarrow$

Algorithm 2.1 Algorithm Q-MATCH

```
ROOT-MATCHES(P, T)
1  for (q ∈ P) L_q ← NULL
2  while (TRUE)
3     repeat NOTHING until (IS-MATCH(ROOT) ≠ NO)
4     OUTPUT(CURRENT(T_ROOT))
5     ADVANCE(T_ROOT)

IS-MATCH(q)
1  if (L_q ≠ CURRENT(T_q))
2     L_q ← CURRENT(T_q)
3     for (n ∈ CHILDREN(q))
4           if (FIND-USEFUL(n) = NO)
5              repeat ADVANCE(T_q)
6              until (CURRENT(T_n).start < CURRENT(T_q).end)
7              return NO
8  return OK

FIND-USEFUL(q)
 1  p ← PARENT(q)
 2  while (CURRENT(T_q).start < CURRENT(T_p).start)
 3     ADVANCE(T_q)
 4  if (CURRENT(T_q).start > CURRENT(T_p).end)
 5     return NO

 6  while (L_q ≠ CURRENT(T_q))
 7     L_q ← CURRENT(T_q)
 8     for (n ∈ CHILDREN(q))
 9           if (FIND-USEFUL(n) = NO)
10              repeat ADVANCE(T_q)
11              until (CURRENT(T_n).start < CURRENT(T_q).end)
12              if (CURRENT(T_q).start > CURRENT(T_p).end)
13                    return NO
14              break
15 return OK
```

T_p be the function that assigns to a call to function FIND-USEFUL(q) the current element of T_p during the call, where p is the parent of q. Lemma 2.3 shows that I_q is well-defined. Let c_q and c'_q be two calls in C_q, we say c_q is less than c'_q (i.e., $c_q \prec c'_q$) if c_q happened before c'_q. Let e_p and e'_p be two elements of stream T_p, we say e_p is less than e'_p (i.e., $e_p \prec e'_p$) if e_p is before e'_p in T_p. Lemma 2.5 tells us that the function I_q is strictly increasing.

Corollary 2.4. *Let q be a non-root query node and p be the parent of q. Let e_q (resp. e_p) be an element of stream T_q (resp. T_p). If e_q is compared with e_p in a call of function* FIND-USEFUL*(q) (let it be c_q), then $e_p = I_q(c_q)$.*

Proof. By Lemma 2.3. □

Lemma 2.5. *Let q be a non-root query node and p be the parent of q. Let c_q and c'_q be two calls of function* FIND-USEFUL*(q). If c_q happens before c'_q, then element $I_q(c_q)$ is before $I_q(c'_q)$ in stream T_p.*

Proof. Function FIND-USEFUL(q) could be called in line 9 of function FIND-USEFUL(p) (or line 4 of function IS-MATCH(p)). Note that variable L_p, which is updated in line 7 of function FIND-USEFUL(p) (or line 2 of function IS-MATCH(p)), stores the element $I_q(lc)$ from stream T_p where lc is the last call of function FIND-USEFUL(q). The condition in line 6 of function FIND-USEFUL(p) (or line 1 of function IS-MATCH(p)) ensures that there won't be another call of function FIND-USEFUL(q) unless the current element of T_p is different from L_p. In other words, there must be some advance(s) on T_p between consecutive calls of function FIND-USEFUL(q). Therefore, each call of function FIND-USEFUL(q) corresponds to a different element in T_p, and an

earlier call of FIND-USEFUL(q) corresponds to an earlier element in T_p, since the only operation we used on stream T_p is ADVANCE. □

Corollary 2.6. *Let q be a non-root query node and p be its parent. Let $G_q : V_q.T_p$ be the function that assigns to a call to function* ADVANCE*(T_q) the current element of T_p, where V_q is the set of all calls of function* ADVANCE*(T_q). Then G_q is a monotonically increasing function.*

Proof. By Lemma 2.5 and the fact that ADVANCE(T_q) is called directly only in function FIND-USEFUL(q), and it can be called multiple times. □

Definition 2.3 (*potentially useful*)**.** Let q be a non-root query node and p be its parent. Let e_p be an element of stream T_p. An element of stream T_q is *potentially useful* with respect to e_p if it is useful with respect to e_p or any element after e_p in T_p.

Definition 2.4 (*safe advance*)**.** Let q be a non-root query node and p be its parent. Operation ADVANCE(T_q) is safe if the current element of stream T_q is not potentially useful with respect to the current element of stream T_p.

Lemma 2.7. *Operation* ADVANCE*(T_q) called in line 3 of function* FIND-USEFUL*(q) is always safe.*

Proof. Let p be the parent of q. Let the current element of stream T_q and T_p be e_q and e_p, respectively. Let e'_p be an arbitrary element after e_p in stream T_p. The condition in line 2 ensures that $e_q.start < e_p.start$, and hence $e_q.start < e'_p.start$, since elements in T_p are sorted by their *start* value in ascending order. Therefore, by Proposition 1.1 e_q can not be a descendant of

e_p or e'_p, so e_q is not potentially useful with respect to e_p, and thus function call ADVANCE(T_q) in line 3 of function FIND-USEFUL(q) is safe. □

Definition 2.5. Given a query node q, we say function FIND-USEFUL(q) is safe if all ADVANCE operations called in it are safe.

Corollary 2.8. *If q is a leaf node, function* FIND-USEFUL*(q) is safe.*

Proof. If q is a leaf node, the only place where operation ADVANCE is called is line 3. By Lemma 2.7, ADVANCE(T_q) called in line 3 is always safe. Therefore, function FIND-USEFUL(q) is safe. □

Lemma 2.9. *Let q be an internal node. If for each child n of q function* FIND-USEFUL*(n) is safe, then operation* ADVANCE*(T_q) called in line 10 of function* FIND-USEFUL*(q) is safe.*

Proof. Let v_q be an arbitrary operation ADVANCE(T_q) called in line 10 of function FIND-USEFUL(q). Let the current element of stream T_q (resp. T_n) right before v_q be e_q (resp. e_n). Let e'_n be an arbitrary element after e_n in stream T_n. Condition in line 9 and line 11 ensures that $e_n.start > e_q.end$, and hence $e'_n.start > e_q.end$, since elements in T_n are sorted by their *start* value in ascending order. Therefore, by Proposition 1.1 e_n or e'_n can not be a descendant of e_q. In other words, e_n or any element after e_n is not useful with respect to e_q.

Let e''_n be an arbitrary element before e_n in stream T_n. Since the cursor of T_n has been moved forward from e''_n to e_n, there must be at least one call of operation ADVANCE(T_n). Let v_n be the call of function ADVANCE(T_n) in which the cursor of T_n is moved away from e''_n forward, and let e'_q be $G_n(v_n)$.

By Corollary 2.6 we know that $e'_q = e_q$ or $e'_q \prec e_q$. Since FIND-USEFUL(n) is safe, v_n is safe, and thus e''_n is not useful with respect to e'_q or any element after e'_q, and thus not useful with respect to e_q.

In conclusion, no element in T_n is useful with respect to e_q. By Corollary 2.2, e_q can not be a q-match, and hence it is not useful with respect to any element in stream $T_{\text{PARENT}(q)}$. Therefore, operation ADVANCE(T_q) called in line 10 of function FIND-USEFUL(q) is safe. □

Corollary 2.10. *Let q be an internal node. If for each child n of q function* FIND-USEFUL*(n) is safe, then function* FIND-USEFUL*(q) is safe.*

Proof. By Lemma 2.7 and Lemma 2.9. □

Corollary 2.11. *Function* FIND-USEFUL*(q) is safe for all query nodes q.*

Proof. By mathematical induction using Corollary 2.8 and 2.10. □

Definition 2.6. Function FIND-USEFUL(q) is correct if

1. it returns OK when the current element of stream T_q is useful with respect to the current element of $T_{\text{PARENT}(q)}$;
2. it returns NO when no element in T_q is useful with respect to the current element of $T_{\text{PARENT}(q)}$.

Lemma 2.12. *If function* FIND-USEFUL*(q) returns* NO*, then there is no element in T_q which is useful with respect to the current element of $T_{\text{PARENT}(q)}$.*

Proof. Suppose function FIND-USEFUL(q) is to return NO, which means that it is executing line 5 or line 13. Let the current element of stream T_q and

$T_{\text{PARENT}(q)}$ be e_q and e_p, respectively. Let e'_q be an arbitrary element after e_q in stream T_q. The conditions in line 4 and line 12 ensure that $e_q.start > e_p.end$, and hence $e'_q.start > e_p.end$, since elements in a stream are sorted by their *start* value in ascending order. Therefore, by Proposition 1.1 e_q or e'_q can not be a descendant of e_p. In other words, e_q or any element after e_q is not useful with respect to e.

Let e''_q be an arbitrary element before e_q in stream T_q. Since the cursor of T_n has been moved forward from e''_q to e_q, there must be at least one call of operation ADVANCE. Let v_q be the call of function ADVANCE(T_q) in which the cursor of T_q is moved away from e''_q forward, and let e'_p be $G_q(v_q)$. By Corollary 2.6 we know that $e'_p = e_p$ or $e'_p \prec e_p$. Since function FIND-USEFUL(q) is safe, every call of operation ADVANCE(T_q) in it is safe. Therefore, e''_q is not useful with respect to e'_p or any element after e'_p, and thus not useful with respect to e_p.

In conclusion, no element in T_q is useful with respect to e_p. □

Lemma 2.13. *Let q be a leaf node. If function* FIND-USEFUL*(q) returns* OK*, then the current element of stream T_q is useful with respect to the current element of $T_{\text{PARENT}(q)}$.*

Proof. Suppose function FIND-USEFUL(q) is to return OK, which means that it is executing line 15. Let the current element of stream T_q and $T_{\text{PARENT}(q)}$ be e_q and e_p, respectively. The condition in line 2 ensures that $e_q.start > e_p.start$, and the condition in line 4 ensures that $e_q.start < e_p.end$, therefore by Proposition 1.1 e_q is a descendant of e_p. Since q is a leaf node, every element in T_q is a q-match, thus, e_q is a useful element with respect to e_p. □

Corollary 2.14. *If q is a leaf node, function* FIND-USEFUL*(q) is correct.*

Proof. By Lemma 2.12 and Lemma 2.13. □

Lemma 2.15. *Let q be an internal node. Function* FIND-USEFUL*(q) is correct if for each child n of q, function* FIND-USEFUL(n) *is correct.*

Proof. Suppose function FIND-USEFUL(q) is to return OK, which means that it is executing line 15. Let the current element of stream T_q and $T_{\text{PARENT}(q)}$ be e_q and e_p, respectively. The condition in line 2 ensures that $e_q.start > e_p.start$, and the condition in line 4 and line 12 ensures that $e_q.start < e_p.end$, therefore by Proposition 1.1 e_q is a descendant of e_p.

The function FIND-USEFUL(q) has to finish executing the **while** loop from line 6 to line 14 before it executes line 15, which requires that for each child n of q, FIND-USEFUL(n) returns OK. Since function FIND-USEFUL(n) is correct, this means that for each child n of q, the current element of T_n is useful respect to e_q, and hence by Corollary 2.2 e_q is a q-match. Therefore, e_q is useful with respect to e_p.

This proves that the case where function FIND-USEFUL(q) returning OK is correct. Since the case where function FIND-USEFUL(q) returning NO has been proved correct in Lemma 2.12, we conclude that function FIND-USEFUL(q) is correct. □

Theorem 2.16. *Function* FIND-USEFUL *is correct.*

Proof. By mathematical induction using Corollary 2.14 and Lemma 2.15. □

Lemma 2.17. *Function* IS-MATCH*(q) advances stream T_q only if the current element of T_q is not a q-match.*

Proof. Line 5 is the only place in function IS-MATCH(q) where T_q is advanced. Suppose the function is to execute line 5, let the current element of stream T_q and T_n be e_q and e_n, respectively. Let e'_n be an arbitrary element after e_n in stream T_n. The conditions in line 4 and line 6 ensure that $e_n.start > e_q.end$, and hence $e'_n.start > e_q.end$, since elements in a stream are sorted by their *start* value in ascending order. Therefore, by Proposition 1.1 e_n or e'_n can not be a descendant of e_q. In other words, e_n or any element after e_n is not useful with respect to e_q.

Let e''_n be an arbitrary element before e_n in stream T_n. Since the cursor of T_n has been moved forward from e''_n to e_n, there must be at least one call of operation ADVANCE(n). Let v_n be the call of function ADVANCE(T_n) in which the cursor of T_n is moved away from e''_n forward, and let e'_q be $G_n(v_n)$. By Corollary 2.6 we know that $e'_q = e_q$ or $e'_q \prec e_q$. Since FIND-USEFUL(n) is safe, e''_n is not useful with respect to e'_q or any element after e'_q, and thus not useful with respect to e_q.

In conclusion, no element in T_n is useful with respect to e_q. By Corollary 2.2, e_q can not be a q-match. □

Corollary 2.18. *Function* IS-MATCH*(q) returns* NO *if the current element of T_q right before it is called is not a q-match.*

Proof. By Lemma 2.17. □

Lemma 2.19. *Function* IS-MATCH*(q) returns* OK *only if the current element of T_q is a q-match.*

Proof. Function FIND-MATCH(q) returns OK only if for each child n of q, FIND-USEFUL(n) returns OK, in other words, the current element of T_n is useful with respect to the current element of T_q. Thus, by Corollary 2.2, the current element of T_q is a q-match. □

Theorem 2.20 (Correctness)**.** *Algorithm* Q-MATCH *will output all and only root-matches of the given Simple AD-edge-only Tree Pattern Query problem.*

Proof. By Lemma 2.17 and Lemma 2.19. □

2.2.4 Complexity

It is easy to verify that the space complexity (memory usage) of Algorithm Q-MATCH is $O(|P|)$ and its I/O complexity is $O(\sum_{q\in P}|T_q|)$ reading and $O(|R|)$ writing, where R is the set of all root-matches.

Atomic operations used in Algorithm Q-MATCH are comparison and ADVANCE. The cost of ADVANCE operations is clearly $O(\sum_{q\in P}|T_q|)$. In Lemma 2.22, we will prove that comparisons can be amortized to stream elements and root-matches, and the cost of comparisons is $O(\sum_{q\in P}|T_q| + |P|\cdot|R|)$. Thus, we have the following theorem (Thm. 2.21) regarding to the complexities of Algorithm Q-MATCH.

Theorem 2.21 (Complexity)**.** *Given pattern tree P and target tree T. Let R be the set of all root-matches. Algorithm* Q-MATCH*'s time complexity is $O(\sum_{q\in P}|T_q|+|P|\cdot|R|)$, its space complexity is $O(|P|)$, and its I/O complexity is $O(\sum_{q\in P}|T_q|)$ reading and $O(|R|)$ writing.*

Lemma 2.22. *Given pattern tree P and target tree T. Let the set of all root-matches be R. The cost of comparisons of Algorithm* Q-MATCH *is* $O(\sum_{q \in P} |T_q| + |P| \cdot |R|)$.

Proof. Let q be a non-root query node and p be the parent of q. Let e_q (resp. e_p) be an element of stream T_q (resp. T_p). The comparison between e_q and e_p can only happen in function FIND-USEFUL(q) and function FIND-USEFUL(p) (or function IS-MATCH(p) if p is the root).

First we show that the cost of all comparisons between any e_q and e_p, which happen in function FIND-USEFUL(p) (or function IS-MATCH(p)), is $O(|T_p|)$. We discuss the case where p is not the root first. The only place in function FIND-USEFUL(p) where e_q and e_p can be compared is line 11. It is easy to see that each comparison happened in line 11 has a corresponding ADVANCE(T_p) operation in line 10. Since the cost of all ADVANCE(T_p) operations is $O(|T_p|)$, the cost of all comparisons between any e_q and e_p, which happen in function FIND-USEFUL(p), is also $O(|T_p|)$. The analysis is similar if p is the root.

Suppose there are two calls of function FIND-USEFUL(q) in which e_q and e_p are compared. Let them be c_q and c'_q. Thus, $e_p = I_q(c_q) = I_q(c'_q)$. Since I_q is strictly increasing, $I_q(c_q) = I_q(c'_q)$ implies that $c_q = c'_q$. This implies that there is at most one call of function FIND-USEFUL(q) in which e_q and e_p are compared.

Suppose e_p and e_q are compared in a call of function FIND-USEFUL(q). Let it be c_q. Line 2, line 4 and line 12 are the only three places where e_q and e_p are compared. Consecutive execution of line 2 are interleaved by

an ADVANCE operation on stream T_q, so line 2 will be executed at most once in c_q for each pair of e_q and e_p. It is easy to see that line 4 will be executed at most once in c_q. Condition in line 6 ensures that each time line 12 is executed, the current element of T_q is different, so line 12 will be executed at most once for each pair of e_q and e_p. Therefore, e_p and e_q will be compared at most three times [4] during c_q.

We discuss the amortization of the comparisons between e_q and e_p in c_q in the following three cases according to the relative position of e_q and e_p.

Case 1. If $e_p.end < e_q.start$, c_q will return NO. Suppose c_q is called by c_p. It is easy to see that after c_q returns NO, c_p will immediately call ADVANCE(T_p), whether c_p is a call of function FIND-USEFUL(p) or function IS-MATCH(p). We can amortize the comparisons between e_p and e_q in this case to e_p, so that only a constant number of comparisons are amortized to each element of T_p.

Case 2. If $e_p.start > e_q.start$, line 2 ensures that T_q will be advanced immediately, so we can amortize the comparisons between e_p and e_q in this case to e_q, so that only a constant number of comparisons are amortized to each element of T_q.

Case 3. Otherwise, e_p is an ancestor of e_q. Suppose there are m elements of T_p which are both ancestors of e_q and are compared with e_q in a call of function FIND-USEFUL(q). Let them be $\{e_p^i\}$ $(1 \leq i \leq m)$ and the call of function FIND-USEFUL(q) corresponding to e_p^i be c_q^i. Assume without loss of generality that e_p^i is an ancestor of e_p^{i+1} $(1 \leq i < m)$. By Corollary 2.29 each

[4] More careful analysis can prove that e_p and e_q will be compared at most twice in c_q.

element $I_q^+(c_q^i)$ $(1 \leq i < m)$ is a distinct root-match (function I_q^+ is defined in Def. 2.7). Therefore, we can amortize the comparisons between e_q and e_p^i $(1 \leq i < m)$ to root-match $I_q^+(c_q^i)$, and amortize the comparisons between e_q and e_p^m to e_q. Let R_{e_q} be $\{I_q^+(c_q^i)\}$ $(1 \leq i < m)$. Lemma 2.30 shows that R_{e_q} is disjoint from $R_{e'_q}$ if $e'_q \in T_q$ and $e'_q \neq e_q$. In other words, for each q only a constant number of comparisons are amortized to each root-match. Therefore, the number of comparisons amortized to each root-match is at most $|P|$.

In conclusion, every comparison in Algorithm Q-MATCH can be amortized to some element or some root-match. Since the number of comparisons amortized to each element is constant and the number of comparisons amortized to each root-match is at most $|P|$, the cost of comparisons of Algorithm Q-MATCH is $O(\sum_{q \in P} |T_q| + |P| \cdot |R|)$, where R is the set of root-matches. □

Lemma 2.23. *Let q be a non-root internal query node and p be the parent of q. Let c_q be a call of function* FIND-USEFUL*(q). If c_q called function* FIND-USEFUL*(n) for some child n (let the call be c_n), then $I_q(c_q)$ is an ancestor of $I_n(c_n)$.*

Proof. Line 2 of function FIND-USEFUL(q) ensures that $I_n(c_n).start > I_q(c_q).start$. Line 4 and line 12 of function FIND-USEFUL(q) ensure that $I_n(c_n).start < I_q(c_q).end$. By Proposition 1.1, $I_q(c_q)$ is an ancestor of $I_n(c_n)$. □

Lemma 2.24. *Let q be a non-root query node and p be the parent of q. Let c_q and c'_q be two calls of function* FIND-USEFUL*(q). If $I_q(c_q)$ is an ancestor of $I_q(c'_q)$, then $I_q(c_q)$ is a p-match.*

Proof. Let $e_p = I_q(c_q)$ and $e'_p = I_q(c'_q)$. Let the children of p be $\{q_i\}$ $(1 \leq i \leq |\text{CHILDREN}(p)|)$. Let c_{q_i} $(1 \leq i \leq k)$ be a set of calls of function FIND-USEFUL(q_i) such that $I_q(c_{q_i}) = e_p$. We know $k \geq 1$, since $I_q(c_q) = e_p$. It is easy to see that $c_{q_{i+1}}$ $(1 \leq i < |\text{CHILDREN}(p)|)$ is called if and only if c_{q_i} returns OK, so

1. either c_{q_i} $(1 \leq i < k)$ returns OK but c_{q_k} returns NO, or
2. c_{q_i} $(1 \leq i \leq k|)$ returns OK and $k = |\text{CHILDREN}(p)|$.

Suppose c_{q_k} returns NO. Let the current element of T_{q_k} after c_{q_k} returns be e_{q_k}. Then $e_{q_k}.start > e_p.end$, and hence $e_{q_k}.start > e'_p.end$. Thus, e_{q_k} is after e_p and e'_p. Let c_p be the call of function FIND-USEFUL(p) (or the call of function IS-MATCH(p)) which called c_q. Since c_{q_k} returns NO, c_p will execute the **repeat-until** loop from line 10 to line 11 of function FIND-USEFUL(p) (or from line 5 to line 6 of function FIND-MATCH(p)). Let the current element of T_p after the loop finishes be e''_p, we know that $e_{q_k}.start < e''_p.end$, and hence e''_p is either an ancestor of e_{q_k} or after e_{q_k}. Therefore, e''_p is after e_p and e'_p. Since the stream T_p has been advanced to an element after e'_p, there won't exist any call of function FIND-USEFUL(q) such that $e'_p = I_q(c'_q)$. Contradiction! Thus, c_{q_i} $(1 \leq i \leq k|)$ returns OK and $k = |\text{CHILDREN}(p)|$. Therefore, by Corollary 2.2 e_p is a p-match. □

Lemma 2.25. *Let q be a non-root internal query node, p be the parent of q, and n be one of the children of q. Let c_n and c'_n be two calls of function* FIND-USEFUL*(n). Suppose c_q (resp. c'_q) is the call of function* FIND-USEFUL*(q) which called c_n (resp. c'_n). If $I_n(c_n)$ is an ancestor of $I_n(c'_n)$, then $I_q(c_q)$ is an ancestor of $I_q(c'_q)$.*

Proof. By Lemma 2.5 function I_n and I_q are strictly increasing, so $I_n(c_n)$ is an ancestor of $I_n(c'_n)$ (i.e., $I_n(c_n) \prec I_n(c'_n)$), implies that $c_n \prec c'_n$. Therefore, $c_q = c'_q$ or $c_q \prec c'_q$, and hence $I_q(c_q) = I_q(c'_q)$ or $I_q(c_q) \prec I_q(c'_q)$.

By Lemma 2.24, $I_n(c_n)$ is a q-match. By Lemma 2.23, $I_q(c_q)$ is an ancestor of $I_n(c_n)$. Therefore, $I_n(c_n)$ is useful with respect to $I_q(c_q)$. Note that the cursor of T_q has been moved away from $I_n(c_n)$ forward to $I_n(c'_n)$. Since function FIND-USEFUL(q) is safe, the cursor of T_q will be moved away from $I_n(c_n)$ only if $I_n(c_n)$ is not potentially useful with respect to the current element of T_p. But we know $I_n(c_n)$ is useful with respect to $I_q(c_q)$, so stream T_p must have been advanced. Therefore, $I_q(c_q) \neq I_q(c'_q)$ and thus $I_q(c_q) \prec I_q(c'_q)$.

By Lemma 2.23, $I_q(c_q)$ is an ancestor of $I_n(c_n)$ and $I_q(c'_q)$ is an ancestor of $I_n(c'_n)$. Since $I_n(c_n)$ is an ancestor of $I_n(c'_n)$, $I_q(c_q)$ is also an ancestor of $I_n(c'_n)$. Since both $I_q(c_q)$ and $I_q(c'_q)$ are ancestors of $I_n(c'_n)$ and $I_q(c_q) \prec I_q(c'_q)$, $I_q(c_q)$ is an ancestor of $I_q(c'_q)$. □

The following notations will be used for Lemma 2.26, 2.27, 2.28 and Definition 2.7. Let q be a non-root query node. Let $q^1, \ldots, q^h$ be query nodes such that $q^1 = q$, $q^h = \text{ROOT}$, and $q^{i+1} = \text{PARENT}(q^i)$ where $1 \leq i < h$. Let c_{q^i} $(1 \leq i < h)$ be a call of function FIND-USEFUL(q^i), and $c_{q^{i-1}}$ $(1 < i < h)$ is called by c_{q^i}.

Lemma 2.26. *If $I_q(c_q)$ is an ancestor of $I_q(c'_q)$, then $I_q(c_{q^i})$ is an ancestor of $I_q(c'_{q^i})$ $(1 \leq i < h)$.*

Proof. By mathematical induction using Lemma 2.25. □

Lemma 2.27. *If $I_q(c_q)$ is an ancestor of $I_q(c'_q)$, then $I_q(c_{q^i})$ is a q^i-match ($1 \leq i < h$).*

Proof. By Lemma 2.24 and Lemma 2.26. □

Lemma 2.28. *If $I_q(c_q)$ is an ancestor of $I_q(c'_q)$, then $I_q(c_{q^i})$ is useful with respect to $I_q(c_{q^{i+1}})$ ($1 \leq i < h$).*

Proof. By Lemma 2.23 and Lemma 2.27. □

Definition 2.7 (I_q^+)**.** Function $I_q^+ : C_q \rightarrow T_{\text{ROOT}}$ is defined as $I_q^+(c_q) = I_{q^{h-1}}(c_{q^{h-1}})$.

Corollary 2.29. *Let q be a non-root query node. Let c_q and c'_q be two calls of function* FIND-USEFUL*(q). If $I_q(c_q)$ is an ancestor of $I_q(c'_q)$, then $I_q^+(c_q)$ is a root-match, and is an ancestor of $I_q^+(c'_q)$.*

Proof. By Lemma 2.26 and Lemma 2.27. □

Lemma 2.30. *Let q be a non-root query node and p be the parent of q. Let e_q be an element of T_q. Let e_p and e'_p be two elements of T_p such that e_p is an ancestor of e'_p and e'_p is an ancestor of e_q. Suppose e_q and e_p (resp. e'_p) are compared in a call of function* FIND-USEFUL*(q), let it be c_q (resp. c'_q). Similarly, let t_q be an different element from e_q of T_q. Let t_p and t'_p be two elements of T_p such that t_p is an ancestor of t'_p and t'_p is an ancestor of t_q. Suppose t_q and t_p (resp. t'_p) are compared in a call of function* FIND-USEFUL*(q), let it be d_q (resp. d'_q). Then, $I_q^+(c_q)$ is different from $I_q^+(d_q)$.*

Proof. Let $q^1, \ldots, q^h$ be query nodes such that $q^1 = q$, $q^h = \text{ROOT}$, and $q^{i+1} = \text{PARENT}(q^i)$ where $1 \leq i < h$. Let c_{q^i} ($1 \leq i < h$) be calls of function

FIND-USEFUL(q^i), such that $c_{q^1} = c_q$ and $c_{q^{i-1}}$ $(1 < i < h)$ is called by c_{q^i}. Let d_{q^i} $(1 \leq i < h)$ be calls of function FIND-USEFUL(q^i), such that $d_{q^1} = d_q$ and $d_{q^{i-1}}$ $(1 < i < h)$ is called by d_{q^i}.

Suppose e_p and t_p are the same element, then c_q and d_q are the same since function I_q is strictly increasing. Therefore, both e_q and t_q are compared with e_p in c_q, thus stream T_q is advanced during c_q. Assume without loss of generality $e_q \prec t_q$, we know that e_q will not compare with other elements after the return of c_q, but this contradicts that e_q is compared with e'_p in c'_q. Therefore, e_p (i.e., $I_q(c_q)$) and t_p (i.e., $I_q(d_q)$) are different.

Next we prove that if $I_q(c_{q^i})$ and $I_q(d_{q^i})$ are different, then $I_q(c_{q^{i+1}})$ and $I_q(d_{q^{i+1}})$ are different. By Lemma 2.28 $I_q(c_{q^i})$ is useful with respect to $I_q(c_{q^{i+1}})$ $(1 \leq i < h)$. Suppose we are at the point of the computation where $I_q(c_{q^i})$ and $I_q(c_{q^{i+1}})$ are the current element of stream T_{q^i} and $T_{q^{i+1}}$, respectively. Since all ADVANCE operations are safe, T_{q^i} will not be advanced if the current element of $T_{q^{i+1}}$ is still $I_q(c_{q^{i+1}})$. Therefore, if $I_q(c_{q^i})$ is different from $I_q(d_{q^i})$, which means that T_{q^i} has been advanced, then $T_{q^{i+1}}$ should also have been advanced, and thus $I_q(c_{q^{i+1}})$ is different from $I_q(d_{q^{i+1}})$.

Since we already know that $I_q(c_q)$ and $I_q(d_q)$ (i.e., $I_q(c_{q^1})$ and $I_q(d_{q^1})$) are different, by mathematical induction, $I_q^+(c_q)$ and $I_q^+(d_q)$ are different. □

2.3 AD-edge-only Tree Pattern Query

In this section we will present an algorithm INTEGRAL-MATCH to solve the AD-edge-only Tree Pattern Query problem, which is based on Algorithm Q-MATCH in Section 2.2.

2.3.1 Notation and Data Structures

We will use a chain of stacks to encode intermediate results compactly, which is first used in [1, 7]. Each node q of the pattern tree P is associated with a stack S_q. Each entry en of stack S_q consists of a pair $< e, pt >$, where e is an element from stream T_q and pt is a pointer to an entry in stack $S_{\text{PARENT}(q)}$. In case q is the root, pt is NULL. We will use $en.e$ (resp. $en.pt$) to refer to the element (resp. pointer) in entry en. For convenience, we abbreviate $en.e.*$ (e.g. $en.e.start$) as $en \Rightarrow *$ (e.g. $en \Rightarrow start$).

Operation PUSH(S_q, e, pt) pushes pair $< e, pt >$ into the top entry of the stack S_q. Operation POP(S_q) pops the top entry out of the stack S_q. Operation TOP(S_q) returns a pointer to the top entry of the stack S_q. Let en be an entry of stack S_q, operation PREV(en) returns the entry below en in S_q, i.e., the entry which was pushed into S_q right before en. If en is already the bottom entry of S_q, PREV(en) will return NULL. All the above operations should have constant costs if the stacks are implemented using linked lists.

Figure 2.6 shows an example of the use of stacks. The intermediate results encoded in the stacks are $\{(D \rightarrow D_1, C \rightarrow C_2, B \rightarrow B_2), (D \rightarrow D_1, C \rightarrow C_2, B \rightarrow B_1), (D \rightarrow D_1, C \rightarrow C_1, B \rightarrow B_1)\}$, which can be gener-

ated by calling function OUTPUT-PATHS with parameters D, TOP(S_D), and NULL. Function OUTPUT-PATHS is defined in Algorithm 2.2.

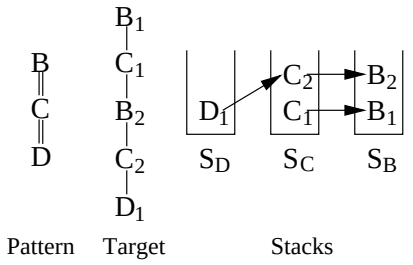

Figure 2.6: Example of using stacks

2.3.2 Algorithm INTEGRAL-MATCH

Recall that a matching is a function from query nodes to stream elements. A matching can also be viewed as a set of pairs $< q, e >$, where q is a query node and e is an element of stream T_q. Note that not all combinations of pairs of query nodes and stream elements can be used to make up a matching.

Definition 2.8 (integral element)**.** Given a pattern tree P, a target tree T, and a query node q, an element e from stream T_q is *integral* if there exists a matching M from P to T such that $< q, e > \in M$.

Theorem 2.31. *All root-matches are integral. Let q be an non-root query node and p be the parent of q, an element e_q from stream T_q is integral iff there exists an element e_p from stream T_p such that e_p is integral and e_q is useful with respect to e_p.*

Proof. By definition of matching (Def. 1.1), useful element (Def. 2.2) and integral element (Def. 2.8). □

It is easy to see that in order to generate all matchings it is necessary to find all integral elements. Based on Theorem 2.31 we define function IS-INTEGRAL(q) in Algorithm 2.2, which checks for any non-root query node q if the current element e_q of stream T_q is integral. If q is the root, we can just use function IS-MATCH(q) to check if e_q is integral.

Once we find that e_q is integral, it will be pushed into stack S_q with a pointer pointing to its closest ancestor in stack $S_{\text{PARENT}(q)}$ by function COPY-TO-STACK(q). Besides pushing e_q into S_q, Function COPY-TO-STACK(q) will also pop any elements in S_q which are not ancestors of e_q, such that all elements below e_q in S_q are ancestors of e_q.

To avoid scanning the streams multiple times, we embed the process of finding integral elements in the process of finding root-matches (line 3 of function FIND-USEFUL(q)). The process of finding integral element will be done in a top-down fashion from root to all leaves. Once an integral element for a leaf query node q is found, function OUTPUT-PATHS will be called on q to output the path-matchings corresponding to the path from the root to q (i.e., intermediate results stored in the chain of stacks along the path from the root to q), as the example in Figure 2.6 shows.

Definition 2.9 (*path-matching*)**.** Let P be a pattern tree, T be a target tree, and p be an arbitrary root-to-leaf path in P. A matching from p to T is called a path-matching.

Definition 2.10 ($\oplus$)**.** Let M and M' be two matchings. If there exists a query node q in the intersection of the domains of M and M' such that $M(q) \neq M'(q)$, then $M \oplus M' = \emptyset$ (i.e., the empty set); otherwise, $M \oplus M' = M \cup M'$ (i.e., the union of M and M').

Lemma 2.32. *Let P be a pattern tree and T be a target tree. Let $\{p_i\}$ $(1 \leq i \leq n)$ be the set of all root-to-leaf paths of P. Let pm_i $(1 \leq i \leq n)$ be a path-matching corresponding to path p_i. Let $M = pm_1 \oplus \cdots \oplus pm_n$. M is either empty or is a matching from P to T.*

Proof. By Definition of matching (Def. 1.1), path-matching (Def. 2.9), and operation $\oplus$ (Def. 2.10). □

Definition 2.11 (*merge*)**.** Let P be a pattern tree and T be a target tree. Let $\{p_1, \ldots, p_n\}$ be the set of all root-to-leaf paths of P. Let PM_i $(1 \leq i \leq n)$ be a set of path-matchings corresponding to path p_i. The result of merge operation on $\{PM_i\}$ $(1 \leq i \leq n)$ is

$$\{\ (pm_1 \oplus \cdots \oplus pm_n) \mid pm_i \in PM_i\ (1 \leq i \leq n)\}.$$

By Theorem 2.48 to generate all matchings we can just perform a merge operation on the sets of path-matchings found. Furthermore, Theorem 2.41 shows that all path-matching found by Algorithm 2.2 is necessary. Note that since the outputs of path-matchings are initiated at leaf query nodes, naturally the path-matchings are sorted in leaf-to-root order. However, in order to perform the merge operation more efficiently, it is desired to have the path-matchings sorted in root-to-leaf order. Details of how to achieve

the root-to-leaf order can be found in [1, 7]. We omit them and the merging method in this thesis.

Algorithm INTEGRAL-MATCH can be modified to use buffers so that at the end of its execution all matchings are already stored compactly in a set of linked buffers and there is no need to output intermediate path-matchings. The use of buffers will be explained in Section 2.7.

2.3.3 Correctness

Lemma 2.33. *Function* FIND-USEFUL(q) *and function* IS-MATCH(q) *in Algorithm* INTEGRAL-MATCH *are still correct.*

Proof. It is easy to see that for leaf query nodes, function FIND-USEFUL(q) and function IS-MATCH(q) are still correct.

Let q be an internal query node. Since there is no change to function IS-MATCH(q), it is correct if for any children n of q function FIND-USEFUL(n) is correct. The only change to function FIND-USEFUL(q) is the addition of line 3 in Algorithm 2.2. The statement and functions called in this line do not change any local variables used in Algorithm 2.1. Some element from stream T_q may be advanced in functions called in this line, which are not a q-match and hence not useful with respect to any element, if for any children n of q function IS-MATCH(n) are correct.

Therefore, we can prove that function FIND-USEFUL(q) and function IS-MATCH(q) are still correct by induction. □

Lemma 2.34. *Let q be a non-root query node and p be the parent of q. Let en_q be an entry in S_q, en_p be a entry in S_p such that $en_p = en_q.pt$, and en'_p*

Algorithm 2.2 Algorithm INTEGRAL-MATCH

```
PATH-MATCHINGS(P, T)
1  for (q ∈ P) {L_q, PT_q} ← NULL
2  while (TRUE)
3      repeat NOTHING until (IS-MATCH(ROOT) ≠ NO)
4      COPY-TO-STACK(ROOT)
5      ADVANCE(T_ROOT)

IS-MATCH(q)
1  if (L_q ≠ CURRENT(T_q))
2      L_q ← CURRENT(T_q)
3      for (n ∈ CHILDREN(q))
4          if (FIND-USEFUL(n) = NO)
5              repeat ADVANCE(T_q)
6              until (CURRENT(T_n).start < CURRENT(T_q).end)
7              return NO
8  return OK

FIND-USEFUL(q)
 1  p ← PARENT(q)
 2  while (CURRENT(T_q).start < CURRENT(T_p).start)
 3      if (IS-INTEGRAL(q) = OK) COPY-TO-STACK(q)
 4      ADVANCE(T_q)
 5  if (CURRENT(T_q).start > CURRENT(T_p).end)
 6      return NO

 7  while (L_q ≠ CURRENT(T_q))
 8      L_q ← CURRENT(T_q)
 9      for (n ∈ CHILDREN(q))
10          if (FIND-USEFUL(n) = NO)
11              repeat ADVANCE(T_q)
12              until (CURRENT(T_n).start < CURRENT(T_q).end)
13              if (CURRENT(T_q).start > CURRENT(T_p).end)
14                  return NO
15              break
16 return OK
```

Algorithm 2.2 Algorithm INTEGRAL-MATCH *(continued)*

```
IS-INTEGRAL(q)
1  if (PT_q = NULL) return NO

2  while (CURRENT(T_q).start > PT_q ⇒ end)
3      PT_q ← PREV(PT_q)
4      if (PT_q = NULL) return NO

5  if (IS-MATCH(q) = NO)
6      return NO

7  return OK

COPY-TO-STACK(q)
1  while (|S_q| ≠ 0 ∧ CURRENT(T_q).start > TOP(S_q) ⇒ end)
2      POP(S_q)

3  PUSH(S_q, CURRENT(T_q), PT_q)

4  for (n ∈ CHILDREN(q))
5      PT_n ← TOP(S_q)

6  if (IS-LEAF(q))
7      OUTPUT-PATHS(q, TOP(S_q), NULL)

OUTPUT-PATHS(q, en, path)
1  path ← path + < q, en.e >

2  if (IS-ROOT(q)) OUTPUT(path)

3  pe ← en.pt
4  while (pe ≠ NULL)
5      OUTPUT-PATHS(PARENT(q), pe, path)
6      pe ← PREV(pe)
```

be an arbitrary entry above en_p in S_p. Then, $en_p.e$ is an ancestor of $en_q.e$, but $en'_p.e$ is not.

Proof. Let $e_p = en_p.e$, $e'_p = en'_p.e$, and $e_q = en_q.e$. Condition in line 2 of function IS-INTEGRAL(q) ensures that $e_p.end > e_q.start > e'_p.end$. By Proposition 1.1, e'_p is not an ancestor of e_q.

Note that en_p is in stack S_p implies that function IS-MATCH(p) should be called when e_p was the current element of stream T_p (Let the call be c_p), and c_p returns OK. Since c_p returns OK, c_p should have called function FIND-USEFUL(q) (let it be c_q), and c_q returns OK. By Theorem 2.16 and Definition 2.6, the current element e'_q of stream T_q, at the time when c_q returns OK, is useful with respect to e_p. Thus e_p is an ancestor of e'_q, and hence $e'_q.start > e_p.start$. It is easy to see that $e'_q = e_q$ or $e'_q \prec e_q$, so $e_q.start > e'_q.start > e_p.start$. Since we already know $e_q.start < e_p.end$, by Proposition 1.1 e_q is a descendant of e_p. □

Lemma 2.35. *All elements pushed into the stacks are integral.*

Proof. Let q be a non-root query node and p be the parent of q. Suppose all elements pushed into stack S_p are integral, we now prove that all elements pushed into stack S_q are integral. An element e_q from stream T_q will be pushed into stack S_q only if function IS-INTEGRAL(q) is called when e_q is the current element of T_q, and the call returns OK. Condition in line 5 ensures that e_q is a q-match, and by Lemma 2.34 there is an element e_p in stack S_p such that e_q is a descendant of e_p. Therefore, e_q is useful with respect to e_p. Since e_p is integral, by Theorem 2.31, e_q is integral.

An element will be pushed into stack S_{ROOT} only if it is a root-match. By Theorem 2.31, it is integral. Therefore, we can prove by induction that all elements pushed into the stacks are integral. □

Lemma 2.36. *Let q be a query node. Let en and en' be two entries of stack S_q. If en is below en' in S_q, then $en.e$ is an ancestor of $en'.e$.*

Proof. It is easy to see that pushing an entry into stack S_q happens only in function COPY-TO-STACK(q), and in one call of function COPY-TO-STACK(q) only one entry is pushed into S_q. Each entry pushed into S_q is a pair consisting of the current element of stream T_q and a pointer. Function COPY-TO-STACK(q) is only called in line 3 of function FIND-USEFUL(q) (or line 4 of function PATH-MATCHINGS if q is the root), and each (potential) call is followed immediately by an ADVANCE operation on T_q. Therefore, if en is below en' in S_q, i.e., en is pushed into stack before en', then $en.e$ is before $en'.e$ in T_q, i.e., $en \Rightarrow start < en' \Rightarrow start$. Line 1 of function COPY-TO-STACK ensures that $en' \Rightarrow start > en \Rightarrow end$, so by Proposition 1.1 $en.e$ is an ancestor of $en'.e$. □

Lemma 2.37. *Algorithm* INTEGRAL-MATCH *outputs only path-matchings.*

Proof. Let pm be an output of Algorithm INTEGRAL-MATCH. It is easy to see that it is of the form $\{< q_i, e_i >\}$ $(1 \leq i \leq h)$, where $\{q_i\}$ are nodes of one of the root-to-leaf paths (h varies for each root-to-leaf path) and e_i is an element from stack S_{q_i}. Thus pm is a function from nodes of a root-to-leaf path of the pattern tree to the target tree. Since e_i is from S_{q_i}, it is from stream T_{q_i}, and hence e_i has the same label as q_i. Let q_j and q_k be two

query nodes from $\{q_i\}$ such that q_j is the parent of q_k, by Lemma 2.36 and Lemma 2.34, e_j is an ancestor of e_k. Therefore, by definition of matching (Def. 1.1) pm is a matching from a root-to-leaf path of the pattern tree to the target tree, and thus it is a path-matching (Def. 2.9). □

Lemma 2.38. *Let P be the pattern tree and T be the target tree. Let pm be a path-matching from a root-to-leaf path p of P to T. If $pm(q)$ is integral for each $q \in p$, then pm is a subset of some matching(s) from P to T.*

Proof. Let $p = q_1 \cdots q_n$ and q_1 is the root. Suppose there is a matching M such that $< q_i, pm(q_i) > \in M$ $(1 \leq i < k \leq n)$, we will prove in the following that there will be a matching M' such that $< q_i, pm(q_i) > \in M'$ $(1 \leq i \leq k)$. Since $pm(q_k)$ is integral, then $pm(q_k)$ is a q_k-match. In other words, there is a matching M_{q_k} from the subtree P_{q_k} rooted at q_k to the target tree. We define M' to be the function such that $M'(q) = M_{q_k}(q)$ if $q \in P_{q_k}$ and $M'(q) = M(q)$ otherwise. It is easy to verify that $M'(q)$ is a matching. Since $pm(q_1)$ is integral, there is a matching M such that $< q_1, pm(q_1) > \in M$. Thus, we can prove by mathematical induction that there is a matching M' such that $pm \subseteq M'$. □

Lemma 2.39. *Each path-matching output by Algorithm* INTEGRAL-MATCH *is a subset of some matching(s) from the pattern tree to the target tree.*

Proof. By Lemma 2.35 and Lemma 2.38. □

Lemma 2.40. *Each path-matching output by Algorithm* INTEGRAL-MATCH *is distinct.*

Proof. It is easy to verify. □

Theorem 2.41 (Soundness)**.** *Each path-matching output by Algorithm* INTEGRAL-MATCH *is necessary for some matching(s) from the pattern tree to the target tree.*

Proof. Let P be a pattern tree and T be a *target* tree. Let $\{p_i\}$ $(1 \leq i \leq n)$ be the set of all root-to-leaf paths of P. Let q_i be the corresponding leaf query node of p_i. Let PM_i be the set of path-matchings corresponding to path p_i. It is easy to see that pairs containing q_i only appear in path-matchings in PM_i. Since the domain of a matching is all query nodes of P, i.e., it contains all leaf query nodes, to make up a matching we need at least one path-matching from each set PM_i $(1 \leq i \leq n)$. It is easy to see that the merge result of two different path-matchings corresponding to the same path is empty, so to make up a non-empty matching we need exactly one path-matching from each set PM_i $(1 \leq i \leq n)$.

Let pm_i be an arbitrary path-matching in set PM_i, and suppose pm_i is a subset of M (by Lemma 2.39). Let pm_i' be an path-matching other than pm_i in the same set. The domain of pm_i and pm_i' are the same, so there must exist a query node q in the domain of pm_i and pm_i' such that $pm_i(q) \neq pm_i'(q)$. Let M' be an arbitrary matching made up by using pm_i', it is easy to see that $M'(q) = pm_i'(q) \neq pm_i(q) = M(q)$, so $M' \neq M$. Therefore, pm_i is necessary for M. □

Lemma 2.42. *Let q be a non-root query node. If e_q is an integral element of stream T_q, then there is a call of function* IS-INTEGRAL*(q) which happens when e_q is the current element of T_q.*

Proof. Line 4 and line 11 are the only two places where stream T_q can

be advanced. The conditions in line 10 and line 12 ensure that elements advanced in line 11 are not q-match and hence not integral. For all the other elements in T_q, before they are advanced, function IS-INTEGRAL(q) is called in line 3 when e_q is the current element of T_q.

The above arguments hold when the streams are infinity long. When the streams are finite long, it is arguable, since condition in line 2 requires that for each e_q advanced in line 4 there must exists an element e_p in stream T_p such that $e_q.start < e_p.start$, which in fact might not exist, e.g, stream T_p is empty. To solve this problem, we append to the end of all streams a dummy element which has start value greater than any start values, e.g., positive infinity. □

Lemma 2.43. *Let q be a non-root query node and p be the parent of q. Let e_p be an element in Stack S_p and e_q be a descendant of e_p. Let c_q be a call of function* IS-INTEGRAL*(q) which happens when e_q is the current element of T_q. Then, e_p is on stack S_p during c_q.*

Proof. First we prove that e_p is pushed into stack S_p before c_q is called. c_q is called only if $e_q.start < e'_p.start$, where e'_p is the current element of stream. Since e_q is a descendant of e_p, $e_q.start > e_p.start$, and hence $e_p.start < e'_p.start$. Thus, stream T_p has been advanced to an element which is after e_p, which implies that e_p has been advanced and pushed into stack S_p.

Next we prove that e_p is popped out of stack S_p after c_q is called. e_p will be popped out of S_p only if we are going to push another integral element e'_p into S_p and $e'_p.start > e_p.end$, which implies that function IS-INTEGRAL(p)

has been called when e'_p is the current element of T_p and the call returns OK. Thus, function FIND-USEFUL(q) has been called (let the call be c'_q) and c'_q returns OK. This implies that the current element of T_q (let it be e_q) at the time c'_q was called is useful with respect to e'_p. Thus, e'_q is a descendant of e'_p and hence $e'_q.start > e'_p.start$. Since $e'_p.start > e_p.end$ and e_q is a descendant of e_p, $e'_q.start > e_p.end > e_q.start$. Therefore, stream T_q has been advanced to an element after e_q, which implies that c_q was already called. □

Corollary 2.44. *Let q be a non-root query node and p be the parent of q. Let e_p be an element in Stack S_p, e_q be an element in Stack S_q. If e_p is an ancestor of e_q, then e_p is pushed into S_p before e_q is pushed into S_q.*

Proof. By Lemma 2.43. □

Lemma 2.45. *Every integral element is pushed into a stack once.*

Proof. It is easy to see that every integral element in stream T_{ROOT} is pushed into stack S_{ROOT} once. Let q be a non-root query node and e_q is a integral element in stack S_q. By Lemma 2.42 there is a call of function IS-INTEGRAL(q) which happens when e_q is the current element of T_q. By Lemma 2.43 and Lemma 2.33, the call should return OK, and hence e_q will be pushed into stack S_q. □

Lemma 2.46. *Let $p = q_1 \cdots q_h$ be a root-to-leaf path and pm be a path-matching corresponding to p output by Algorithm* INTEGRAL-MATCH. *Let c_{q_h} be a call of function* IS-INTEGRAL*(q_h) which happens when $pm(q_h)$ is the*

current element of T_{q_h}. *Then,* $pm(q_i)$ *(*$1 \leq i < h$*) is on stack* S_{q_i} *during* c_{q_h}.

Proof. If $i = h - 1$ or $h \leq 2$, this lemma is equivalent to Lemma 2.43, which has been proved. Assume $i < h - 1$ and $h > 2$. Recursively using Corollary 2.44 we can prove that $pm(q_i)$ is pushed into stack S_{q_i} before $pm(q_{h-1})$ is pushed into stack $S_{q_{h-1}}$. By Lemma 2.43 $pm(q_{h-1})$ is in stack $S_{q_{h-1}}$ during c_q. Therefore, $pm(q_i)$ is pushed into stack S_{q_i} before c_{q_h}.

Next we prove that $pm(q_i)$ is popped out of stack S_{q_i} after c_{q_h}. $pm(q_i)$ will be popped out of S_{q_i} only if we are going to push another integral element e'_{q_i} into S_{q_i} and $e'_{q_i}.start > pm(q_i).end$, which implies that function IS-INTEGRAL(q_i) has been called when e'_{q_i} is the current element of T_{q_i} and the call returns OK. Thus, function FIND-USEFUL(q_{i+1}) has been called (let the call be $c'_{q_{i+1}}$) and $c'_{q_{i+1}}$ returns OK. This implies that the current element of $T_{q_{i+1}}$ (let it be $e'_{q_{i+1}}$) at the time $c'_{q_{i+1}}$ was called is useful with respect to e'_{q_i}. Thus, $e'_{q_{i+1}}$ is a descendant of e'_{q_i} and hence $e'_{q_{i+1}}.start > e'_{q_i}.start$. Furthermore, since $e'_{q_{i+1}}$ is a q_{i+1}-match, similar analysis can keep going on. At the end, we can prove there was a set of function calls $\{c'_{q_j}\}$ $(i < j \leq h)$ and a set of elements $\{e'_{q_j}\}$ $(i < j \leq h)$ such that e'_{q_j} is the current element of T_{q_j} at the time $\{c'_{q_j}\}$ was called, and $e'_{q_j}.start > e'_{q_{j-1}}.start$. Therefore, $e'_{q_h}.start > e'_{q_i}.start > pm(q_i).end > pm(q_h).start$. Since stream T_{q_h} has been advanced to an element after $pm(q_h)$, c_{q_h} should have been called. □

Corollary 2.47. *Let* $p = q_1 \cdots q_h$ *be a root-to-leaf path and pm be a path-matching corresponding to p output by Algorithm* INTEGRAL-MATCH. *Let* c_{q_h} *be a call of function* COPY-TO-STACK(q_h) *which happens when* $pm(q_h)$ *is*

the current element of T_{q_h}. *Then,* $pm(q_i)$ $(1 \leq i < h)$ *is on stack* S_{q_i} *during* c_{q_h}.

Proof. Let c'_{q_h} be a call of function IS-INTEGRAL(q_h) which happens when $pm(q_h)$ is the current element of T_{q_h}. It is easy to see that c_{q_h} is immediately after c'_{q_h} and there is no popping operation on any stacks other than S_{q_h} during c'_{q_h} and c_{q_h}, since by Lemma 2.46 $pm(q_i)$ is in stack S_{q_i} during c'_{q_h}, $pm(q_i)$ is still in stack S_{q_i} during c_{q_h}. □

Theorem 2.48 (Completeness)**.** *Let* P *be a pattern tree and* T *be a target tree. Let* $\{p_1, \ldots, p_n\}$ *be the set of all root-to-leaf paths of* P. *Let the output of Algorithm* INTEGRAL-MATCH *be* $\{PM_i\}$ $(1 \leq i \leq n)$, *where* PM_i *is the set of path-matchings corresponding to path* p_i. *Let* M *be a matching from* P *to* T, *then there exist* n *path-matchings* $\{pm_1, \ldots, pm_n\}$ *such that* $pm_i \in PM_i$ $(1 \leq i \leq n)$ *and* $M = pm_1 \oplus \cdots \oplus pm_n$.

Proof. Let pm_i $(1 \leq i \leq n)$ be a path-matching corresponding to path p_i such that $pm_i(q) = M(q)$ for $q \in p_i$. It is easy to see that $M = pm_1 \oplus \cdots \oplus pm_n$, so we only need to prove that $pm_i \in PM_i$.

Let $p_i = q_1 \cdots q_h$ where q_1 is the root. It is easy to see that $pm_i(q_j)$ $(1 \leq j \leq h)$ are integral. By Lemma 2.45, $pm_i(q_h)$ was pushed into stack S_{q_h} once. Let c_{q_h} be the call of function COPY-TO-STACK(q) which pushed $pm_i(q_h)$ into stack S_{q_h}. By Corollary 2.47, $pm_i(q_j)$ $(1 \leq j < h)$ was in stack S_{q_j} during c_{q_h}. Therefore, during c_{q_h} pm_i was encoded in the stacks as an intermediate result, and thus it will be output in line 7 of c_{q_h}. □

2.3.4 Complexity

Definition 2.12. Let P be a pattern tree, T be a target tree, and q be a query node of P. Let H be the set of all root-to-leaf paths of T. Let E_q^h be the set of all elements in h having the same label as q, where h is one of the paths in H. We define the *degree of self containment* of q as

$$D_q = \max_{h \in H} \{|E_q^h|\}.$$

Definition 2.13 (decomposition)**.** Let M be a matching from pattern tree P to target tree T. Let $\{p_i\}$ $(1 \leq i \leq n)$ be the set of all root-to-leaf paths of P. The decomposition of M is a set $\{pm_i\}$ $(1 \leq i \leq n)$, where pm_i is a path-matching from p_i to T such that $pm_i(q) = M(q)$ for $q \in p_i$.

Definition 2.14. Let m be a matching, we define $|m|$ as the domain size of m. Let M be a set of matchings, we define $|M|$ as the number of matchings in M, and define $\|M\|$ as $\sum_{m \in M} |m|$.

Theorem 2.49 (Complexity)**.** *Given pattern tree P and target tree T. Let M be the set of all matchings from P to T. Let PM be the set of path-matchings decomposed from M. Algorithm* INTEGRAL-MATCH *'s time complexity is $O(\sum_{q \in P} |T_q| + \|PM\|)$, its space complexity is $O(\sum_{q \in P} D_q)$, and its I/O complexity is $O(\sum_{q \in P} |T_q|)$ reading and $O(\|PM\|)$ writing.*

Proof. The correctness of I/O complexity and space complexity of Algorithm INTEGRAL-MATCH is easy to verify.

The cost of all calls of function OUTPUT-PATHS is clearly $O(\|PM\|)$. Let q be an arbitrary query node. The cost of all calls of function COPY-TO-

STACK(q) (excluding the cost of calls of function OUTPUT-PATHS) can be amortized to integral elements in stream T_q, such that the cost amortized to each integral element of stream T_q is $O(\text{CHILDREN}(q))$. Note that if e_q is an integral element, there will be at least CHILDREN(q) path-matchings, each of which contains the pair $< q, e_q >$. Thus, the cost of all calls of function COPY-TO-STACK is $O(\|PM\|)$.

The cost of the **while** loop in function IS-INTEGRAL(q) can be amortized to elements of stream T_q and integral elements of stream T_p, where p is the parent of q, such that the cost amortized to each element of T_q is constant and the cost amortized to each integral element of T_p is $O(\text{CHILDREN}(q))$. Therefore, the cost of **while** loops in function IS-INTEGRAL is $O(\sum_{q \in P} |T_q| + \|PM\|)$.

We will prove in the following that the cost of all calls of function IS-MATCH in function IS-INTEGRAL is $O(\sum_{q \in P} |T_q| + \|PM\|)$. The cost of function IS-MATCH(q) depends on the number of comparisons happened in it. Similar to the proof of complexity of Algorithm Q-MATCH, we can prove that for an arbitrary element e_n, the comparison between e_n and other elements, which happened in a call of function IS-MATCH(q), can be amortized to stream elements and q-matches such that only a constant number of comparisons are amortized to each stream elements and $O(|P_q|)$ comparisons are amortized to each q-matches, where P_q is the subtree rooted at q. The comparisons amortized to q-matches are those comparisons between e_n and its ancestors $\{e_p^i\}$ $(1 \leq i \leq m)$ which happened in function FIND-USEFUL(n). Similar to the proof of Lemma 2.27 and Lemma 2.28, we can prove that e_n and each element in $\{e_p^i\}$ (except e_p^m) is integral. Thus, there will be at

least $m-1$ path-matchings. Instead of amortizing the comparisons between e_n and $\{e_p^i\}$ $(1 \leq i \leq m)$ to q-matches, we could amortize the comparisons between e_n and $\{e_p^i\}$ $(1 \leq i < m)$ to the corresponding path-matching and amortize the comparisons between e_n and e_p^m to e_n. Therefore, the cost of all calls of function IS-MATCH in function IS-INTEGRAL is $O(\sum_{q \in P} |T_q| + \|PM\|)$.

The cost of all other statements in one call of function IS-INTEGRAL (excluding the **while** loop and calls to function IS-MATCH) is constant. The number of calls of function IS-INTEGRAL is linear to the number of stream elements.

In conclusion, the time complexity of Algorithm INTEGRAL-MATCH is $O(\sum_{q \in P} |T_q| + \|PM\|)$. □

2.4 Simple PC-edge-only Tree Pattern Query

The correctness of Algorithm Q-MATCH is based on the fact that the pattern tree contains only ancestor-descendant edges. Figure 2.7 shows an example of Tree Pattern Query with parent-child edges, which can not be solved by Algorithm Q-MATCH correctly.

The streams corresponding to the target tree are $T_A = \{A_0, A_1, \ldots, A_n\}$, $T_B = \{B_1, B_2, \ldots, B_{2n}\}$, $T_C = \{C_1, C_2, \ldots, C_n, [C_{n+1}]\}$, where C_{n+1} may or may not exist. Initially we are at the beginning of all streams, i.e., the current elements of streams are A_0, B_1, and C_1. We can not advance stream T_B or T_C because it will violate the property that all ADVANCE operations are safe, since B_1 and C_1 are potentially useful with respect to A_0. We have to advance stream T_A first, and hence A_0 has to be either output or discarded. Both operations have risk: we may either output an incorrect result or lose a correct one, depending on whether C_{n+1} exists or not.

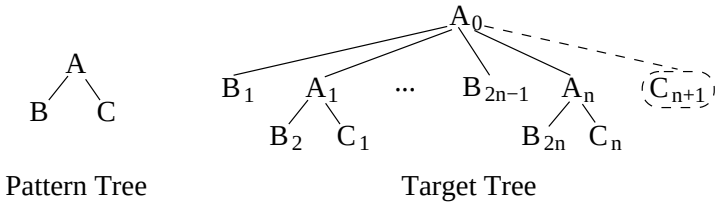

Figure 2.7: Tree Pattern Query with PC-edges

The core of the above problem is that we need to know if there exists an element in T_C which is a child of A_0, i.e., if C_{n+1} exits. Note that if we are only allowed to sequentially scan a stream, the cost of checking if C_{n+1} exists is linear to the size of the whole stream, since we have to advance over all the

elements before C_{n+1}. Carefully examining the element C_{n+1}, we find that although it is at a very late position (actually the last one) in stream T_C, it is the first element among those elements which have a level value equal to $A_0.level+1$. If the elements of stream T_C were indexed by their level values, then we could access C_{n+1} without the cost of advancing over elements C_1 to C_n. In this section, we will discuss several indexing methods, and show that with the use of the level index we can modify Algorithm Q-MATCH to solve the PC-edge-only Tree Pattern Query problem optimally.

2.4.1 Indexing Stream By Level

We would like to index the elements of a stream by level, and at the same time still keep them sorted by start value, since our previous algorithms use this property. A naive solution is to add one extra field to each element, which is a pointer pointing to the next element having the same level value in the same stream. Figure 2.8 shows an example of indexing using pointers. The problem with this method is the cost of extra pointers and the cost of maintaining these pointers, which is much higher on secondary storage than in memory.

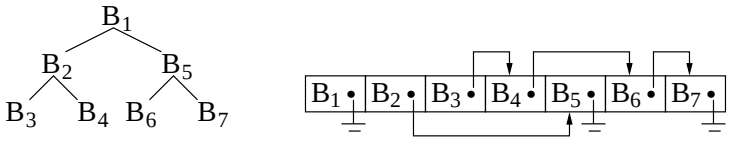

Figure 2.8: Indexing Stream Using Pointers

Another way, which overcomes these inefficiencies, is to split the stream

physically into several sub-streams, each of which only contains elements having the same level value. Elements in the same sub-stream are still sorted by their start value. Clearly this method has less storage and maintenance cost, and the cost to retrieve each element of a sub-stream in a sequential scan is constant. Stream T_q can be obtained by merging all corresponding sub-streams on line.

In the following sections, we assume the second more efficient method is used.

2.4.2 Notation and Data Structures

Each query node q is associated with a set of virtual sub-streams $\{T_q^\ell\}$ $(0 \leq \ell < H)$, where H is the height of the target tree. T_q^ℓ is an image of the sub-stream in which all elements have the same label as q and have level value equal to ℓ. Like the cursor of a virtual stream, we assume each virtual sub-stream T_q^ℓ has its own cursor pointing to the current element and initially the cursor is pointing to the first element of the sub-stream. Function CURRENT(T_q^ℓ) returns the current element of the sub-stream T_q^ℓ, and function ADVANCE(T_q^ℓ) moves the cursor of T_q^ℓ forward by one element. Function EOF(T_q^ℓ) checks if the cursor is at the end of stream T_q^ℓ.

2.4.3 Algorithm LEVEL-MATCH

Algorithm LEVEL-MATCH is based on Algorithm Q-MATCH. The only difference is the use of sub-streams. For example, if we need to check if an element e_q is a q-match, instead of looking for corresponding useful elements in stream T_n where n is an arbitrary children of q (our example showed that

this approach is too costly), we look for corresponding useful elements in sub-stream $T_n^{\ell+1}$ where $\ell = e_q.level$.

2.4.4 Correctness

Lemma 2.50. *Let P be the pattern tree, T be the target tree, and q be a query node of P. Let P_q be the subtree rooted at q. Then the following properties hold*

1. *during any calls of function* FIND-USEFUL*(q, h) only sub-streams $\{T_n^k\}$ are used, where $n =$* PARENT*(q) or $n \in P_q$, $k = \ell + \ell_n - \ell_q$, and ℓ_n (resp. ℓ_q) is the level of n (resp. q).*

2. *during any calls of function* IS-MATCH*(q, ℓ) only sub-streams $\{T_n^k\}$ are used, where $n \in P_q$, $k = \ell + \ell_n - \ell_q$, and ℓ_n (resp. ℓ_q) is the level of n (resp. q).*

Proof. When q is a leaf query node, the lemma is easy to verify. When q is an internal query node, it can be proved recursively by mathematical induction. □

Lemma 2.51. *Let P be a PC-edge-only pattern tree and T be the target tree. Let e be an element in sub-stream T_{ROOT}^{ℓ}. If e is a root-match, then there exists a matching M from P to T such that $M(\text{ROOT}) = e$ and for any query node q of P, $M(q)$ is from sub-stream $T_q^{\ell+\ell_q}$ where ℓ_q is the level of q in P.*

Proof. By definition of root-match (Def. 1.2) and matching (Def. 1.1). □

Algorithm 2.3 Algorithm LEVEL-MATCH

```
ROOT-MATCHES(P, T)
1  for (q ∈ P) L_q ← NULL
2  for (ℓ ∈ [0, H))
3      while (¬EOF(T^ℓ_ROOT))
4          repeat NOTHING until (IS-MATCH(ROOT, ℓ) ≠ NO)
5          OUTPUT(CURRENT(T^ℓ_ROOT))
6          ADVANCE(T^ℓ_ROOT)

IS-MATCH(q, ℓ)
1  if (L_q ≠ CURRENT(T^ℓ_q))
2      L_q ← CURRENT(T^ℓ_q)
3      for (n ∈ CHILDREN(q))
4          if (FIND-USEFUL(n, ℓ + 1) = NO)
5              repeat ADVANCE(T^ℓ_q)
6              until (CURRENT(T^{ℓ+1}_n).start < CURRENT(T^ℓ_q).end)
7              return NO
8  return OK

FIND-USEFUL(q, ℓ)
 1  p ← PARENT(q)
 2  while (CURRENT(T^ℓ_q).start < CURRENT(T^{ℓ-1}_p).start)
 3      ADVANCE(T^ℓ_q)
 4  if (CURRENT(T^ℓ_q).start > CURRENT(T^{ℓ-1}_p).end)
 5      return NO

 6  while (L_q ≠ CURRENT(T^ℓ_q))
 7      L_q ← CURRENT(T^ℓ_q)
 8      for (n ∈ CHILDREN(q))
 9          if (FIND-USEFUL(n, ℓ + 1) = NO)
10              repeat ADVANCE(T^ℓ_q)
11              until (CURRENT(T^{ℓ+1}_n).start < CURRENT(T^ℓ_q).end)
12              if (CURRENT(T^ℓ_q).start > CURRENT(T^{ℓ-1}_p).end)
13                  return NO
14              break
15 return OK
```

Theorem 2.52 (Correctness)**.** *Algorithm* LEVEL-MATCH *outputs all and only root-matches of the given Simple PC-edge-only Tree Pattern Query problem.*

Proof. Algorithm LEVEL-MATCH can be viewed as solving H Simple PC-edge-only Tree Pattern Query problems, each of which is to find root-matches in one sub-stream T^{ℓ}_{ROOT} ($\ell \in [0, H)$). By Lemma 2.50, if $\ell_1 \neq \ell_2$, function IS-MATCH(ROOT, ℓ_1) and function IS-MATCH(ROOT, ℓ_2) will not use any sub-streams in common. Therefore, the processes of finding root-matches of each sub-stream are all independent from each other.

Let the pattern tree be P and the target tree be T. Let P' be a pattern tree which is the result of replacing every parent-child edge of P with a corresponding ancestor-descendant edge. Let c be a call of function IS-MATCH(ROOT, ℓ) where $\ell \in [0, H)$. Since Algorithm LEVEL-MATCH is based on Algorithm Q-MATCH, c will return OK if the current element of stream T^{ℓ}_{ROOT} corresponds to the root query node of P' in some matching (let it be M) from P' to T. Let p and q be two query nodes of P'. By Lemma 2.50 element $M(p)$ (resp. $M(q)$) is from sub-stream $T_p^{\ell+\ell_p}$ (resp. $T_p^{\ell+\ell_q}$), and thus $M(q).level = M(p).level + 1$. Since $M(p)$ is a ancestor of $M(q)$, $M(p)$ is a parent of $M(q)$. Thus, M is also a matching from P to T. Therefore, c will return OK if the current element of stream T^{h}_{ROOT} is a root-match.

Let e be an element of sub-stream T^{ℓ}_{ROOT} which is advanced during c. We know that there is no matching M from P' to T such that $M(\text{ROOT}) = e$. It is easy to see that a matching from P to T is also a matching from P' to T. Thus, there is no matching from P to T such that $M(\text{ROOT}) = e$ and

for each query node q of P, $M(q)$ is from sub-stream $\{T_q^{\ell+\ell_q}\}$ where ℓ_q is the level of q in P. By Lemma 2.51, e is not a root-match. Similarly, we can prove that if c returns NO, then the current element of sub-stream T_{ROOT}^{ℓ} right before c is called is not a root-match.

Therefore, Algorithm LEVEL-MATCH outputs all and only root-matches of the given Simple PC-edge-only Tree Pattern Query problem. □

2.4.5 Complexity

Theorem 2.53 (Complexity)**.** *Given pattern tree P and target tree T. Let R be the set of all root-matches. Algorithm* LEVEL-MATCH*'s time complexity is $O(\sum_{q\in P}|T_q|)$, its space complexity is $O(|P|)$, and its I/O complexity is $O(\sum_{q\in P}|T_q|)$ reading and $O(|R|)$ writing.*

Proof. It is easy to verify the space complexity and I/O complexity of Algorithm LEVEL-MATCH.

In the proof of complexity of Algorithm Q-MATCH, we showed that for an arbitrary element e_q, the comparison between e_q and other elements can be amortized to stream elements and root-matches such that only a constant number of comparisons are amortized to each stream elements and $O(|P|)$ comparisons are amortized to each root-matches. Recall that the comparisons amortized to root-matches are those comparisons between e_q and its ancestors $\{e_p^i\}$. However, for Simple PC-edge-only Tree Pattern Query, e_q will be only compared with one ancestor, i.e., its parent. We do not need to amortize the comparisons between e_q and its parent to any root-match, we can just amortize the comparisons to e_q itself such that the

number of comparisons amortized to each stream element is still constant. Therefore, the time complexity of Algorithm LEVEL-MATCH is $O(\sum_{q \in P} |T_q|)$.

□

2.5 PC-edge-only Tree Pattern Query

With the use of sub-streams, we could modify Algorithm INTEGRAL-MATCH to solve PC-edge-only Tree Pattern Query problems. However, unlike in AD-edge-only Tree Pattern Query problems, where one element could have multiple ancestors, in PC-edge-only Tree Pattern Query problems one element can only have one parent, so we could actually eliminate the use of stacks. Therefore, instead of modifying Algorithm INTEGRAL-MATCH, in this section we will present a new algorithm, Algorithm EXTEND-MATCH, to solve PC-edge-only Tree Pattern Query problem optimally.

2.5.1 Algorithm EXTEND-MATCH

Algorithm EXTEND-MATCH is based on Algorithm LEVEL-MATCH. Once we find a root-match e_{ROOT}, we will call function EXTEND-MATCH on ROOT, which will scan corresponding sub-streams of children of ROOT, and find all elements which are useful with respect to e_{ROOT}. By Theorem 2.31 those elements are also integral. Once we find one of these elements (let it be e_q from a sub-stream of query node q), we will continue the same expanding process by calling function EXTEND-MATCH on q. Once we find an integral element corresponding to a leaf query node, which implies that we have found a path-matching, we will output the path-matching, which is stored in variable *path* and accumulated during the expanding process. It is easy to see that the path-matchings output by Algorithm EXTEND-MATCH are sorted in root-to-leaf order, so they can be merged directly to generate matchings.

Algorithm 2.4 Algorithm EXTEND-MATCH

```
PATH-MATCHINGS(P, T)
1  for (q ∈ P) L_q ← NULL
2  for (ℓ ∈ [0, H))
3      while (¬EOF(T^ℓ_ROOT))
4          repeat NOTHING until (IS-MATCH(ROOT, ℓ) ≠ NO)
5          EXTEND-MATCH(ROOT, ℓ, NULL)
6          ADVANCE(T^ℓ_ROOT)

IS-MATCH(q, ℓ)
1  if (L_q ≠ CURRENT(T^ℓ_q))
2      L_q ← CURRENT(T^ℓ_q)
3      for (n ∈ CHILDREN(q))
4          if (FIND-USEFUL(n, ℓ + 1) = NO)
5              repeat ADVANCE(T^ℓ_q)
6              until (CURRENT(T^{ℓ+1}_n).start < CURRENT(T^ℓ_q).end)
7              return NO
8  return OK

FIND-USEFUL(q, ℓ)
 1  p ← PARENT(q)
 2  while (CURRENT(T^ℓ_q).start < CURRENT(T^{ℓ-1}_p).start)
 3      ADVANCE(T^ℓ_q)
 4  if (CURRENT(T^ℓ_q).start > CURRENT(T^{ℓ-1}_p).end)
 5      return NO

 6  while (L_q ≠ CURRENT(T^ℓ_q))
 7      L_q ← CURRENT(T^ℓ_q)
 8      for (n ∈ CHILDREN(q))
 9          if (FIND-USEFUL(n, ℓ + 1) = NO)
10              repeat ADVANCE(T^ℓ_q)
11              until (CURRENT(T^{ℓ+1}_n).start < CURRENT(T^ℓ_q).end)
12              if (CURRENT(T^ℓ_q).start > CURRENT(T^{ℓ-1}_p).end)
13                  return NO
14              break
15 return OK
```

Algorithm 2.4 Algorithm EXTEND-MATCH *(continued)*

```
EXTEND-MATCH(q, ℓ, path)
1  path ← path + < q, CURRENT(T_q^ℓ) >
2  if (IS-LEAF(q)) OUTPUT(path)

3  for (n ∈ CHILDREN(q))
4      while (FIND-USEFUL(n, ℓ + 1) = OK)
5          EXTEND-MATCH(n, ℓ + 1, path)
6          ADVANCE(T_n^{ℓ+1})
```

2.5.2 Correctness

Lemma 2.54. *Algorithm* EXTEND-MATCH *outputs only path-matchings. Let pm be a path-matching output by Algorithm* EXTEND-MATCH*. For all query nodes q in the domain of pm, pm(q) is integral.*

Proof. It is easy to see that each output pm is a set of pairs $\{< q_i, e_i >\}$ where $\{q_i\}$ is a root-to-leaf path and $q_1 =$ ROOT. Pair $< q_i, e_i >$ is generated by a call of function EXTEND-MATCH(q_i, h_i, *path*) (let it be c_{q_i}), where e_i is the current element of $T_{q_i}^{h_i}$ when c_{q_i} is called. Line 4 of function PATH-MATCHINGS ensures that e_1 is a root-match. Line 4 of function EXTEND-MATCH ensures that e_i is useful with respect to e_{i-1}. Thus, each e_i is integral, and pm is a path-matching. □

Lemma 2.55. *Each path-matching output by Algorithm* EXTEND-MATCH *is a subset of some matching(s) from the pattern tree to the target tree.*

Proof. By Lemma 2.38 and Lemma 2.54. □

Lemma 2.56. *Each path-matching output by Algorithm* EXTEND-MATCH *is distinct.*

Proof. Follows immediately from the algorithm. □

Theorem 2.57 (Soundness)**.** *Each path-matching output by Algorithm* INTEGRAL-MATCH *is necessary for some matching(s) from the pattern tree to the target tree.*

Proof. By Lemma 2.55 and Lemma 2.56. The proof is similar to that of Theorem 2.41. □

Theorem 2.58 (Completeness)**.** *Let P be a pattern tree and T be a target tree. Let $\{p_1, \ldots, p_n\}$ be the set of all root-to-leaf paths of P. Let the output of Algorithm* EXTEND-MATCH *be $\{PM_i\}$ $(1 \leq i \leq n)$, where PM_i is the set of path-matchings corresponding to path p_i. Let M be a matching from P to T, then there exist n path-matchings $\{pm_1, \ldots, pm_n\}$ such that $pm_i \in PM_i$ $(1 \leq i \leq n)$ and $M = pm_1 \oplus \cdots \oplus pm_n$.*

Proof. By Theorem 2.31 every integral element, except a root-match, has to be useful with respect to one other integral element. Line 4 of function EXTEND-MATCH ensures that every integral element has been considered during the path-matching generating process. Based on this fact, the rest proof is similar to that of Theorem 2.48. □

2.5.3 Complexity

Theorem 2.59. *Given pattern tree P and target tree T. Let M be the set of all matchings from P to T. Let PM be the set of path-matchings which*

are decomposed from M. Algorithm EXTEND-MATCH*'s time complexity is $O(\sum_{q\in P}|T_q|+\|PM\|)$, its space complexity is $O(|P|)$, and its I/O complexity is $O(\sum_{q\in P}|T_q|)$ reading and $O(\|PM\|)$ writing.*

Proof. The correctness of I/O complexity and space complexity of Algorithm INTEGRAL-MATCH is easy to verify.

The cost of one call of function EXTEND-MATCH itself (excluding the cost of calls to other functions) is constant. Function OUTPUT, function FIND-USEFUL and function EXTEND-MATCH are called in function EXTEND-MATCH. Each call of function EXTEND-MATCH is followed by at least one ADVANCE operation. Thus, the cost of all calls of function EXTEND-MATCH (excluding the cost of calls to function FIND-USEFUL and function OUTPUT) is linear in the size of stream elements. The cost of all calls of function OUTPUT is clearly $O(\|PM\|)$. Similar to the proof of Theorem 2.53, we can prove that the cost of all calls of function FIND-USEFUL is linear in the size of stream elements. Therefore, the time complexity of Algorithm EXTEND-MATCH is $O(\sum_{q\in P}|T_q|+\|PM\|)$. □

2.6 Simple Tree Pattern Query

In this section we will present Algorithm VERIFY-MATCH to solve the Simple Tree Pattern Query problem, which is based on Algorithm INTEGRAL-MATCH.

2.6.1 Notation and Data Structure

Recall that in Algorithm INTEGRAL-MATCH we associate one stack S_q with each node q of the pattern tree P. Each entry en of stack S_q consists of a pair $< e, pt >$, where e is an element from stream T_q and pt is a pointer to an entry in stack $S_{\text{PARENT}(q)}$. In Algorithm VERIFY-MATCH, we associate with each entry en of stack S_q an integer C_{en} and a boolean array M_{en} of size $|\text{CHILDREN}(q)|$. The value of C_{en} is initially 0. Let n be a child of q. The value of $M_{en}[n]$ can be OK or NO, and is initially NO. We will increase C_{en} by 1 and set $M_{en}[n] = \text{OK}$ if originally $M_{en}[n] = \text{NO}$ and we find an element in T_n which is useful with respect to $en.e$. Thus, if C_{en} becomes equal to $|\text{CHILDREN}(q)|$, then we know that for all $n \in \text{CHILDREN}(q)$ $M_{en}[n] = \text{OK}$, and hence $en.e$ is a q-match.

2.6.2 Algorithm VERIFY-MATCH

Let the pattern tree be P and the target tree be T. Let P' be the result of replacing all parent-child edges of P with ancestor-descendant edges. Recall that Algorithm INTEGRAL-MATCH will find all elements which are integral to matchings from P' to T and push them into corresponding stacks for the use of outputting path-matchings. Note that some of those elements may not

be integral to matchings from P to T, but all elements which are integral to matchings from P to T must be among those elements. In Algorithm INTEGRAL-MATCH, once we find an integral element e_q corresponding to a leaf query node q, we output the corresponding path-matchings. Instead of outputting, in Algorithm VERIFY-MATCH, we will call function MARK-PARENT to set $M_{pe}[q] = \text{OK}$ for each unset entry pe in stack $S_{\text{PARENT}(q)}$ which e_q is useful with respect to. For each such entry pe, we will also increase C_{pe} by 1. Here, variable $M_{pe}[q]$ indicates that we have found an element in T_q which is useful with respect to $pe.e$. Once we find that for some entry pe in stack S_p $M_{pe}[q] = \text{OK}$ for all $q \in \text{CHILDREN}(p)$, i.e., $C_{pe} = |\text{CHILDREN}(p)|$, we have verified that $pe.e$ is truly a p-match from the subtree of P rooted at p to T. This is the origin of the name of Algorithm VERIFY-MATCH. After we have verified that $pe.e$ is a p-match, we will call function MARK-PARENT recursively to mark corresponding entries in stack $S_{\text{PARENT}(p)}$. Once we have verified a root-match, we will output it.

2.6.3 Correctness

Lemma 2.60. *Let q be a query node and en be a entry in stack S_q. If $C_{en} = |\text{CHILDREN}(q)|$, then $en.e$ is a q-match.*

Proof. This lemma holds when q is a leaf query node. When q is an internal node, it is easy to see that $C_{en} = |\text{CHILDREN}(q)|$ only if for any $n \in \text{CHILDREN}(q)$ $M_{en}[n] = \text{OK}$, which implies that there is an entry nn in S_n such that $C_{nn} = |\text{CHILDREN}(n)|$ and $nn.e$ is useful with respect to $en.e$. Thus, we can prove the lemma recursively. □

Algorithm 2.5 Algorithm VERIFY-MATCH

```
ROOT-MATCHES(P, T)
1  for (q ∈ P) {L_q, PT_q} ← NULL
2  while (TRUE)
3      repeat NOTHING until (IS-MATCH(ROOT) ≠ NO)
4      COPY-TO-STACK(ROOT)
5      ADVANCE(T_ROOT)

IS-MATCH(q)
1  if (L_q ≠ CURRENT(T_q))
2      L_q ← CURRENT(T_q)
3      for (n ∈ CHILDREN(q))
4          if (FIND-USEFUL(n) = NO)
5              repeat ADVANCE(T_q)
6              until (CURRENT(T_n).start < CURRENT(T_q).end)
7              return NO
8  return OK

FIND-USEFUL(q)
 1  p ← PARENT(q)
 2  while (CURRENT(T_q).start < CURRENT(T_p).start)
 3      if (IS-INTEGRAL(q) = OK) COPY-TO-STACK(q)
 4      ADVANCE(T_q)
 5  if (CURRENT(T_q).start > CURRENT(T_p).end)
 6      return NO

 7  while (L_q ≠ CURRENT(T_q))
 8      L_q ← CURRENT(T_q)
 9      for (n ∈ CHILDREN(q))
10          if (FIND-USEFUL(n) = NO)
11              repeat ADVANCE(T_q)
12              until (CURRENT(T_n).start < CURRENT(T_q).end)
13              if (CURRENT(T_q).start > CURRENT(T_p).end)
14                  return NO
15              break
16 return OK
```

Algorithm 2.5 Algorithm VERIFY-MATCH *(continued)*

```
IS-INTEGRAL(q)
1  if (PT_q = NULL) return NO
2  while (CURRENT(T_q).start > PT_q ⇒ end)
3      PT_q ← PREV(PT_q)
4      if (PT_q = NULL) return NO
5  if (IS-MATCH(q) = NO)
6      return NO
7  return OK

COPY-TO-STACK(q)
1  while (|S_q| ≠ 0 ∧ CURRENT(T_q).start > TOP(S_q) ⇒ end)
2      POP(S_q)
3  PUSH(S_q, CURRENT(T_q), PT_q)
4  for (n ∈ CHILDREN(q))
5      PT_n ← TOP(S_q)

6  if (IS-LEAF(q))
7      MARK-PARENT(q, TOP(S_q))

MARK-PARENT(q, en)
 1  if (IS-ROOT(q)) OUTPUT(en)
 2  p ← PARENT(q)
 3  pe ← (en.pt)

 4  while (pe ≠ NULL)
 5      if (PC-EDGE(p, q) ∧ (pe ⇒ level < en ⇒ level −1))
 6          break

 7      if (M_pe[q] ≠ OK)
 8          M_pe[q] ← OK
 9          C_pe ← C_pe + 1
10          if (C_pe = |CHILDREN(p)|) MARK-PARENT(p, pe)

11      pe ← PREV(pe)
```

Lemma 2.61. *Function* MARK-PARENT*(q, en) is called only if en.e is a q-match.*

Proof. By Lemma 2.60 and the fact that for any leaf node q every element in T_q is a q-match. □

Lemma 2.62. *Every integral element is pushed into a stack once.*

Proof. By Lemma 2.45 and the fact that all integral elements are also integral to the corresponding AD-edge-only Tree Pattern Query problem, which is generated by replacing all parent-child edges in the patten tree with ancestor-descendant edges. □

Lemma 2.63. *Let q be a query node and en be an entry in stack S_q. If $en.e$ is a q-match, then en is popped out of stack S_q after function* MARK-PARENT*(q, en) is called.*

Proof. Let P be the pattern tree and T be the target tree. Let P' be the result of replacing all parent-child edges of P with ancestor-descendant edges.

Let Q_0 be the set of leaf query nodes and Q_i $(1 \leq i)$ be the set of query nodes such that for any query node q in Q_i, all children of q are in $\bigcup_{0 \leq j < i} Q_j$ but $q \notin \bigcup_{0 \leq j < i} Q_j$. It is easy to see that this lemma holds for query nodes in Q_0.

Let q be a query node in Q_1, en be a entry in stack S_q, and $e_q = en.e$. Since e_q is a q-match from P to T, for each child n_i $(1 \leq i \leq k)$ of q, there exists an element e_{n_i} which is useful with respect e_q. Since e_q is in stack S_q, then e_q is integral to matchings from P' to T. Thus, e_{n_i} is integral to

matchings from P' to T, and hence e_{n_i} is pushed into stack S_{n_i} once. Let c_{n_i} be the call of function COPY-TO-STACK which pushed e_{n_i} into stack S_{n_i}. By Corollary 2.47, e_q is in S_q during c_{n_i}. Let en_i be the entry containing e_{n_i}. Note that MARK-PARENT(n_i, en_i) is called during c_{n_i}, which implies that $M_{en}[n_i]$ is set to OK before or during c_{n_i}. Assume without loss of generality that c_{n_k} is the last call among $\{c_{n_i}\}$. In function MARK-PARENT(n_k, en_k) $M_{en}[n_i] =$ OK for all n_i, and hence the condition in Line 10 is true. Thus, MARK-PARENT(q, en) will be called. Therefore, this lemma holds for query nodes in Q_1.

Let q be a query node in Q_i $(1 < i)$. Similarly we can prove that for each child n_i $(1 \leq i \leq k)$ of q there exists a set of calls MARK-PARENT(n_i, en_i) which happen before en is popped out of stack S_q, and in the last call MARK-PARENT(n_k, en_k) $M_{en}[n_i] =$ OK for all n_i. Thus, MARK-PARENT(q, en) will be called. □

Theorem 2.64 (Correctness)**.** *Algorithm* VERIFY-MATCH *outputs all and only root-matches of the given Simple Tree Pattern Query problem.*

Proof. By Lemma 2.61, Lemma 2.62, and Lemma 2.63. □

2.6.4 Complexity

Theorem 2.65 (Complexity)**.** *Given pattern tree P and target tree T. Let R be the set of all root-matches. Let P' be the result of replacing all parent-child edges of P with ancestor-descendant edges. Let M' be the set of all matchings from P' to T. Let PM' be the set of path-matchings decomposed from M'. Algorithm* VERIFY-MATCH*'s time complexity is $O(\sum_{q \in P} |T_q| +$*

$\|PM'\|$), *its space complexity is* $O(\sum_{q \in P} (D_q \cdot |\text{CHILDREN}(q)|))$ *(*D_q *is defined in Def. 2.12), and its I/O complexity is* $O(\sum_{q \in P} |T_q|)$ *reading and* $O(|R|)$ *writing.*

Proof. It is easy to verify that Algorithm VERIFY-MATCH has the same complexities as Algorithm INTEGRAL-MATCH except the output is of size $|R|$ and the extra cost of memory usage for additional variables. □

2.6.5 Algorithm VERIFY-MATCH-IMPROVED

Algorithm VERIFY-MATCH solves the Simple Tree Pattern Query problem with optimal I/O complexity, but it might not have the optimal time and space complexity. In fact, without affecting its correctness, we could easily improve its speed and decrease its memory usage by some modifications, as shown in Algorithm 2.6. For example:

1. parent-child relationship could be checked before an element is pushed into a stack, as shown in line 6 of function NEED-VERIFY(q);

2. an element e_q could be pushed into stack S_q only if there exists an entry en in stack $S_{\text{PARENT}(q)}$ such that $M_{en}[q]$ is unmarked and e_q may be useful to mark it, as shown in line 1 and line 4 of function NEED-VERIFY(q);

3. at the end of function MARK-PARENT(q, en), i.e., after we have marked all entries in $S_{\text{PARENT}(q)}$ which $en.e$ is useful with respect to, we could remove entry en.

Although these modifications will improve Algorithm VERIFY-MATCH in terms of speed and memory usage, we doubt the worst case asymptotic bound of time and space complexity will change.

Algorithm 2.6 Algorithm VERIFY-MATCH-IMPROVED

```
ROOT-MATCHES(P, T)
1  for (q ∈ P) {L_q, PT_q} ← NULL
2  while (TRUE)
3      repeat NOTHING until (IS-MATCH(ROOT) ≠ NO)
4      COPY-TO-STACK(ROOT)
5      ADVANCE(T_ROOT)

IS-MATCH(q)
1  if (L_q ≠ CURRENT(T_q))
2      L_q ← CURRENT(T_q)
3      for (n ∈ CHILDREN(q))
4          if (FIND-USEFUL(n) = NO)
5              repeat ADVANCE(T_q)
6              until (CURRENT(T_n).start < CURRENT(T_q).end)
7              return NO
8  return OK

FIND-USEFUL(q)
 1  p ← PARENT(q)
 2  while (CURRENT(T_q).start < CURRENT(T_p).start)
 3      if (NEED-VERIFY(q) = OK) COPY-TO-STACK(q)
 4      ADVANCE(T_q)
 5  if (CURRENT(T_q).start > CURRENT(T_p).end)
 6      return NO

 7  while (L_q ≠ CURRENT(T_q))
 8      L_q ← CURRENT(T_q)
 9      for (n ∈ CHILDREN(q))
10          if (FIND-USEFUL(n) = NO)
11              repeat ADVANCE(T_q)
12              until (CURRENT(T_n).start < CURRENT(T_q).end)
13              if (CURRENT(T_q).start > CURRENT(T_p).end)
14                  return NO
15              break
16 return OK
```

Algorithm 2.6 Algorithm VERIFY-MATCH-IMPROVED *(continued)*

```
NEED-VERIFY(q)
 1  PT_q ← UNMARKED(PARENT(q), PT_q)
 2  if (PT_q = NULL) return NO

 3  while (CURRENT(T_q).start > PT_q ⇒ end)
 4      PT_q ← UNMARKED(PARENT(q), PREV(PT_q))
 5      if (PT_q = NULL) return NO

 6  if (PC-EDGE(PARENT(q), q)
        ∧ (PT_q ⇒ level < CURRENT(T_q).level −1))
 7      return NO

 8  if (IS-MATCH(q) = NO)
 9      return NO

10  return OK

UNMARKED(q, en)
1  while ((en ≠ NULL) ∧ (M_en[q] = OK))
2      en ← PREV(en)
3  return en

COPY-TO-STACK(q)
1  while (|S_q| ≠ 0 ∧ CURRENT(T_q).start > TOP(S_q) ⇒ end)
2      POP(S_q)

3  PUSH(S_q, CURRENT(T_q), PT_q)

4  for (n ∈ CHILDREN(q))
5      PT_n ← TOP(S_q)

6  if (IS-LEAF(q))
7      MARK-PARENT(q, TOP(S_q))
```

Algorithm 2.6 Algorithm VERIFY-MATCH-IMPROVED *(continued)*

```
MARK-PARENT(q, en)
 1  if (IS-ROOT(q)) OUTPUT(en)

 2  pe ← UNMARKED(PARENT(q), en.pt)
 3  while (pe ≠ NULL)
 4      if (PC-EDGE(PARENT(q), q) ∧ (pe ⇒ level < en ⇒ level −1))
 5          break

 6      M_pe[q] ← OK

 7      if (VERIFY(pe) = OK)
 8          MARK-PARENT(PARENT(q), pe)

 9      pe ← UNMARKED(PARENT(q), PREV(pe))
10  REMOVE(en)

VERIFY(en)
1  for (mark ∈ M_en)
2      if (mark = NO)
3          return NO

4  return OK

REMOVE(en)
   ▷ Delete entry en and
   ▷ if there is any pointer pointing to en,
   ▷ let it pointing to PREV(en).
```

2.7 Tree Pattern Query

In this section we will present Algorithm BUFFER-MATCH to solve the Tree Pattern Query problem, which is based on Algorithm VERIFY-MATCH.

2.7.1 Notation and Data Structure

We associate with each entry en in stack S_q an array of pointers B_{en} of size $|\text{CHILDREN}(q)|$. Each pointer points a buffer. At any time, the buffer pointed by $B_{en}[n]$ ($n \in \text{CHILDREN}(q)$) is either empty or is a list of items of the form $<< n, e_n >, B_{nn} >$ where B_{nn} is the array of pointers associated with entry nn in stack S_n and $e_n = nn.e$. We use buffers B_{en} (together with other buffers linked from them) to store compactly those matchings $\{M\}$ from the subtree of P rooted at q to T such that $M(q) = en.e$. The buffers can grow very large so we assume that they are stored in secondary storage instead of main memory. Operation APPEND($B_{pe}[q]$, m, B_{en}) appends a new item $< m, B_{en} >$ to buffer B_{pe}. We also allocate one buffer BM to store compactly all the matchings found for each entry in stack S_{ROOT}. Each item of BM is of the form $<< \text{ROOT}, e_{\text{ROOT}} >, B_{en_{\text{ROOT}}} >$, where $B_{en_{\text{ROOT}}}$ is the array of pointers associated with entry en_{ROOT} in stack S_{ROOT} and $e_{\text{ROOT}} = en_{\text{ROOT}}.e$. Let $it =<< n, e_n >, B_{nn} >$ be an item in a buffer, we use $it.match$ to refer the pair $< n, e_n >$, and use $it.buffers$ to refer the pointer array B_{nn}.

Figure 2.9 shows an example of the use of buffers. For simplicity, each item $<< q, e >, p >$ in a buffer is shown as an element and a dot. The matchings encoded in buffer BM are $\{(A \rightarrow A_2, B \rightarrow B_1, C \rightarrow C_1, D \rightarrow$

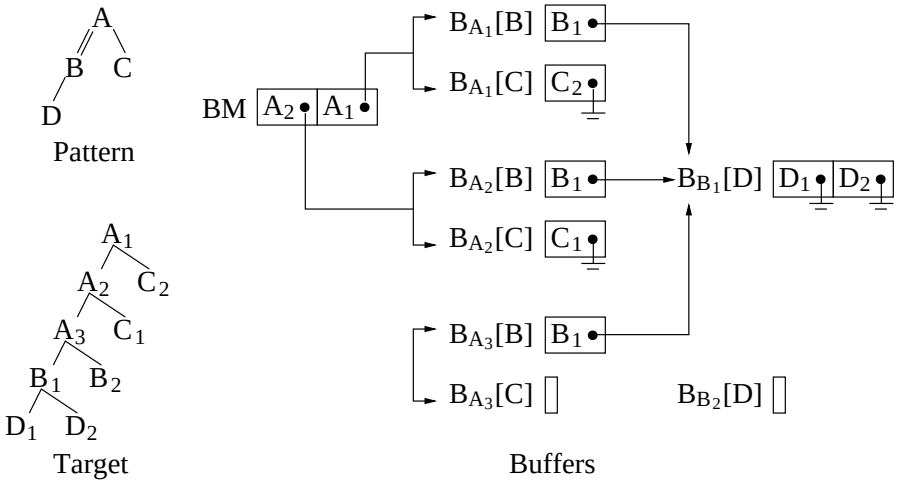

Figure 2.9: Example of Using Buffers

$D_1), (A \rightarrow A_2, B \rightarrow B_1, C \rightarrow C_1, D \rightarrow D_2), (A \rightarrow A_1, B \rightarrow B_1, C \rightarrow C_2, D \rightarrow D_1), (A \rightarrow A_1, B \rightarrow B_1, C \rightarrow C_2, D \rightarrow D_2)\}$, which can be generated by calling function EXTRACT on buffer BM. Function EXTRACT is defined in Algorithm 2.8.

2.7.2 Algorithm BUFFER-MATCH

Algorithm BUFFER-MATCH is based on Algorithm VERIFY-MATCH. Recall that in Algorithm VERIFY-MATCH after we find an entry en in stack S_q such that $en.e$ is a q-match, we will call function MARK-PARENT(q, en) to set $M_{pe}[q] =$ OK for each entry pe in stack $S_{\text{PARENT}(q)}$ where $en.e$ is useful with respect to $pe.e$. In function MARK-PARENT(q, en) of Algorithm BUFFER-MATCH, besides marking entry pe, we will also append to the buffer pointed to by $B_{pe}[q]$ an item $<< q, en.e >, B_{en} >$. If en is an entry in stack S_{ROOT}, we will append to the buffer BM an item $<<$ ROOT$, en.e >, B_{en} >$. After

Algorithm BUFFER-MATCH is finished, the buffer BM will contain all the matchings compactly. Those matchings can be extracted by calling function EXTRACT on buffer BM.

2.7.3 Correctness

Theorem 2.66 (Correctness)**.** *Given pattern tree P and target tree T, after Algorithm* BUFFER-MATCH *is finished, buffer BM stores compactly all and only matchings from P to T.*

Proof. Let $<<$ ROOT$, en.e >, B'_{en} >$ be an item in buffer BM. By Lemma 2.61, $en.e$ is a root-match, and hence it is integral. Let B_{en} be an array of pointers to buffers such that $en.e$ is integral. Let $it =<< q, en_q.e >, B_{en_q} >$ be an item in $B_{en}[q]$. It is easy to see that it is appended to $B_{en}[q]$ only if $en_q.e$ is useful with respect to $en.e$. Since $en.e$ is integral, $en_q.e$ is integral. Therefore, we can prove recursively that all elements in buffer BM and buffers linked with BM directly and indirectly are integral. Similar to the proof of Theorem 2.41, we can prove that BM stores only matchings from P to T.

Corollary 2.47, Lemma 2.62 and Lemma 2.63 ensure that for every entry en in stack S_q that contains an integral element, buffers B_{en} are linked to every buffer $B_{pe}[q]$ such that $en.e$ is useful with respect to $pe.e$. Thus, we can prove recursively that for every item $<<$ ROOT$, e >, B_{en} >$ in buffer BM, all integral elements which can compose a matching with e are linked to this item directly or indirectly. Every matching has to be composed of integral elements. Therefore, all matchings are stored compactly in BM. □

Algorithm 2.7 Algorithm BUFFER-MATCH

```
COMPACT-MATCHINGS(P, T)
1  for (q ∈ P) {L_q, PT_q} ← NULL
2  while (TRUE)
3     repeat NOTHING until (IS-MATCH(ROOT) ≠ NO)
4     COPY-TO-STACK(ROOT)
5     ADVANCE(T_ROOT)

IS-MATCH(q)
1  if (L_q ≠ CURRENT(T_q))
2     L_q ← CURRENT(T_q)
3     for (n ∈ CHILDREN(q))
4           if (FIND-USEFUL(n) = NO)
5              repeat ADVANCE(T_q)
6              until (CURRENT(T_n).start < CURRENT(T_q).end)
7              return NO
8  return OK

FIND-USEFUL(q)
 1  p ← PARENT(q)
 2  while (CURRENT(T_q).start < CURRENT(T_p).start)
 3     if (IS-INTEGRAL(q) = OK) COPY-TO-STACK(q)
 4     ADVANCE(T_q)
 5  if (CURRENT(T_q).start > CURRENT(T_p).end)
 6     return NO

 7  while (L_q ≠ CURRENT(T_q))
 8     L_q ← CURRENT(T_q)
 9     for (n ∈ CHILDREN(q))
10           if (FIND-USEFUL(n) = NO)
11              repeat ADVANCE(T_q)
12              until (CURRENT(T_n).start < CURRENT(T_q).end)
13              if (CURRENT(T_q).start > CURRENT(T_p).end)
14                    return NO
15              break
16 return OK
```

Algorithm 2.7 Algorithm BUFFER-MATCH *(continued)*

```
IS-INTEGRAL(q)
1  if (PT_q = NULL) return NO
2  while (CURRENT(T_q).start > PT_q ⇒ end)
3      PT_q ← PREV(PT_q)
4      if (PT_q = NULL) return NO
5  if (IS-MATCH(q) = NO)
6      return NO
7  return OK

COPY-TO-STACK(q)
1  while (|S_q| ≠ 0 ∧ CURRENT(T_q).start > TOP(S_q) ⇒ end)
2      POP(S_q)
3  PUSH(S_q, CURRENT(T_q), PT_q)
4  for (n ∈ CHILDREN(q))
5      PT_n ← TOP(S_q)
6  if (IS-LEAF(q))
7      MARK-PARENT(q, TOP(S_q))

MARK-PARENT(q, en)
 1  if (IS-ROOT(q)) APPEND(BM, < q, en.e >, B_en)
 2  p ← PARENT(q)
 3  pe ← (en.pt)
 4  while (pe ≠ NULL)
 5      if (PC-EDGE(p, q) ∧ (pe ⇒ level < en ⇒ level −1))
 6          break
 7      APPEND(B_pe[p], < q, en.e >, B_en)
 8      if (M_pe[q] ≠ OK)
 9          M_pe[q] ← OK
10          C_pe ← C_pe + 1
11          if (C_pe = |CHILDREN(p)|) MARK-PARENT(p, pe)
12      pe ← PREV(pe)
```

Algorithm 2.8 Extracting Matchings from Buffers

```
EXTRACT(BM)
1  GENERATE(NULL, {BM})

GENERATE(M, Buffers)
1  if (| Buffers | = 0) OUTPUT(M)

2  buffer ← FIRST(Buffers)
3  REMOVE-FIRST(Buffers)

4  for (item ∈ buffer)
5      NM ← M + item.match
6      NB ← Buffers + item.buffers
7      GENERATE(NM, NB)
```

2.7.4 Complexity

Theorem 2.67 (Complexity)**.** *Given pattern tree P and target tree T. Let P' be the result of replacing all parent-child edges of P with ancestor-descendant edges. Let M' be the set of all matchings from P' to T. Let PM' be the set of path-matchings decomposed from M'. Algorithm* BUFFER-MATCH*'s time complexity is $O(\sum_{q \in P} |T_q| + \|PM'\|)$, its space complexity is $O(\sum_{q \in P} (D_q \cdot |\text{CHILDREN}(q)|))$, and its I/O complexity is $O(\sum_{q \in P} |T_q|)$ reading and $O(\|PM'\|)$ writing.*

Proof. Algorithm BUFFER-MATCH has the the same complexities as Algorithm VERIFY-MATCH except the worst case output is of size $\|PM'\|$. □

2.8 Future Work

Algorithm BUFFER-MATCH solves Tree Pattern Query problem, but its time complexity is not linear in the sum of the input and output lengths. In other words, there might exist an algorithm which is better than Algorithm BUFFER-MATCH in terms of time complexity. On the other hand, both Algorithm INTEGRAL-MATCH and Algorithm EXTEND-MATCH have time complexity linear in the sum of the input and output lengths, and they can solve AD-edge-only Tree Pattern Query problem and PC-edge-only Tree Pattern Query problem, respectively. Is it possible to combine them to create a new algorithm to solve Tree Pattern Query problem? Simply combining them does not work, since Lemma 2.50 will not hold and it is the foundation to prove the correctness of Algorithm EXTEND-MATCH. We are investigating the possibility of combining them plus using B-Tree index to solve Tree Pattern Query problem.

We are also studying summarizing the context of each element and then indexing the stream by different element contexts, where the context of an element could be all or part of its parent, children, ancestors, or descendants. Indexing on element contexts not only facilities the process of solving ordinary Tree Pattern Query problems, but also helps solving the problem of Tree Pattern Query with Wildcards, which seems impossible to solve without additional index. Preliminary results can be found in [17].

Chapter 3

Clone Detection Using Abstract Syntax Tree [5]

3.1 Background and Problem Definition

3.1.1 Clone Detection Techniques

A clone is a program fragment that is identical or almost identical to another fragment. It occurs when a fragment of a program is duplicated and then the copied fragment is, perhaps, edited.

In looking for nearly identical program fragments, most previous methods start with what might be called *lexical abstraction*, a process akin to a compiler's lexical analyzer identifies the elements that can become parameters to an abstracted procedure. Typically, the process treats identifiers and numbers for source code or register numbers and literals for assembly code as equivalent; or, alternatively, replaces them with a canonical form (a "wildcard") in order to detect similar clones. For example, it treats the source codes `i=j+1` and `p=q+4` as if they were identical. Lexical abstraction can generate many false positives, but Baker's parameterized pattern matching

[5] Part of the work described in Chapter 3 appears in the paper by Ma et al. [18].

[5] compensates by requiring a one-for-one correspondence between parallel parameters.

Lexical abstraction has much to recommend it. It is fast and catches many clones. But does it miss clones? That is, could a more general mechanism catch more clones?

One generalization takes advantage of instruction predication (found, for example, in the ARM instruction set [20]) to nullify instructions that differ between similar clones. The parameters to the clones' representative procedure are the predication flags, which select the instructions to execute for each invocation. One flag setting could select an entirely different sequence of instructions than another, however for the representative to be small, many instructions should be common to many clones. A shortest common super-sequence algorithm finds the best representative for a set of similar clones [9]. The method is not intended for large numbers of clones with many parameters.

Another generalization uses slicing to identify non-contiguous clones and then moves irrelevant code out of the way [15]. This extension catches more clones than lexical abstraction, but parameterization remains based on lexical elements. This extension is orthogonal to this thesis's generalization. The two methods could be used together and ought to catch more clones together than separately.

Another natural generalization of lexical abstraction allows the creation of parameters that represent entire subtrees of a parser's tree representation of source code. Parse or derivation trees are riddled with artificial duplication that interests only parsers, but abstract syntax trees project out this

duplication and are roughly equivalent to source code. Subtrees of an AST may correspond to lexical constructs (identifiers or numbers) but they may also correspond to more general constructs that capture more complicated program structures. This generalization we call *structural abstraction*.

There is some prior work on clone detection in ASTs, though not fully general structural abstraction as defined above. One method tracks only a subset of the AST features [16]. Another method [6] uses the full AST, starts with lexical abstraction, and adds a method specialized to detect the common prefix of lists of statements, expressions, etc.

This thesis presents the results of applying general structural abstraction to ASTs. Our work has no special treatment for identifiers, literals, lists, or any other language feature. It bases parameterization only on the abstract syntax tree. It abstracts identifiers, literals, lists, and more, but it does so simply by abstracting subtrees of an AST.

The objective of this work is to determine if full structural abstraction on ASTs is affordable and if it improves significantly on lexical abstraction. Structural abstraction seems inherently more costly, and there is no prima facie evidence that it finds more or better clones.

Finding clones in an AST might appear to be a special case of the problem of mining frequent subtrees [10, 21], but closer examination shows that the two problems operate at two ends of a spectrum. Algorithms that mine frequent trees scan huge forests for subtrees that appear under many roots. The size and exact number of occurrences are secondary to the "support" or number of roots that hold the pattern. An AST-based clone detector makes the opposite trade-off. The best answer may be a clone that occurs

only twice, if it is big enough. Size and exact number of occurrences are important. Support is secondary; indeed, some interesting forests hold only a single root. The existing subtree mining algorithms are not designed or tuned to find clones in ASTs.

3.1.2 Problem Definition

Let S be an AST, and q be an node of S. We use S_q to denote the subtree of S rooted at q. We define function $W(S, q)$ as: if $q \notin S$, $W(S, q) = S$; otherwise, $W(S, q)$ is the result of replacing S_q of S with a wildcard "?". Let V be a set of nodes of S. We define function $W^+(S, V)$ as: if V is empty, $W^+(S, V) = S$; otherwise, $W^+(S, V) = W^+(W(S, q), V - q)$ where $q \in V$. We define N_S as the set $\{W^+(S_q, V) | q \in S \wedge V \subseteq S_q\}$. Elements in set N_S are called *patterns* generated from S. The wildcard nodes in a pattern are also called *holes*. Figure 3.1 shows an example of an AST S, one of its subtree S_B, and a patterns $W^+(S_B, \{D, E\})$, which is generated by replacing node D and E of subtree S_B with wildcard nodes. Note that holes can only appear in the leaf node position of a pattern.

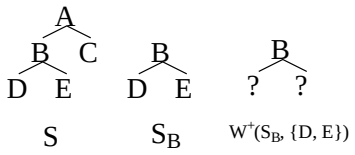

Figure 3.1: An Example of Pattern

Definition 3.1 (*Clone*)**.** Let S be an AST. Let P be a pattern generated from S. A clone is a one-to-one function C that maps each node of the

pattern P to a node of S such that for each non-wildcard node x of P:

1. x and $C(x)$ have the same label,

2. x and $C(x)$ have the same number of children, and

3. if x is not a leaf, then the i^{th} child of x maps to the i^{th} child of $C(x)$.

Note that a clone is an ordered matching. It is different from the ordered matching defined in Section 1.1.4 due to the requirement that x and $C(x)$ should have the same degree. Let C be a clone from P to S and q be the root of P. We call $C(q)$ an *occurrence* of P in S. C is called an *exact clone* if there is no hole in P; C is called a *lexical clone* if for each wildcard node w $C(w)$ is a leaf node of S; C is called a *structural clone* if it is not a lexical clone. Figure 3.2 shows an example of a clone, where node B_1 in S is an occurrence of the pattern. It is easy to see that it is a lexical clone.

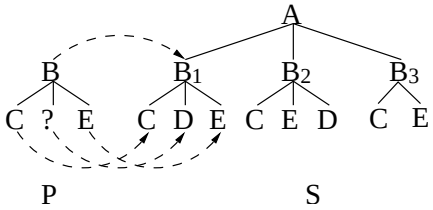

Figure 3.2: An Example of Clone

It is easy to see that the size of N_S is exponential to the size of S. In other words, there are too many possible patterns which can be generated from an AST. In most cases, we would like to consider only those patterns with multiple occurrences having large size and containing few holes, since clones corresponding to them are more meaningful. We define $N_S(Z, W)$ as

the set of patterns from N_S which have size greater than or equal to Z and contains less than or equal to W holes, where the size of a pattern is defined as the number of non-wildcard nodes in the pattern. Figure 3.3 shows an AST S and all patterns of set $N_S(3,1)$.

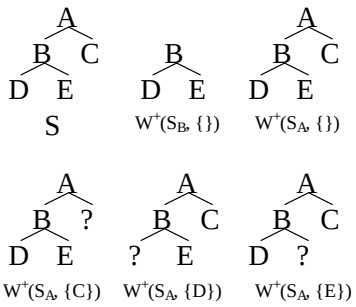

Figure 3.3: An Example of Patterns

Let P and P' be two patterns. Let C (resp. C') be a clone from P (resp. P') to S. We say P' is more general than P if P' is in N_P. We say C dominates C' if P' is more general than P and $C'(v')$ is in the subtree rooted at $C(v)$ where v (resp. v') is the root of P (resp. P'). A clone is a dominated clone if it is dominated by another clone. A pattern P is a dominated pattern if each clone from P to S is a dominated clone. Figure 3.4 shows an example of dominated clones, where the clone from P to S dominates the other two clones.

Given an AST S, and two parameters Z and W, the problem of *Clone Detection using Abstract Syntax Tree* is to find all multiply occurring, non-dominated patterns in $N_S(Z,W)$, and their corresponding occurrences in S.

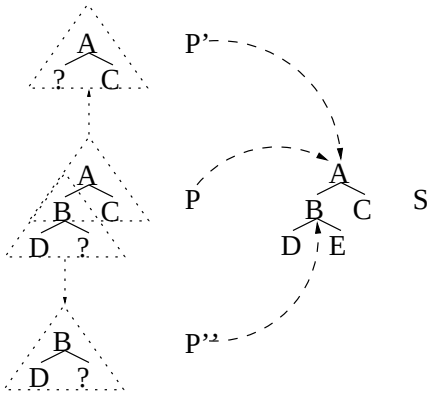

Figure 3.4: An Example of Dominated Clones

3.1.3 Chapter Overview

In Section 3.2 we present our first algorithm ASTA, which solves the problem of Clone Detection using Abstract Syntax Tree in the case that the AST can be held entirely in main memory.

If the AST is too large to be held in the main memory, we divide it into several modules such that each module fits in the main memory. We will use Algorithm ASTA to find clones within each module. In Section 3.3, we will present Algorithm ALL-CLONES to find cross-module clones, which assumes that the AST is stored as data streams on secondary storage, instead of in main memory.

In Section 3.4 we present the experimental results (with analysis) of applying our algorithms on a 440,000-line code corpus containing both Java and C# source codes.

3.2 In Memory Algorithm ASTA

Our first algorithm ASTA solves the problem of Clone Detection using Abstract Syntax Tree in the case that the AST can be held entirely in main memory. We explain the idea of Algorithm ASTA in plain English in this section.

3.2.1 Pattern generation

Algorithm ASTA first generates a set of candidate patterns that occur at least twice in S and have at most W holes.

Candidate generation starts by creating a set of simple patterns. Given an integer parameter D, ASTA generates, for each node v in S, at most D patterns called *caps*. The d-cap ($1 \leq d \leq D$) for v is the pattern obtained by taking the depth d subtree rooted at v and adding holes in place of all the children of nodes at depth d. If the subtree rooted at v has no nodes at depth d (i.e. the subtree has depth less than d) then node v has no d-cap. ASTA also generates a pattern called the *full cap* for v, which is the full subtree rooted at v. For example, if $D = 2$ and the subtree rooted at v is:

```
add(local(a),sub(local(b),formal(c)))
```

then ASTA will generate

1. `add(?,?)` (1-cap),
2. `add(local(?),sub(?,?))` (2-cap), and
3. `add(local(a),sub(local(b),formal(c)))` (full cap).

The set of all caps for all nodes in S forms the initial set, Π, of candidate patterns.

ASTA finds the occurrences of every cap by building an associative array called the *clone table*, indexed by pattern. Each entry of the clone table is a list of occurrences of the pattern in S. ASTA removes from Π any cap that occurs only once.

3.2.2 Pattern Improvement

After creating the set of repeated caps, ASTA performs the closure of the *pattern improvement* operation on the set. Pattern improvement creates a new pattern by replacing or "specializing" the holes in an existing pattern. Given a pattern P, pattern improvement produces a new pattern Q by replacing every hole v in P with a pattern $F(v)$ [6] such that (i) $F(v)$ has at most one hole (thus, Q has at most the same number of holes as P), and (ii) Q occurs wherever P occurs (i.e. $F(v)$ matches every subtree, from every occurrence of P, that fills hole v). It is possible that for some holes v, the only pattern $F(v)$ that matches all the subtrees is a hole. In this case, no specialization occurs for hole v.

In order to perform pattern improvement somewhat efficiently, we store with each node r in S a list of patterns that match the subtree rooted at r. The list is ordered by the number of nodes in the pattern in decreasing order. Given a pattern P to improve and a hole v in P, ASTA finds an arbitrary occurrence of P (with matching function f) in S and finds the list of patterns stored with the node $f(v)$. ASTA considers the patterns in this

[6] The notation emphasizes the fact that each hole may be filled with a different pattern.

list, in order, as candidates for $F(v)$. Any candidate with more than one hole is rejected (to satisfy condition (i)). In order to satisfy condition (ii), a candidate pattern must match the subtree rooted at $f(v)$ for all matching functions f associated with occurrences of P. Another way of saying this is that every node $f(v)$ (over all matching functions f from occurrences of P) must be the root of an occurrence of the candidate pattern. Thus ASTA looks up the candidate pattern in the clone table and checks that each $f(v)$ is the root of an occurrence in that table entry. (We actually store this list of occurrences as an associative array indexed by the root of the occurrence, so the check is quite efficient.)

ASTA repeats the pattern improvement operation on every pattern in Π, adding any newly created patterns to Π, until no new patterns are created.

3.2.3 Best-pair Specialization

Pattern improvement is a conservative operation. It only creates a more specialized pattern if it occurs in the same places as the original pattern. Some patterns can't be specialized without reducing the number of occurrences. We may still want to specialize these patterns because our focus is on finding large patterns that occur at least twice. best-pair specialization performs a greedy version of pattern specialization, called *best-pair specialization*, that attempts to produce large patterns that occur at least twice. It does this by performing pattern improvement but requires only that the specialization preserve two of the occurrences of the original pattern.

For each pair of occurrences, T_i and T_j $(1 \leq i < j \leq m)$ of a given pattern P with m occurrences, ASTA produces a new pattern Q_{ij} that is

identical to P except that every hole v in P is replaced by a pattern $F_{ij}(v)$ such that (a) $F_{ij}(v)$ has at most one hole, and (b) Q_{ij} matches T_i and T_j. The largest Q_{ij} (over $1 \leq i < j \leq m$) is the best-pair specialization of P. ASTA creates the best-pair specialization for every pattern P in the set of patterns, Π, and adds those patterns to Π. It then computes, again, the closure of Π using the pattern improvement operation.

3.2.4 Thinning and ranking

As the last step, Algorithm ASTA removes from Π all dominated patterns (defined in Sec. 1.2.2), and all patterns having size less than Z (i.e., the minimal clone size allowed). The output of ASTA is the rest patterns in Π and their corresponding occurrences in the clone table. The patterns may be ranked along several dimensions:

Size: Size is the number of nodes in the pattern, not counting holes.

Frequency: Frequency is the total number of occurrences of the pattern.

Similarity: Similarity is the size of the clone divided by the average size of its occurrence. If the clone has no holes, every occurrence is the same size as the clone and the similarity is 100%, Clones that take large subtrees as parameters have much lower similarity percentages.

3.2.5 Pattern with Internal Holes

According to our definition of pattern, wildcard nodes must replace a complete subtree. For example,

```
?(local(a),formal(b))
```

is not a valid pattern because it contains an internal hole, i.e., the wildcard replaces an operator but not the full subtree labeled with that operator. This restriction suits conventional programming languages (Pascal, C, and etc.), which do not easily support abstraction of operators. Languages (LISP, ICON, and etc.) with higher order functions do support such abstraction, so ASTA would ideally be extended to offer operator wildcards if it were used with ASTs from such languages.

Algorithm ASTA can be easily extended to offer operator wildcards. In the Pattern Improvement step (or Best-pair Specialization step), when no pattern can be specialized any more, a special pattern P' for each pattern P in Π will be created and then the specialization process will be restarted by using these new patterns. P' is created in the following way: if all occurrences of P have the same ordering and all parents of these occurrences have the same degree (let it be m), then P' will be created as a tree having a wildcard node as the root and having another m wildcard nodes as the children of the root; otherwise, no new pattern is created. The occurrences of the new pattern P' are the parents of the occurrences of P.

3.3 Data Stream Based Algorithm ALL-CLONES

Algorithm ASTA requires that the main memory is large enough to contain the given AST and all associated data structures. However, in case of a large AST, this requirement is not realistic. In our case, the size of an AST is normally four times the original source code size. For example, the complete JDK 1.4 source code is about 70 MB and the AST is about 300 MB. Including all the data structures, which are four times the size of the AST, to apply Algorithm ASTA on JDK 1.4, we need more than 1 GB of main memory. To reduce the cost of memory but still use Algorithm ASTA, we divide the large AST into modules, each of which can fit into the main memory. Then ASTA is applied to find clones within each module. In this section, we present Algorithm ALL-CLONES to find cross-module clones, assuming the AST is stored as data streams on secondary storage, instead of in main memory.

3.3.1 Notation and Data Structure

As long as each module can be handled in main memory, the division is arbitrary. In the example of JDK 5.0 source code, a module can be anything from one file to several directories. Figure 3.5 shows an example of modules, where each module is enclosed by dotted lines. Each module itself is also an AST. Let m be a module. We use function ROOT(m) to retrieve the root of m. Let q be a node in m. Function CHILDREN(q) returns the children of q. Function LABEL(q) returns the label of q.

The whole AST S is stored as data streams on secondary storage. The

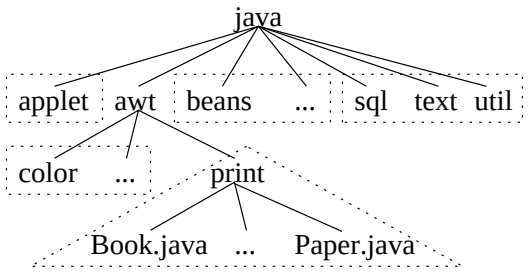

Figure 3.5: An Example of Modules

data streams are clustered by node labels, degree of the nodes, and level of the nodes. Each node q in S is associated with a set of virtual streams $\{T_q^\ell\}$ $(0 \leq \ell < H)$, where H is the height of S. T_q^ℓ is an image of the data stream in which all elements have the same label and same degree as q and have level value equal to ℓ. Each virtual stream T_q^ℓ has its own cursor pointing to the current element of a data stream and initially the cursor is pointing to the first element of the stream. Function CURRENT(T_q^ℓ) returns the current element of the stream T_q^ℓ, and function ADVANCE(T_q^ℓ) moves the cursor of T_q^ℓ forward by one element. Function EOF(T_q^ℓ) checks if the cursor is at the end of stream T_q^ℓ. Each element e in a data stream is a tuple $< start, end, ordering >$, where $e.ordering$ represents the position of e among all children of the parent of e.

We also maintain a clone table on secondary storage. Each entry of the table is a tuple $< P, B_P >$, where P is a pattern, and B_P is an array of H pointers. Pointer $B_P[\ell]$ $(0 \leq \ell < H)$ points to a buffer containing the occurrences of P that have level value equal to ℓ. The clone table is indexed by pattern. Function CONTAINS(Π, P) checks if pattern P is in

the table Π. Function ADD(Π, $< P$, $B_P >$) adds a new entry into the table. Function DELETE-FROM(Π, P) removes the entry corresponding to P from Π. Function CREATE-BUFFER() creates a new empty buffer and returns the pointer to it. Let $B_P[\ell]$ be a buffer. Function APPEND($B_P[\ell]$, E) appends a new occurrence E to buffer $B_P[\ell]$. Function LAST($B_P[\ell]$) returns the last occurrence in buffer $B_P[\ell]$. Function REMOVE($B_P[\ell]$) deletes the whole buffer $B_P[\ell]$.

3.3.2 Algorithm ALL-CLONES

Algorithm ALL-CLONES is based on Algorithm LEVEL-MATCH. Recall that Algorithm LEVEL-MATCH finds all root-matches from a pattern tree to the target tree. Let m be a module, N_m be the set of pattern generated from m, and P be a pattern in N_m. We apply Algorithm LEVEL-MATCH with P as the pattern tree and S as the target tree. Disregarding the difference between unordered matching and ordered matching, each root-match found by Algorithm LEVEL-MATCH is actually an occurrence of the pattern P in S. Thus, we could find all clones by applying Algorithm LEVEL-MATCH on every pattern in N_m and S. This is not very efficient, since there are possibly an exponential number of patterns in N_m.

Let q be a node in m and n be an ancestor of q. Let m_q (resp. m_n) be the subtree of m rooted at q (resp. n). Note that Algorithm LEVEL-MATCH is a recursive algorithm. Thus, during the process of finding q-matches (i.e., occurrences of pattern m_q), we have already found n-matches (i.e., occurrences of pattern m_n). Based on this observation, we modify Algorithm LEVEL-MATCH into Algorithm ALL-CLONES (Alg. 3.1), such that given two

patterns P and P', if P dominates P', then the process of finding occurrences of pattern P' is embedded in the process of finding occurrences of pattern P. In this way, we can just scan the streams of S once for each module m. In addition to the above change, in Algorithm 3.1 we also check the ordering and degree of nodes such that matchings found obey the definition of clones (ordered matching).

Algorithm 3.1 Algorithm ALL-CLONES

```
CLONE-DETECTION(S, Z, W)
1  for each module of S
2      ALL-CLONES(ROOT(module))
3  CLEAN-UP()

ALL-CLONES(q)
1  for (ℓ ∈ [0, H))                                  ▷ H is the height of T
2      while (¬EOF(T_q^ℓ))
3          OUTPUT-CLONE(CURRENT-CLONE(q, ℓ), ℓ)
4          ADVANCE(T_q^ℓ)
5  for (n ∈ CHILDREN(q))
6      ALL-CLONES(n)

OUTPUT-CLONE(< P, E >, ℓ)
1  if (|P| < Z) return

2  if (CONTAINS(Π, P))
3      for (ℓ ∈ [0, H))
4          B_P[ℓ] ← CREATE-BUFFER()
5      ADD(Π, < P, B_P >)
6      APPEND(B_P[ℓ], E)
7      return

8  if (LAST(B_P[ℓ]).start < E.start)
9      APPEND(B_P[ℓ], E)

CLEAN-UP()
1  for (P ∈ Π)
2      if (|B_P| < 2)
3          for (ℓ ∈ [0, H))
4              REMOVE(B_P[ℓ])
5          DELETE-FROM(Π, P)

NUM-HOLES(P)
    ▷ Return the number of holes in pattern P
```

Algorithm 3.1 Algorithm ALL-CLONES *(continued)*

```
CURRENT-CLONE(q, ℓ)
 1  for (n ∈ CHILDREN(q))
 2      while (CURRENT(T_n^{ℓ+1}).start < CURRENT(T_q^ℓ).start)
 3          OUTPUT-CLONE(CURRENT-CLONE(n, ℓ+1), ℓ+1)
 4          ADVANCE(T_n^{ℓ+1})

 5      while (CURRENT(T_n^{ℓ+1}).start < CURRENT(T_q^ℓ).end)
 6          if (CURRENT(T_n^{ℓ+1}).ordering = n.ordering)
 7              C[n] ← CURRENT-CLONE(n, ℓ+1)
 8              ADVANCE(T_n^{ℓ+1})
 9              break

10          OUTPUT-CLONE(CURRENT-CLONE(n, ℓ+1), ℓ+1)
11          ADVANCE(T_n^{ℓ+1})

12      if (CURRENT(T_n^{ℓ+1}).start > CURRENT(T_q^ℓ).end)
13          C[n] ←<<?, 0 >, CURRENT(T_n^{ℓ+1}) >

14  holes ← 0
15  for (n ∈ CHILDREN(q))
16      holes ← holes +NUM-HOLES(C[n].pattern)

17  pattern ←< LABEL(q), |CHILDREN(q)| >

18  if (holes > W)
19      for (n ∈ CHILDREN(q))
20          OUTPUT-CLONE(C[n], ℓ+1)
21  else
22      for (n ∈ CHILDREN(q))
23          pattern ← pattern + C[n].pattern

24  return < pattern, CURRENT(T_q^ℓ) >
```

3.3.3 Analysis of Correctness and Complexity

If there is no constraint on the number of holes in a pattern, Algorithm ALL-CLONES can be proved correct. The proof is similar to that of Algorithm LEVEL-MATCH. If we limit the number of holes in a pattern to be at most W, Algorithm ALL-CLONES may output dominated patterns or miss non-dominated patterns, because of the greedy decision we made in line 18 of function CURRENT-CLONE(q, ℓ). We explain our greedy decision and its reason using the example shown in Figure 3.6, where P_q is the pattern and E_q is the corresponding occurrence. Suppose $W = 10$, since P_q has 16 holes, we cannot output P_q and E_q. We should output all the 12 patterns $\{P_q^1 \dots P_q^{12}\}$, each of which has E_q as its occurrence. Instead, we output the 4 patterns $\{P_{n_1} \dots P_{n_4}\}$ and each pattern P_{n_i} has occurrence E_{n_i}. It is hard to see from this example the advantage of outputting 4 patterns instead of 12 patterns. However, suppose the node q in pattern P_q has 10 children $\{n_1 \dots n_{10}\}$, and each child of q has 10 wildcard children. Pattern P_q now has 100 holes. Suppose $W = 55$, then we cannot output P_q. It is easy to see that there will be $\binom{10}{5} = 252$ non-dominated patterns with occurrence E_q. Instead of outputting all these 252 patterns, Algorithm ALL-CLONES will greedily output the 10 patterns corresponding to the children $\{n_i\}$ of q, each of which has occurrence E_{n_i}.

Both the I/O and CPU costs of Algorithm ALL-CLONES (except the cost of operations on the clone table) are $O(\sum_{q \in S} |T_q|)$. Let the number of patterns in the clone table be $|CT|$. If the clone table uses a B-tree index, each insertion or searching operation will cost $O(\log |CT|)$. Then, the total

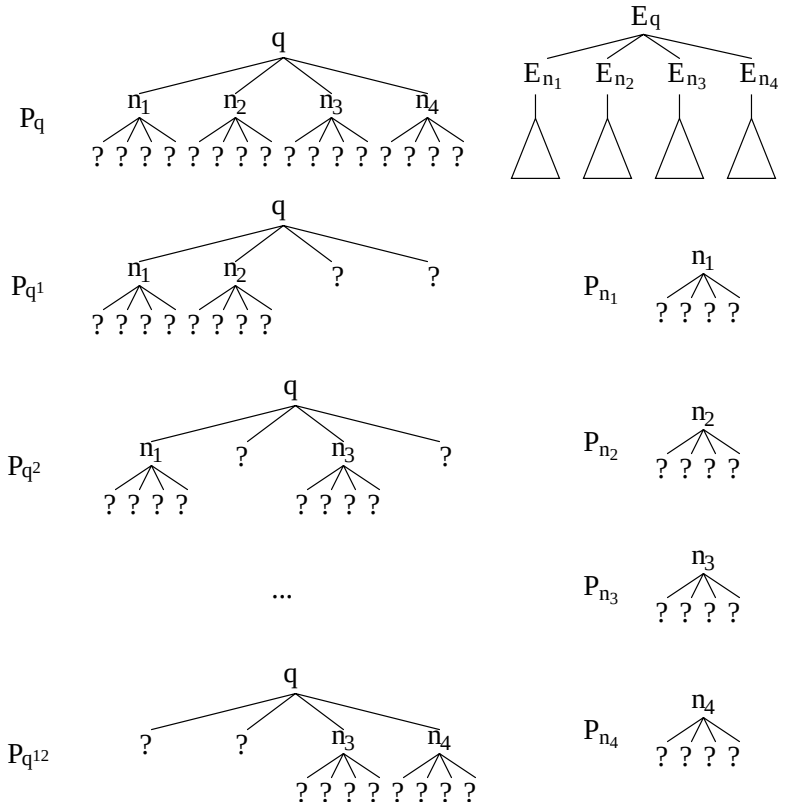

Figure 3.6: An example of greedy decision

I/O and CPU costs of Algorithm ALL-CLONES are $O(\log |CT| \cdot \sum_{q \in S} |T_q|)$. It is hard to give an analytical bound on $|CT|$. Let m be an arbitrary module of S, the space complexity is $O(|m|)$.

3.4 Experiments

3.4.1 Datasets

The ASTs used for our algorithms are created by JavaML from Java code [3] and with ASTs created by the C# compiler lcsc[14]. A custom back end for JavaML and lcsc emits each file as a single AST. A simple tool combines multiple ASTs into a single XML string to find clones across multiple files.

The ASTs are easily pretty-printed to reconstruct a source program that is very similar to the original input. The ASTs are also annotated with pointers to the associated source code. There are thus two different ways to present AST clones to the programmer in a recognizable form.

We have run Algorithm ASTA and Algorithm ALL-CLONES on a corpus of 1,141 Java programs (from the Java 2 platform, standard edition (version 1.4.2)'s `java` directory[7]) and 58 C# programs (mostly from the lcsc compiler [14]). Figures 3.7 and 3.8 give their sizes.

For each file (ordered by number of AST nodes along the x-axis), the figures show the number of nodes, characters, tokens, and lines. These measures can be sensitive to design or formatting issues that merit elaboration. Node counts are easy to compute but depend somewhat on the design of the abstract syntax tree's node types. Fortunately, both JavaML and lcsc's abstract syntax closely matches the source language definition and introduces little noise.

The other three measures count elements of the source code. Our ASTs are labeled with line numbers and character positions within each line,

[7] http://java.sun.com/j2se/1.4.2/download.html

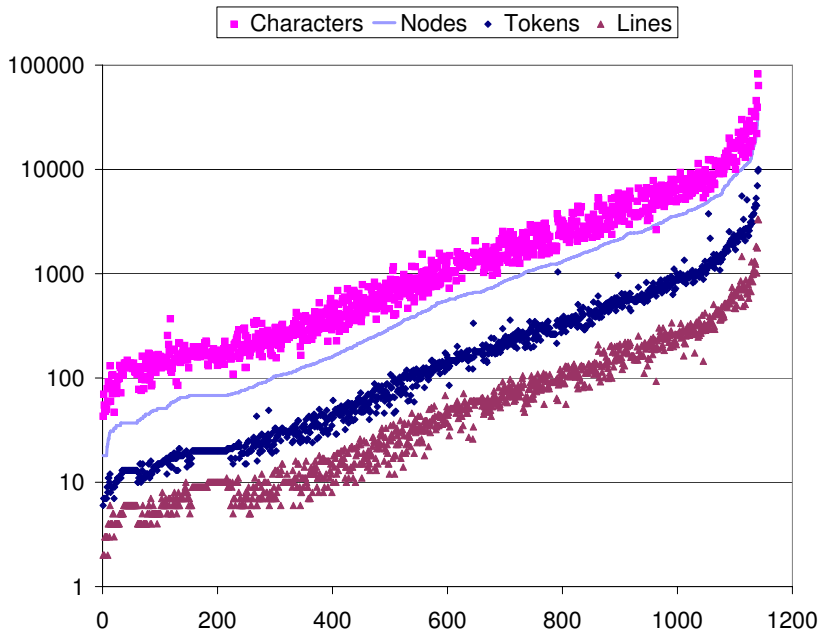

Figure 3.7: Java source file metrics

though a few nodes are synthesized (such as the null nodes that end lists of statements and parameters), so coordinates must be inferred for them. Also, line and character counts are sensitive to white space, comments, and the fact that some programmers use longer variable names than others. Our algorithms partially compensate by discounting blank lines and comments, though this discounting is an approximation. The token counter gets accurate data from the ASTs and is thus the only source-based measure that is immune to formatting issues.

Multiple measures raise the question of which to use. Line counts are the most commonly quoted measure of savings, but character counts better reflect the source size, and token counts better reflect the number of primi-

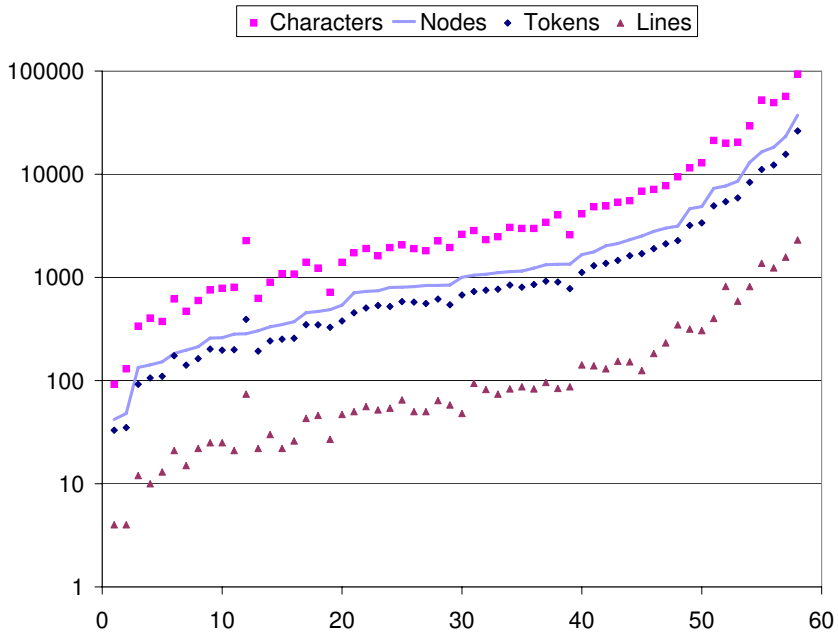

Figure 3.8: C# source file metrics

tive elements in the program. For example, is one clone covering some long variable names really any better than the same clone covering short variable names?

Figures 3.7 and 3.8 show that, at least for our samples, node counts are roughly linear in the number of characters, tokens, and lines. Lines and characters show a few more outliers, but tokens track nodes well. This finding is consistent with the fact that characters and lines involve more formatting issues, which are peripheral to the issue of cloning.

In what follows, we will use node counts as a proxy for size of source code, avoiding measures that are more influenced by formatting.

3.4.2 Output Format

Our algorithms produce a list of clones as an HTML document with three parts: a table with one row per pattern, a list of patterns with their occurrences, and the source code. Each part hyperlinks to an elaboration in the next part. For example, Figure 3.9 shows most of the report for an eight-queens solver written in C# (source code is shown in Appendix A). The first row of the table presents the top-ranked clone:

```
rows[r]=up[r-c+7]=down[r+c]=?(2);
```

That text is a hyperlink to the second section, which shows all occurrences of the clone. The second section points, in turn, into the third section, which holds the source code for the original input (elided from Figure 3.9 to save space), so that the occurrences can be viewed in context.

The first column of the table gives the percentage of similarity. Columns 2 and 3 of the table report the number of nodes (excluding holes) and the number of holes in the clone. Column 4 gives the number of occurrences. The top-ranked clone in Figure 3.9 has 40 nodes and one hole or, equivalently, one formal parameter. It has two occurrences, which appear in the second section. One occurrence takes `true` as its actual argument; the other takes `false`.

In patterns, `?(n)` flags a hole and gives the number of nodes in the largest argument tree that occupies that hole. For example, the second row abstracts two simple loops with bodies that comprise a single assignment. The `?(11)` reports eleven nodes in the largest right-hand side in either of the two occurrences. The ASTs used in this paper typically have `?(1)` or `?(2)`

%sim	nodes	holes	hits	Source
0.952	40	1	2	`rows[r] = up[r-c+7] = down[r+c] = ?(2);`
0.607	24	3	2	`for (int i = 0; i < ?(1) ; i++)` `  ?(8) = ?(11);`
0.122	11	3	2	`?(1) static void ?(1) () { ?(92) }`
0.916	11	1	2	`?(1) = new Boolean[15]`

Patterns and occurrences:

- `rows[r] = up[r-c+7] = down[r+c] = ?(2);`
  - `rows[r] = up[r-c+7] = down[r+c] = true;`
  - `rows[r] = up[r-c+7] = down[r+c] = false;`
- `for (int i = 0; i < ?(1) ; i++) ?(8) = ?(11);`
  -
```
for (int i = 0; i < 15; i++)
  up[i] = down[i] = true;
```
  -
```
for (int i = 0; i < 8; i++)
  rows[i] = true;
```
- `?(1) static void ?(1) () { ?(92) }`
  -
```
private static void print() {
  for (int k = 0; k < 8; k++)
    Console.Write("{0} ", x[k]+1);
  Console.Write("\n");
}
```
  -
```
public static void Main() {
  for (int i = 0; i < 8; i++)
    rows[i] = true;
  for (int i = 0; i < 15; i++)
    up[i] = down[i] = true;
  queens(0);
}
```
- `?(1) = new Boolean[15]`
  - `down = new Boolean[15]`
  - `up = new Boolean[15]`

Figure 3.9: Sample clone report

for identifiers and constants, which lexical abstraction can catch. Larger values denote holes that require more general structural abstraction.

Figure 3.9 illustrates both pros and cons of structural abstraction. The pattern in the first row catches an important bit of logic from eight queens and arguably merits refactoring. The other rows are less promising and appear as a result of low thresholds. The second row abstracts a specialized loop, but the repeated pattern is smaller (only 24 nodes and thus roughly 1.5 lines) and the similarity percentage is small. The third row is small (only 11 nodes) and has a very small similarity percentage; a threshold on similarity is indicated. The last row improves similarity but not size. A macro would scarcely be worth the effort. We could clearly run our algorithms with higher thresholds to eliminate these poor clones but we wanted to illustrate the need for them here.

3.4.3 Measured Clones

Our primary goal is to report a list of clones that merit procedural abstraction, refactoring, or some other action. What merits abstraction is a subjective decision that is difficult to quantify. It is therefore difficult to quantitatively measure how well a system achieves this goal. Historically, research in clone detection (procedural abstraction) used the number of source lines (or instructions) saved after abstraction as a measure of system performance. This goal is easy to quantify. A clone with p elements (lines, tokens, characters, or nodes) and h occurrences saves $p(h-1)$ elements[8]. Subtracting one accounts for the one copy of the clone that must remain.

A focus on savings tempts one to use a greedy heuristic that chooses clones based on the number of, for example, source lines they save. The clones that result may not be the ones that subjectively merit abstraction. For example, the clone that saves the most source lines in `8q.cs` (Figure 3.9) is the rather dubious:

```
for (int i = 0; i < ?; i++)
    ? = ?;
```

To our eyes, reporting clones based on the number of nodes in the clone itself (rather than the number in all occurrences) produced better clones, at least from the point of view of manual refactoring. Whenever our ranking factored in number of occurrences, we tended to see less attractive clones. However, it may be that the purpose of performing clone detection is, in fact,

[8]This does not consider the cost of the $h-1$ call instructions that replace $h-1$ of the occurrences.

to compact the source code via procedural abstraction. For that application, small, frequent clones are desirable.

We explore both our primary goal of finding clones that merit abstraction and the historical goal of maximizing the number of source lines saved after abstraction. The first goal we equate with finding large clones (with many nodes). To accomplish this, we rank clones by size (number of nodes) and report the size of the non-overlapping clones that we find (Figures 3.10 and 3.11). The second, historical goal, we approach by ranking clones by the number of nodes saved and report the percentage of nodes saved after abstraction (Figure 3.12). In both cases, we select, in a greedy fashion, those clones that (locally) most increase the measure (eliminating from future consideration the clones they overlap).

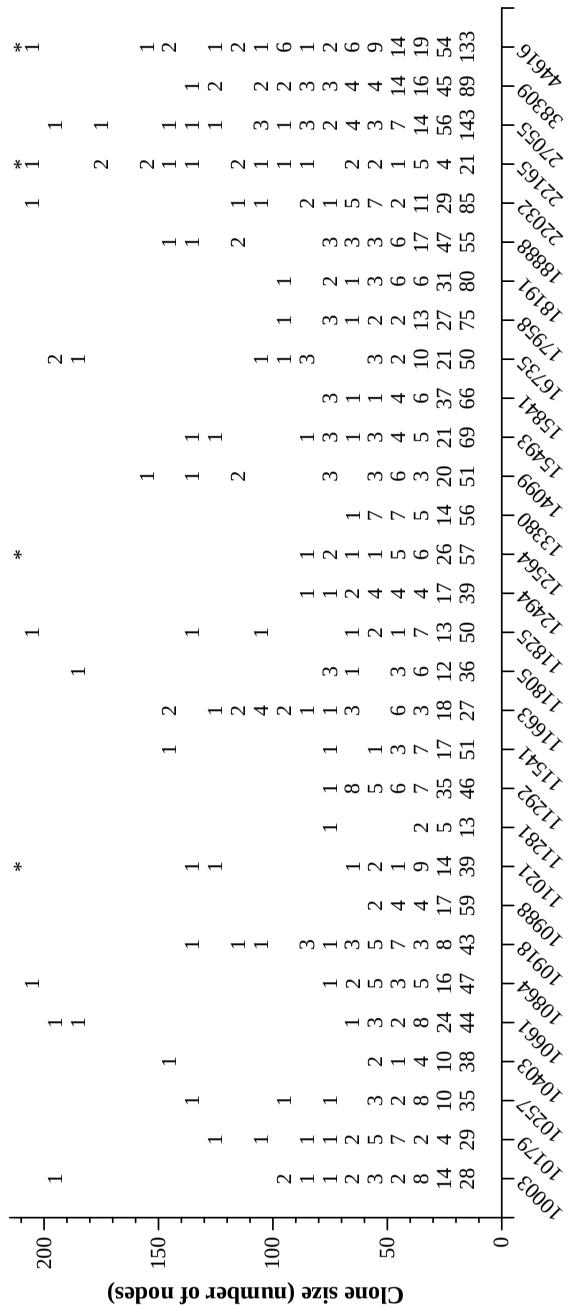

Figure 3.10: Number of non-overlapping clones in Java source files. For example, the 54 in the rightmost column indicates that the largest source file (44,616 nodes) has 54 non-overlapping clones, each with 20-30 nodes. An asterisk indicates a clone whose size is off the scale. (The maximum size clone has 913 nodes.)

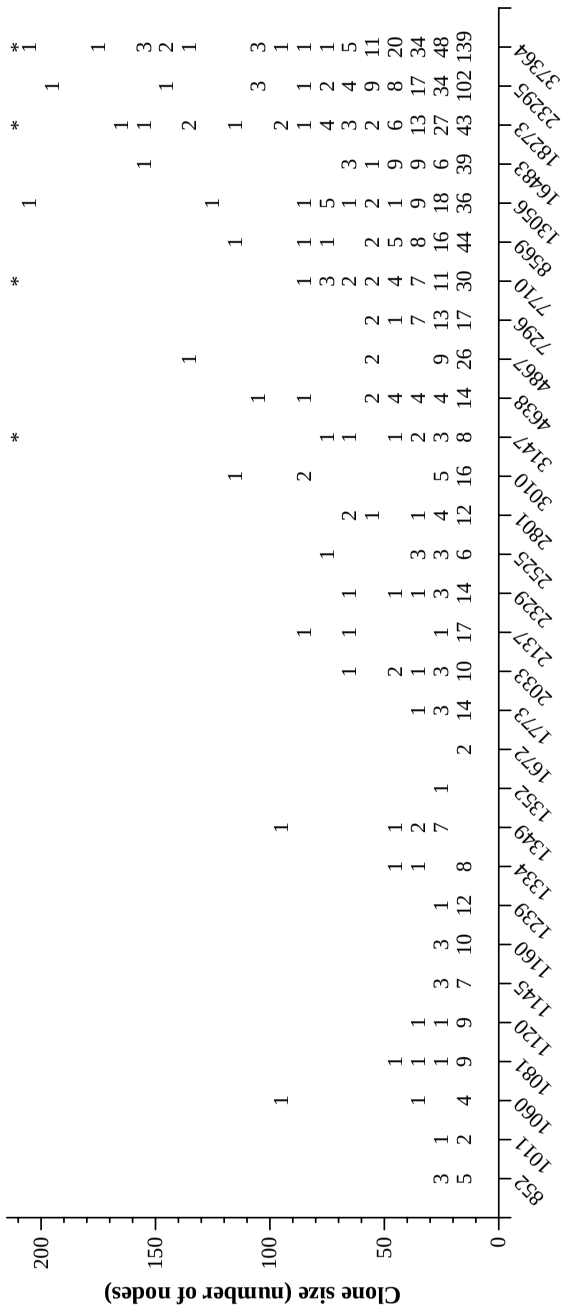

Figure 3.11: Number of non-overlapping clones in C# source files. For example, the 48 in the rightmost column indicates that the largest source file (37,364 nodes) has 48 non-overlapping clones, each with 20-30 nodes. An asterisk indicates a clone whose size is off the scale. (The maximum size clone has 605 nodes.)

Figures 3.10 and 3.11 show the numbers of non-overlapping clones of various sizes found in the largest files of the Java and C# corpora. There are many small clones but also a significant number that merit abstraction.

We hand-checked all 48 clones of at least 80 nodes in the C# examples, and found that 44 represent copying that we would want to eliminate. This high success ratio suggests that many of the smaller clones should also be actionable. The number of significantly smaller clones prohibits grading by hand, but skimming suggests that a 40-node threshold gives many actionable clones and that a 20-node threshold is probably too low, just as 80 is too high.

The size of actionable Java clones is similar. A sampling of 40-node clones revealed many useful clones, while many 20-node clones are too small to warrant abstraction.

We now consider the historical goal of maximizing the number of nodes saved by abstraction. Reporting total savings is complicated by the fact that it varies significantly with the threshold on clone size. Figure 3.12 shows that, for our corpus, the total savings drops from 24% to 1% as the threshold for clone size increases from 10 to 160 nodes. If maximizing total savings is our goal, we should allow the automatic abstraction of small clones, even though these clones may not be large enough to merit abstraction by hand. If we would rather avoid abstracting small clones, thresholds between 20 and 80 nodes eliminate many of the small, dubious clones and still yield savings of 4-16%.

We should emphasize that our results represent the execution of ASTA on each individual file in isolation. If instead, we use ALL-CLONES to find clones

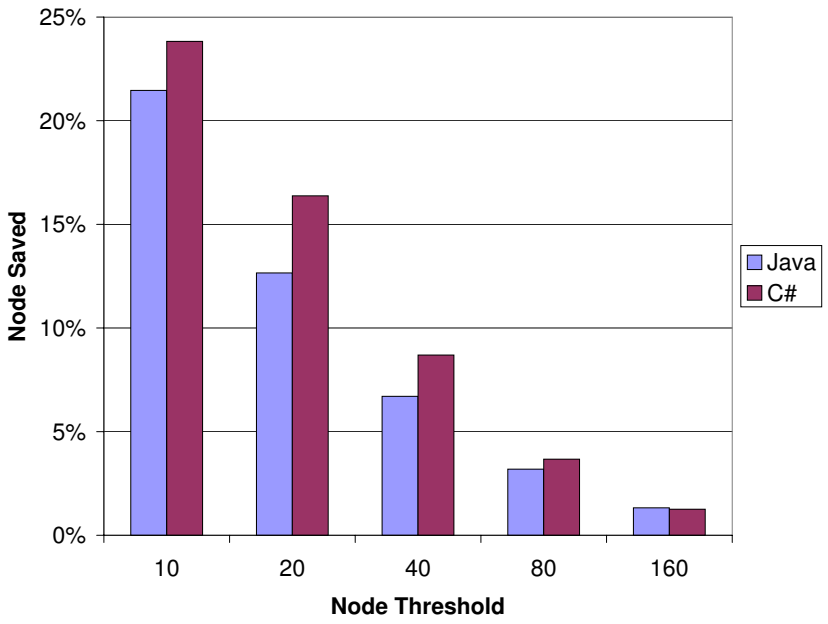

Figure 3.12: Total savings for various node thresholds

that occur in multiple files, we obtain greater savings. Figure 3.13 shows the difference for the Java corpus. The savings across multiple modules is obtained by finding clones that occur anywhere within the approximately 400,000 lines of Java source.

By comparison, Baker [4] reports saving about 12% by abstracting clones of at least 30 lines in inputs with 700,000 to over a million lines of code; she reports that most 20-line clones are actionable and that most 8-line clones are not. Baxter et al. also report saving roughly 12% on inputs of about 400,000 lines of code; they too use a threshold but do not specify what its value is.

Our threshold of 20 nodes is far smaller than Baker's 30-line thresh-

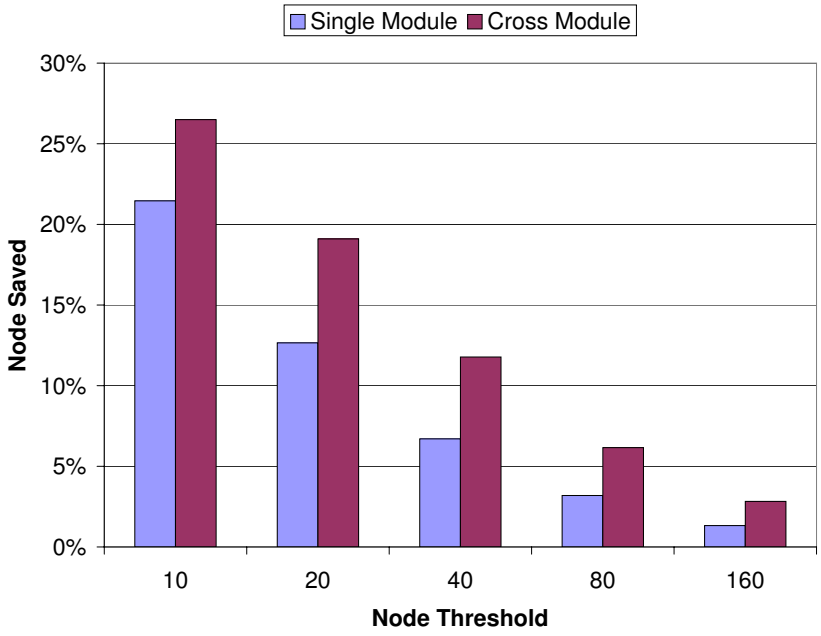

Figure 3.13: Total savings from finding clones within each single module versus across all modules

old. That we still observe mostly actionable clones at this threshold may be understood as a difference in the definition of *actionable*, or as a difference in the corpora, source languages, or abstraction mechanism. Our smaller threshold is matched by our smaller input sizes: our largest module contains about 45,000 lines of source. As mentioned, we can apply our techniques across multiple modules (as shown in Figure 3.13), but there is also redundancy and duplication within individual files.

Remarkably, the savings we obtain by abstracting actionable clones in isolated files is roughly the same as that obtained by both Baker and Baxter et al. This is somewhat disappointing since our system finds clones based

not only on lexical abstraction (as in Baker and Baxter et al.) but also on structural abstraction. Either there are very few clones that are purely structural in nature, or individual files contain fewer clones (that we view as actionable) than the large corpora examined by Baker and Baxter et al. The following section makes the case for the latter interpretation.

3.4.4 Lexical versus structural abstraction

Prior clone detection algorithms are based on lexical abstraction, which abstracts lexical tokens. Structural abstraction can abstract arbitrary subtrees and thus should be expected to find more clones. One objective of this research has been to determine if this generality translates into practical benefit and, if so, to characterize the gain.

In JavaML's and lcsc's ASTs, identifiers and literals appear as leaves but, depending on context, can be wrapped in or hung below one or more unary nodes. We classify arguments or holes conservatively: if an argument is a leaf or a chain of unary nodes leading to a leaf, then we count it as a lexical abstraction. Only more complex arguments are counted as structural abstractions. For example, suppose the clone `a[?] = x;` occurs twice:

```
 a[i] = x;
a[i+1] = x;
```

The argument to the first occurrence is lexical because it includes only a leaf and, perhaps, a unary node that identifies the type of leaf. The argument to the second occurrence is, however, structural because it includes a binary AST node.

The HTML output optionally shows the arguments to each occurrence of each clone, and it classifies each argument as lexical or structural. Because our algorithms can generate clones that a human might reject, we checked a selection of C# source files by hand. Figure 3.11 includes 48 clones of 80 or more nodes. 32 were structural and 16 were lexical. 28 of the structural clones and all of the lexical clones were deemed useful. Thus a significant fraction of these large clones are structural, and most of them merit abstraction.

There are, of course, too many clones to check all of them by hand, so we present summary data on the ratio of structural to lexical clones. This ratio naturally varies with the thresholds for holes and clone size.

First, fixing the hole threshold at 3 and raising the node threshold from 10 to 160 gives the results in Figure 3.14. As the threshold on clone size rises, our algorithms naturally find fewer clones, but note that structural clones account for an increasing fraction of the clones found.

If, instead, we vary the hole threshold, we obtain Figure 3.15, in which the node threshold is fixed at 10 and the hole threshold varies from zero to five. The ratio of structural to lexical clones rises because each additional hole increases the chance that a clone will have a structural argument and thus become structural itself.

Clones with zero holes are always lexical because they have no arguments at all, much less the complex arguments that define structural clones. Predictably, the number of structural clones grows with the number of holes. At the same time, the number of lexical clones may decline slightly because some of the growing number of structural clones out-compete some of the

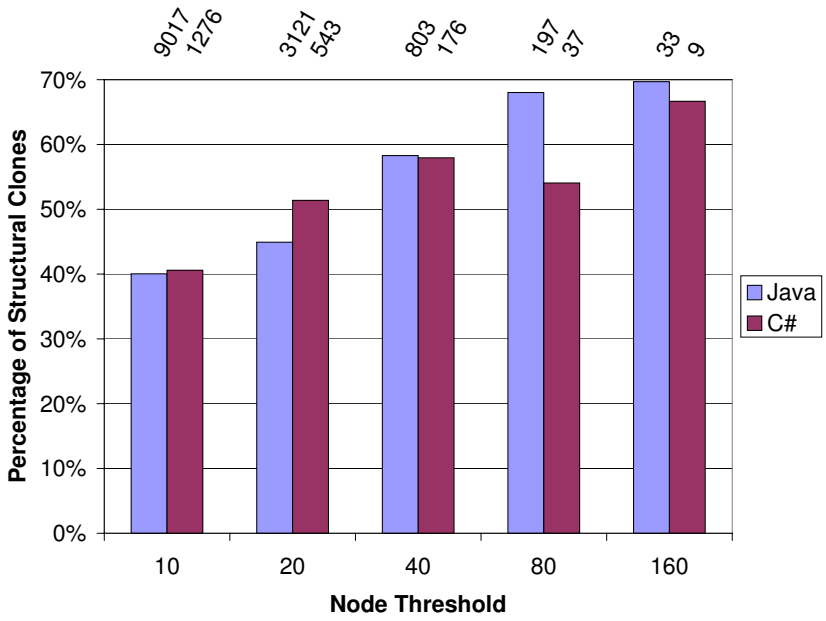

Figure 3.14: Lexical and structural clones for various node thresholds. The number above each column denotes the total number of clones for given node threshold.

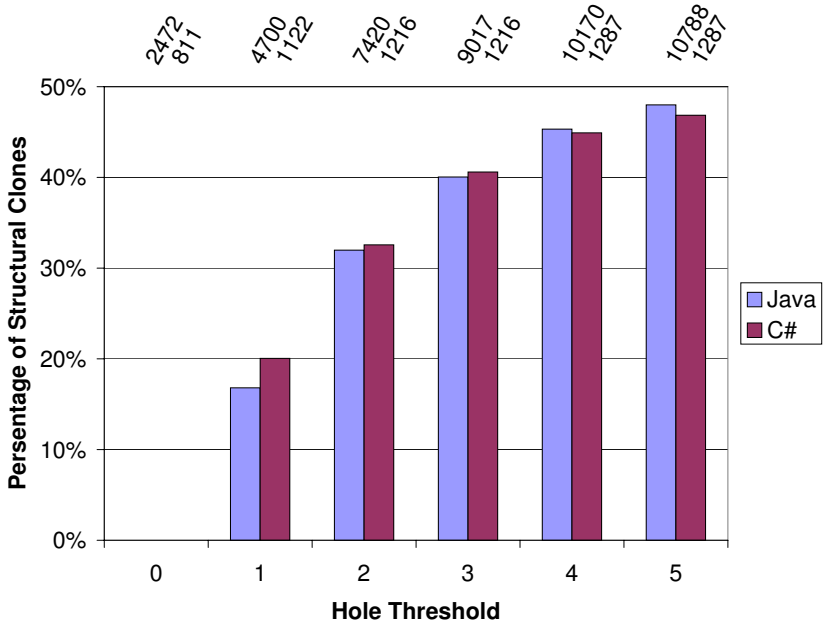

Figure 3.15: Lexical and structural clones for various hole thresholds. The number above each column denotes the total number of clones for given hole threshold.

lexical clones in the rankings.

Clones with fewer holes are generally easier to exploit—just as library routines with fewer parameters are easier to understand and use—but even one-parameter macros miss roughly 20% of the opportunities without structural abstraction, which is significant. Optimizations are deemed successful with much smaller gains, and improving source code is surely as important as improving object code. Figures 3.14 and 3.15 explore a large range of the configuration options that are most likely to be useful, and they show significant numbers of structural clones for all of the non-trivial settings.

3.5 Future Work

Experiments using Algorithm ASTA show that there are not many meaningful clones with internal holes within each module, if we use ASTs generated by JavaML and lcsc. We are trying to extend Algorithm ALL-CLONES so that it can find clones with internal holes across all modules. One possible solution is to store with each stream element the pointer to its parent.

Our structural abstraction method can benefit from variable renaming (a technique described by Baker [5]) since variables that can be named consistently in all clone occurrences no longer need to be represented as holes in the clone. This means these clones save more when abstracted as procedures. Preliminary experimental results show an extra savings of about 20% for our Java corpus when combining structural abstraction with variable renaming.

Chapter 4

Conclusion

In this thesis we have studied Tree Pattern Query problem and its variations. We have given one algorithm for each TPQ variation. All the TPQ algorithms presented are both sound and complete. To the best of our knowledge, there is no existing algorithm to solve Tree Pattern Query with parent-child edges, even if the query is parent-child edge only. Furthermore, our algorithms have better time complexity than any existing algorithms.

We have also designed, implemented, and experimented with two new methods for detecting cloned code, one of which is based on our algorithm for PC-edge-only TPQ problem. Heretofore, abstraction parameterized lexical elements such as identifiers and literals. Our method generalizes these methods and abstracts arbitrary subtrees of an AST. We have shown that the new method is affordable and finds a significant number of clones that are not found by lexical methods.

Bibliography

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152. IEEE Computer Society, 2002.

[2] Alberto Apostolico and Zvi Galil, editors. *Pattern matching algorithms*. Oxford University Press, Oxford, UK, 1997.

[3] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[5] Brenda S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal of Computing*, 1997.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In

Proceedings of the International Conference on Software Maintenance, pages 368–377, 1998.

[7] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 310–321. ACM, 2002.

[8] D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal, 41(4):597–615*, 2002.

[9] Warren Cheung, William Evans, and Jeremy Moses. Predicated instructions for code compaction. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, pages 17–32, 2003.

[10] Yun Chi, Siegfried Nijssen, Richard R. Muntz, and Joost N. Kok. Frequent subtree mining–an overview. *Fundamenta Informaticae*, 66(1–2):161–198, March 2005.

[11] Mariano P. Consens and Tova Milo. Optimizing queries on files. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 301–312. ACM Press, 1994.

[12] Mariano P. Consens and Tova Milo. Algebras for querying text regions. In *PODS*, pages 11–22. ACM Press, 1995.

[13] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, , 1998.

[14] David R. Hanson and Todd A. Proebsting. A research C# compiler. *Software-Practice and Experience*, 34(13):1211–1224, Nov. 2004.

[15] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the Eighth International Symposium on Static Analysis*, pages 40–56, 2001.

[16] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3:77–108, 1996.

[17] Fei Ma. Context search with summary trie on XML, 2005. CPSC 534B Project Report, University of British Columbia.

[18] Fei Ma, William S. Evans, and Christopher W. Fraser. Clone detection via structural abstraction, 2006. submitted.

[19] Gerard Salton and Michael McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1984.

[20] David Seal, editor. *ARM Architecture Reference Manual.* Addison-Wesley, second edition, 2001.

[21] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, August 2002.

[22] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

Appendix A

Source code for eight-queens example

```
using System;

public class Queens {
  static Boolean[] up = new Boolean[15],
    down = new Boolean[15],
    rows = new Boolean[8];
  static int[] x = new int[8];

  public static void Main() {
    for (int i = 0; i < 8; i++)
      rows[i] = true;
    for (int i = 0; i < 15; i++)
      up[i] = down[i] = true;
    queens(0);
  }
```

```
  private static void queens(int c) {
    for (int r = 0; r < 8; r++)
      if (rows[r] && up[r-c+7] && down[r+c]) {
        rows[r] = up[r-c+7] = down[r+c] = false;
        x[c] = r;
        if (c == 7)
          print();
        else
          queens(c + 1);
        rows[r] = up[r-c+7] = down[r+c] = true;
      }
  }

  private static void print() {
    for (int k = 0; k < 8; k++)
      Console.Write("{0} ", x[k]+1);
    Console.Write("\n");
  }
}
```