ABSTRACT
A fundamental problem in program analysis and optimization concerns the discovery of structural similarities between different sections of a given program and/or across different programs. Specifically, there is a need to find topologically identical segments within compiler intermediate representations (IRs).

Such topological isomorphism has many applications. For example, finding isomorphic sub-trees within different expression trees points to common computational resources that can be shared when targeting application-specific hardware. Isomorphism in the control-flow graph can be used to discovery of custom instructions for customizable processors. Discovering isomorphism in context call trees during program execution is invaluable to several JIT compiler optimizations. Thus, all these different applications rely on the fundamental ability to find topologically identical segments within a given tree or graph representation.

In this paper, we present a generic formulation of the subtree isomorphism problem that is more powerful than previous proposals. We prove that an optimal quadratic time solution exists for this problem. We employ a dynamic programming based algorithm to efficiently enumerate all isomorphic sub-trees within given reference trees and also demonstrate its efficacy in a production compiler.

Categories and Subject Descriptors
C.3 [Special-purpose and Application-based Systems]: Real-time and Embedded Systems

General Terms
Algorithms, Design

Keywords
Subtree Isomorphism, Subtree Matching Algorithm, Embedded Systems

1. INTRODUCTION

Two subgraphs of a given set of graphs can be said to be isomorphic when there is a one-to-one correspondence between their nodes and edges in terms of some chosen attributes. The problem of subgraph isomorphism is quite ubiquitous in the domain of compilers, hardware synthesis and embedded systems. However, this problem is known to be NP-complete [14]. Therefore, in practice, these graphs are transformed into trees by replicating nodes having multiple fanouts in order to obtain a reasonable solution [14, 3]. Thus, the problem of subgraph isomorphism is often translated into the simpler problem of subtree isomorphism.

In the domain of compilers and architectures, the subtree isomorphism problem is known to appear in instruction selection [9], where trees representing ISA instructions are matched against the compiler’s intermediate representation (IR). Most IRs are directed acyclic graphs (DAGs) to begin with, but can be converted in to a tree representation. Furthermore, the subtree isomorphism problem can also be effectively applied to common subexpression elimination and code reuse optimizations.

In the hardware synthesis domain, technology mapping [10] is a well-known problem, where tree patterns from a given standard cell library are matched against a given abstract netlist of the hardware circuit. In the domain of behavioral synthesis, resource sharing is an important problem that aims to maximize the use of arithmetic and logic operators like adders or multipliers. However, a more sophisticated approach can be taken whereby an entire subtree pattern, instead of a single operation, is chosen for sharing resources. Finding isomorphic subtrees is useful for all these synthesis-related problems.

In the last few years, application-specific customizable processors of different flavors have been proposed as viable solutions for meeting the rapidly changing demands of applications in embedded systems. One of the key problems in such systems is automatic generation of instruction set extensions (ISEs) [5]. Increasing the effectiveness and reusability of the chosen ISEs requires the identification of isomorphic subtree patterns of instructions in the IR, such that a frequently occurring subtree isomorph can become a custom instruction in the ISE.

Thus, we can see that finding an optimal solution to the subtree isomorphism problem would have a great impact in several domains. In this paper, we propose a novel algorithm to enumerate all isomorphic subtrees occurring within two given trees. We formally prove that the proposed algorithm is optimal in that it is guaranteed to enumerate all subtree matches and that the algorithm has quadratic time and space complexity. This is the main contribution of the paper.

Consider two reference trees (shown in Fig. 1), labeled as T1 and T2. The shaded subtrees in T1 and T2 represent an example of isomorphic subtrees that can be matched. The enumeration process of the isomorphic subtrees must also take care of some key properties of a topological match. First, for some nodes, the children
are unordered; so a match may be formed by flipping the children, as is the case for the \( \times \) operation in Fig. 1. Second, the common subtree match may exist anywhere in the two reference trees. Several existing subtree isomorphism problems define a match only if the matched subtrees share the same root with the reference trees or the matched subtrees share all their leaves with the reference trees [15, 14, 7, 2]. Thus, the general problem we address in this paper is a superset of the subtree isomorphism problems addressed previously.

The rest of the paper is organized as follows. In the next section, we compare our algorithm with the state of the art. Section 3 formally describes the problem statement. In Section 4, we present our isomorphic subtree enumeration algorithm and then prove its optimality and complexity. In Section 5, we explore its implications when applied on real-world applications and finally conclude in Section 6.

2. RELATED WORK

Subtree isomorphism is a general graph theory problem and has been employed in several domains like data mining, circuit synthesis, compiler code generation, etc. The most conventional form of the problem is the pattern matching variety—given a finite set of pattern tiles, it is the problem of finding all isomorphic subtrees that match one of the these tiles in a reference graph [7]. This is akin to technology mapping [10] and instruction selection [3]. There have been several advances made in the graph theory world that have looked at alternative algorithms to explore time-space complexity trade-offs of this problem [6, 7, 14, 2]. The data mining domain employs such a version of the problem to find frequencies of occurrences of each pattern tile within a database of forests [7, 16, 4]. There are also other variants of this problem, like the tree inclusion problem [11] that is similar yet different from the problem we are solving here.

We explore a different version of the graph isomorphism problem—given two reference trees, the goal is to enumerate all matching subtrees common to the two reference trees. Bottom-up search [2] is one of the oldest attempts at solving this problem. It simply traverses the trees from leaves up and at each node, records the subtree matches found leading to a natural dynamic programming formulation. However, this algorithm will always find subtrees that must share their leaves with the reference trees—such subtrees are sometimes referred to as maximal subtrees. Other advancements have been made on this algorithm; a common theme includes labeling nodes in the trees such that two nodes in a tree will have the same number if the subtrees under the nodes are isomorphic [15]. This is essentially similar to the value numbering algorithm used in traditional compilers [13, 1]. All these algorithms inherently find only those subtrees whose leaves are also the leaves of the reference trees. Thus, these algorithms cannot (by definition) find the subtree match shown in Fig. 1. There are related problems of finding similarity between two trees [8, 17], but their objective is similarity rather than isomorphism.

In this paper, we solve a more general problem of finding all isomorphic subtrees, including those that are not maximal. On first glance, this problem may seem to have a potentially exponential search space. We show, however, that the problem is well structured and solvable in polynomial time. We present a dynamic programming algorithm to search the reference trees for subtree isomorphism. The algorithm enumerates all (maximal and non-maximal) isomorphic subtree matches. For two reference trees of sizes \( G_1 = (V_1, E_1) \) and \( G_2 = (V_2, E_2) \), it has an \( \mathcal{O}(|V_1||V_2|) \) time and space complexity. To the best of our knowledge, this is the first paper to solve this general problem, which is a superset of traditional subtree isomorphism problems where the isomorphic trees are maximal subtrees [15].

3. PROBLEM DESCRIPTION

This section provides a formal description of the subtree isomorphism problem. First, we define some terminologies and then present the problem statement.

3.1 Definitions

We define tree as a directed acyclic graph, \( T = (V, E, r, \text{type}) \), with a set of nodes \( V \), a set of edges \( E \subseteq V \times V \), and a root node, \( r \in V \), such that all nodes except the root have exactly one parent. Every node has a specific type given by the mapping, \( \text{type} : V \mapsto \mu \). The direction of the edges is from leaves to root. The node sets, \( \text{In}(v) \) and \( \text{Out}(v) \), define the inputs and outputs of a given node. The tree invariants are given by: \( \forall \ v, s.t., v \neq r, |\text{Out}(v)| = 1 \) and for the root node, \( |\text{Out}(r)| = 0 \).

If the children of a node can be interchanged, then we say that the node is unordered. Conversely, an ordered node, \( v \), imposes a left-to-right sequencing of its children, given by \( \text{Child}(v) = (u_1, \ldots, u_n) \), where \( \forall \ 1 \leq i \leq n, u_i \in \text{In}(v) \). Whether or not a node is ordered is determined by its type and is given by the mapping \( \text{Order} : \mu \mapsto \{0,1\} \).

Putting these definitions in context, a tree in the compiler’s intermediate representation is typically an expression tree with some functional operations as nodes and the root, \( r \) computing the final result. The set, \( \mu \), represents all the operations performed by the program. Thus, \( \text{type}(v) \) specifies the operation encapsulated by \( v \). Finally, an unordered node maps to commutative operations, while ordered nodes are non-commutative.

Without loss of generality, we also add pseudo leaf nodes to the trees we consider. This is especially designed to deal with expression trees where nodes are operations and are typically associated with some inputs. The pseudo leaf nodes represent a terminal input, e.g., a variable or constant, that is typically supplied as input to the expression tree. We assume that every leaf of a tree is a pseudo node, \( v_p \), such that \( \text{In}(v_p) = \emptyset \) and \( \text{type}(v_p) = $ \), i.e., the type of a pseudo node is considered to be a special alphabet, $$. All non-pseudo nodes must have children (inputs) associated with them.

**Definition 1.** Two trees, \( T_1 = (V_1, E_1, r_1, \text{type}_1) \) and \( T_2 = (V_2, E_2, r_2, \text{type}_2) \), are isomorphic or topologically equivalent,
if there exists a relation, $\tau : V_1 \rightarrow V_2$, between $T_1$ and $T_2$, which satisfies the following conditions:

1. $\tau$ is bijective, i.e., for $\forall u, v \in V_1$, $u \neq v \implies \tau(u) \neq \tau(v)$, and $\forall v' \in V_2$, $\exists v \in V_1$, s.t., $\tau(v) = v'$.

2. $\tau$ preserves the edge relations, i.e., for $(u, v) \in E_1$, there exists $(\tau(u), \tau(v)) \in E_2$.

3. $\tau$ preserves the types, i.e., for $v \in V_1$, $\text{type}_1(v) = \text{type}_2(\tau(v))$.

4. For every ordered node, $v \in V_1$, the ordering relation is preserved, i.e., if $\text{Child}_1(v) = (u_1, \ldots, u_n)$, then $\text{Child}_2(\tau(v)) = (\tau(u_1), \ldots, \tau(u_n))$.

A subtree, $S = (V_s, E_s, r_s, \text{type})$ of a tree, $T = (V, E, r, \text{type})$, is a tree that contains a subset of the nodes of $T$ and preserves the edge, type, and ordering relations. In other words, $V_s \subseteq V$, both trees share the same type mapping, the edge relations in $E$ are preserved in $E_s$, and the ordering relations in $T$ are preserved in $S$. Further, $S$ is a maximal subtree iff $\text{Leaves}(S) \subseteq \text{Leaves}(T)$. It follows that for every node, $u$, in the tree, there exists a unique maximal subtree rooted at $u$. We refer to this mapping from a given tree node, $u$, in $T$ to its maximal subtree by $\text{MaxSubtree}(u)$.

### 3.2 Problem Statement

Given two trees, $T_1 = (V_1, E_1, r_1, \text{type}_1)$ and $T_2 = (V_2, E_2, r_2, \text{type}_2)$, find the set of all isomorphic subtrees contained within these two trees. If $T_1^*$ and $T_2^*$ are the domains of all subtrees contained in $T_1$ and $T_2$ respectively, then the goal of the problem is to find the exhaustive set of subtree isomorphisms, $ST1 \subseteq T_1^* \times T_2^*$, that satisfies the following conditions:

1. Every member is a pair of isomorphs: $\forall (S_1, S_2) \in STI$, the subtrees $S_1 \in T_1^*$ and $S_2 \in T_2^*$ are isomorphic as per Definition 1.

2. If a subtree pair is not a member of $STI$, then they are not isomorphic, i.e., $\forall (S_1 \in T_1^*, S_2 \in T_2^*) \notin STI$, the subtrees $S_1$ and $S_2$ violate some condition in Definition 1 and are therefore not isomorphic.

### 4. SUBTREE MATCHING ALGORITHM

A large body of previous work has focused on maximal subtree isomorphism. This has been shown to be a linear-time search complexity [2, 15]. The problem stated here is to find a set of subtree isomorphisms that includes not only maximal subtree matches but also those subtree matches that are not maximal. If we were to naively use maximal subtree isomorphism algorithms, then we would have to enumerate all subtrees within the given reference trees and then use the maximal subtree isomorphism algorithm on each pair of subtrees in this space. Clearly, this has exponential complexity since we are enumerating all subtrees in the reference trees. The key insight to combating this complexity is recognizing that the problem in Section 3.2 has an optimal sub-structure that we can exploit. In particular, we use the concept of maximal isomorphs to identify this sub-structure.

**Definition 2.** Given two reference trees, $T_1$ and $T_2$, let the domains, $T_1^*$ and $T_2^*$, represent all subtrees contained within these trees respectively. A maximal isomorph is a pair of subtrees, $(S_1 \in T_1^*, S_2 \in T_2^*)$, that satisfy the following conditions:

1. $S_1$ and $S_2$ are isomorphic as per Definition 1.

![Figure 2: Two sample reference trees, their corresponding prefix strings, and the maximal isomorphs](image-url)
is determined as follows: let the position in Child(u) of an input node, u, be $\pos(u) \in \mathbb{N}$. Then, the ordering of Child(v) is given by the following implication:

$$\forall u, w \in \text{Child}(v),$$

$$\text{strcmp}(\text{Prefix}(\text{MAX}(u)), \text{Prefix}(\text{MAX}(w))) < 0 \quad \implies \quad \pos(u) < \pos(w)$$

Canonical form is achieved by enforcing a lexicographical ordering on unordered nodes. Specifically, the order of children of an unordered node is the same as the lexicographic order of the children’s maximal subtrees’ prefix strings. In the above definition, $\text{MAX}(u)$ represents the maximal subtree under a given tree node, $u$ and $\text{Prefix}(\text{MAX}(u))$ gives the prefix string encoding for this maximal subtree. Converting a given tree into its canonical form involves a post-order traversal: at each unordered node, we reorder its children to obey the conditions in Definition 3. Observe the importance of canonicalization—we use it primarily to prune the search space without sacrificing optimality. In other words, we do not have to flip the children of unordered trees in looking for a match because the canonical representation of two identical subtrees is guaranteed to be the same.

4.2 The Common Prefix Substring Matching (CPSM) Formulation

After obtaining the canonical prefix strings for two reference trees, one may think of running a simple maximal substring matching on the prefix strings to obtain the maximal isomorphs of the reference trees. Unfortunately, that is not a solution, as illustrated in Fig. 2. There are two maximal isomorphs, rooted at ‘*’ and ‘-’. However, a naive substring matching algorithm can only yield the matching at ‘*’ by matching ‘(S, S)’. In fact, it can be proved that simple substring matching will only yield isomorphs that are also maximal subtrees, i.e., those sharing their leaves with the reference trees. Internal (non-maximal) subtrees like the ‘-’ in Fig. 2 will never be matched by substring matching. Therefore we need a more context-aware string scanning mechanism in order to find all maximal isomorphs.

Note that traversing an expression tree in the prefix order amounts to performing a linear scan of its prefix string. During the process of scanning a prefix string, skipping an entire operation string pattern starting at an operation effectively means stopping the traversal of the expression tree at the corresponding operation node. If we stop the traversal at an operation before reaching the terminal (‘$’), we represent that point in the string by $\text{OP}$. Fig. 3 illustrates this principle. The arrows in Fig. 3(a) show the directions of traversal and scan. After visiting node 1, pruning the subtree rooted at node 2 is tantamount to skipping the whole string pattern, $+$($S, +($S, S)$)), clearly indicating the places the traversal stopped by ‘$’ or $\text{OP}$ depending on whether the stopping point is a terminal node or intermediate node respectively.

The following theorem formally brings out important properties of the prefix strings, which are used in formulating our dynamic programming algorithm.

**Theorem 1.** Let $\text{PS}_1[1..m]$ and $\text{PS}_2[1..n]$ be the two prefix strings of length $m$ and $n$ respectively. Let $k$ be the length of the longest common prefix substring and $\text{LCPS}(k)$ be the set of all longest common prefix substring matches, each of length $k$, ending at positions $1 \leq p_1, p_2, \ldots, p_m \leq m$ in $\text{PS}_1$ and $1 \leq q_1, q_2, \ldots, q_n \leq n$ in $\text{PS}_2$, where $t = |\text{LCPS}(k)|$. Let $L_k$ refer to a member of $\text{LCPS}(k)$.

- If $\text{PS}_1[p] = \text{PS}_2[q]$, where $1 \leq p \leq m$, $1 \leq q \leq n$, then the following must hold true:

  1. If $\text{PS}_1[p - 1] = ‘$ and $\text{PS}_2[q - j + 1..q - 1] = \text{OP}$, $(2 \leq j \leq q)$, then $L_{k - 1}$ is a longest common prefix string for $\text{PS}_1[1..p - 2]$ and $\text{PS}_2[1..q - 1]$ ending at the alphabet positions $p - 2$ and $q - j$ of $\text{PS}_1$ and $\text{PS}_2$ respectively.

  2. If $\text{PS}_1[p - i + 1..p - 1] = \text{OP}$ and $\text{PS}_2[q - j = ‘$’, $(2 \leq i \leq p)$, then $L_{k - 1}$ is a longest common prefix string for $\text{PS}_1[1..p - i]$ and $\text{PS}_2[1..q - 2]$ ending at the alphabet positions $p - i$ and $q - 2$ of $\text{PS}_1$ and $\text{PS}_2$ respectively.

  3. If $\text{PS}_1[p - i + 1..p - 1] = \text{OP}$ and $\text{PS}_2[q - i + 1..q - 1] = \text{OP}$, $(2 \leq i \leq p, 2 \leq j \leq q)$, then $L_{k - 1}$ is a longest common prefix string for $\text{PS}_1[1..p - i]$ and $\text{PS}_2[1..q - j]$ ending at the alphabet positions $p - i$ and $q - j$ of $\text{PS}_1$ and $\text{PS}_2$ respectively.

  4. Otherwise, $L_{k - 1}$ is a longest common prefix string for $\text{PS}_1[1..p - 1]$ and $\text{PS}_2[1..q - 1]$, ending at the alphabet positions $p - 1$ and $q - 1$ of $\text{PS}_1$ and $\text{PS}_2$ respectively.

**Proof.** We prove these assertions using contradiction.

1. Let $L \in \text{LCPS}(k) \Rightarrow L_{k - 1} \in \text{LCPS}(k - 1)$, where $\ell^{th}$ alphabet is a ‘$’ or $\text{OP}$. In this case, $\text{PS}_1[p - 1] = ‘$’ and $\text{PS}_2[q + j + 1..q - 1] = \text{OP}$, $(2 \leq j \leq q)$. Let $L_{k - 1} \notin \text{LCPS}(k - 1)$ be a longest common prefix string for $\text{PS}_1[1..p - 2]$ and $\text{PS}_2[1..q - j]$ ending at the alphabet positions $p - 2$ and $q - j$ of $\text{PS}_1$ and $\text{PS}_2$ respectively. Therefore, by skipping over ‘$’ at $\text{PS}_1[p - 1]$ and $\text{OP}$ in $\text{PS}_2[q - j + 1..q - 1]$, one can match
The four cases presented in Theorem 1.

Case 1

$\text{PS}_1 = \ldots \text{OP}( \ldots, \ldots ) \ldots$

Case 2

$\text{PS}_2 = \ldots \text{OP}( \ldots, \ldots ) \ldots$

Case 3

$\text{PS}_1 = \ldots \text{OP}( \ldots, \ldots ) \ldots$

Case 4

$\text{PS}_2 = \ldots \text{OP}( \ldots, \ldots ) \ldots$

The above four cases are depicted in Fig. 4. The arrows pinpoint the cases clearly using two sample reference trees and prefix strings ($\text{PS}_1$ and $\text{PS}_2$). The cases 1 through 3 involve skipping over an entire operation substring or a terminal, while case 4 is an exact match of a non-leaf alphabet and therefore does not involve any skipping transforms.

This shows that the problem of canonical prefix substring matching (CPSM) has an optimal substructure, i.e., an optimal solution to this problem contains within it optimal solutions to its subproblems. For $1 \leq p \leq m$ and $1 \leq q \leq n$, let $\text{LCPSL}(p, q)$ be the length of a longest common prefix substring for $\text{PS}_1[1..p]$ and $\text{PS}_2[1..q]$. We express the three conditions presented by the first three cases in Theorem 1 as follows:

1. \text{PS}_1[p] = \text{PS}_2[q]$, and thus come up with $L'_k$ as another longest common prefix string. Now, $L'_{k-1} \notin \text{LCPS}(k-1) \Rightarrow L'_k \notin \text{LCPS}(k)$. Since $L'_k \notin \text{LCPS}(k)$, this is a contradiction, as $\text{LCPS}(k)$ is the set of all largest common prefix string matches of length $k$. Therefore, $L'_{k-1}$ is a longest common prefix string for $\text{PS}_1[1..p-2]$ and $\text{PS}_2[1..q-j]$ ending at the alphabet positions $p-2$ and $q-j$ of $\text{PS}_1$ and $\text{PS}_2$ respectively. Now, if $L'_{k-1}$ is a longest common prefix string that does not end at $\text{PS}_1[p-2]$ and $\text{PS}_2[q-j]$, it can only extend through the other three possibilities, because there is no other way to extend the common prefix string.

2. $\text{PS}_1[p-i+1..p-1] = \text{OP}$ and $\text{PS}_2[q-1] = \text{OP}$, $(2 \leq i \leq p)$. The proof of this assertion is symmetric to the above reasoning.

3. $\text{PS}_1[p-i+1..p-1] = \text{OP}$ and $\text{PS}_2[q-i+1..q-1] = \text{OP}$, $(2 \leq i \leq p, 2 \leq j \leq q)$. The proof of this assertion is also symmetric to the above reasoning.

4. When none of the above conditions are true, but only $\text{PS}_1[p] = \text{PS}_2[q]$, the previous match has to be a direct match.

The above four cases are depicted in Fig. 4. The arrows pinpoint the cases clearly using two sample reference trees and prefix strings ($\text{PS}_1$ and $\text{PS}_2$). The cases 1 through 3 involve skipping over an entire operation substring or a terminal, while case 4 is an exact match of a non-leaf alphabet and therefore does not involve any skipping transforms.

Based on the above equation, one could easily write an exponential-time recursive algorithm to compute the matching prefix substrings. However, because there are only $O(m \times n)$ distinct subproblems as per Theorem 1, we can use dynamic programming to obtain the solutions bottom up.

The dynamic programming implementation employs an $m \times n$ $\text{LCPSL}$ matrix to store the intermediate lengths of the matching substrings. In addition, it encodes the substring enumeration solu-
tion with the help of \( P_{\text{PREV}}(p, q) \) and \( Q_{\text{PREV}}(p, q) \), which are respectively defined as the last positions in the prefix strings \( PS_1 \) and \( PS_2 \) that were matched by the algorithm, prior to matching the current positions \( p \) and \( q \) in the two strings (i.e., \( PS_1[p] = PS_2[q] \), where \( 1 \leq p \leq m, 1 \leq q \leq n \)). This encoding alleviates the exponential space overhead that would be otherwise required to enumerate all solutions.

\[
P_{\text{PREV}}(p, q) = \begin{cases} 
  p - 2 & \text{if condition1} \\
  p - i & \text{if condition2} \\
  p - i & \text{if condition3} \\
  p - 1 & \text{otherwise}
\end{cases}
\]

\[
Q_{\text{PREV}}(p, q) = \begin{cases} 
  q - j & \text{if condition1} \\
  q - 2 & \text{if condition2} \\
  q - j & \text{if condition3} \\
  q - 1 & \text{otherwise}
\end{cases}
\]

Using \( LCPSL, P_{\text{PREV}}, \) and \( Q_{\text{PREV}} \), we can generate all the common prefix strings enumerating \( PS_1 \) (using \( P_{\text{PREV}} \)) and \( PS_2 \) (using \( Q_{\text{PREV}} \)). This in turn corresponds to all the maximal isomorphs in the original expression tree. In order to walk the generated substring positions in \( LCPSL \) in the forward direction, we define \( P_{\text{NEXT}} \) and \( Q_{\text{NEXT}} \) as follows: \( P_{\text{NEXT}}(p_{\text{PREV}}(p), q_{\text{PREV}}(q)) = p \) and \( Q_{\text{NEXT}}(p_{\text{PREV}}(p), q_{\text{PREV}}(q)) = q \), where \( 0 \leq p \leq m, 0 \leq q \leq n \). To enumerate all the matching prefix strings, we need to simply start with positions having \( LCPSL \) set to 1 and then walk the \( PS_1 \) or \( PS_2 \) strings using \( P_{\text{NEXT}} \) and \( Q_{\text{NEXT}} \) respectively. These matched prefix strings in turn enumerate all the maximal isomorphs in the reference trees.

In practice, in order to enumerate the longest common prefix string or the maximal isomorphs (instead of capturing all the internal ones as well), we check before setting \( LCPSL(p, q) \) whether the newly computed value is greater than the stored value of \( LCPSL(p, q) \). This is because any one of the above 4 cases can be the reason for finding a match at location \((p, q)\).

### 4.3 The CPSM Algorithm

We present the CPSM algorithm in Figure 5, which uses the above formulation. The algorithm takes the prefix strings \( PS_1 \) and \( PS_2 \) as inputs, and populates the solution in terms of matrices \( LCPSL, \Delta P_{\text{NEXT}} \) and \( \Delta Q_{\text{NEXT}} \), where \( LCPSL \) tracks the lengths of substrings at various positions in \( PS_1 \) and \( PS_2 \), \( P_{\text{NEXT}}(p, q) = p + \Delta P_{\text{NEXT}}(p, q) \) points to the next matching position for \( PS_1 \), and \( Q_{\text{NEXT}}(p, q) = q + \Delta Q_{\text{NEXT}}(p, q) \) points to the next matching position for \( PS_2 \).

The algorithm begins by building operation position tables for strings \( PS_1 \) and \( PS_2 \), which mark the beginning and ending of each valid operation in the prefix strings. The dynamic computation of the solution matrices based on the above formulation is shown between lines 21 and 57. Note that in order to favor the growth of the largest valid substring, we explicitly check whether the new substring length is greater than any existing substring length evaluated so far, at a particular matching point \((p, q)\). This is done in lines 36, 44, and 52. The dynamic updates of \( LCPSL, \Delta P_{\text{NEXT}}, \) and \( \Delta Q_{\text{NEXT}} \) can be tracked in lines 28-30, 37-39, 45-47, and 53-55.

**Theorem 2.** The running time of the dynamic programming (CPSM) algorithm as well as its space complexity is \( O(m \cdot n) \).

**Proof.** Each of the computations presented in the formulation is \( O(1) \), which is run within two nested loops of size \( m \) and \( n \).

```plaintext
Figure 5: The CPSM Algorithm
```

182
Therefore the time complexity is $O(m \times n)$. Since, we are employing three $m \times n$ matrices to capture the results, the space complexity is also $O(m \times n)$.  

4.4 A Running Example

In order to illustrate the working of the CPSM algorithm, let us consider again the reference trees shown in Fig. 2 and their corresponding prefix string representations:

$$
PS_1 = -(\ast(S,S), +(S,S))
$$

$$
PS_2 = -(S, \ast(S,S))
$$

The solution matrices $LCPSL$, $P_{NEXT}$ and $Q_{NEXT}$, built according the CPSM algorithm are shown together in Table 1a and Table 1b. There are two maximal substring matches encoded within the solution matrices. To enumerate each matching substring, we find the starting position by the entry 1 in the $LCPSL$ matrix and with the assistance of $(P_{NEXT}, Q_{NEXT})$, we find the full matching substring. The first match is $-(\ast(OP, OP))$ in $PS_1$, $-(\ast(OP, OP))$ in $PS_2$, which starts at (0,0) (as indicated by $LCPSL$) and then walking down string positions as guided by the entries (1,1), (8,3), and (15,10). When the jump of $P_{NEXT}$ or $Q_{NEXT}$ is greater than 2, it indicates stopping at an operation subtree, and we show that by putting the symbol ‘OP’. The second match is $(\ast(S,S))$ in $PS_1$, $(\ast(S,S))$ in $PS_2$, corresponding to the entries (2,4), (3,5), (4,6), (5,7), (6,8), and (7,9). The second match is a straight-forward match as there is no jump in $P_{NEXT}$ or $Q_{NEXT}$ greater than 2.

5. CASE STUDY: COMBINATIONAL RESOURCE SHARING

Thus far in the paper, we have described the theoretical rigor behind subtree isomorphism mining. In this section, we test the practical usefulness of the algorithm. Although there are many applications of this algorithm, we explore its effectiveness in the combinational resource sharing problem [12].

This is a well known problem in high-level synthesis, where the goal is to share commonly used resources in mutually exclusive control paths in the given source program. For example, if a multiplier is used within the true and false branches of an if-else statement, then we know that the result of only one multiplier is needed. Thus, it is possible to share this multiplier across both conditional branches and muxing its inputs.

Of course, we need not stop at just a single operation. We can imagine sharing entire expression trees between mutually exclusive code regions, which necessitates finding common code segments between these regions that can form the resource (or expression tree) to be shared.

To enumerate all opportunities for such resource sharing, we create expression forests for each control-flow region—one forest for each mutually exclusive control branch. Next, we can use the proposed subtree isomorphism algorithm to scan the forests to find common structural expression tree patterns that form potential sharable resources. The inputs to the algorithm are two trees from different forests. A quadratic complexity search will enumerate all subtree matches that are present in these forests and each of these matches will form a candidate for combinational resource sharing.

This scan algorithm was applied on a variety of representative input programs that are used to test and benchmark the Simulink® HDL Coder™ product. Due to proprietary issues, we cannot reveal the names of these tests but mention here that they are various applications from the signal processing and control systems domains. A total of about 50 different test programs were used. We categorized the subtree matches found into four groups. If $S_1$ and $S_2$ are the matched subtrees within reference trees, $T_1$ and $T_2$, then the four categories are illustrated in Fig. 6 and can be described as:

- **Whole**: If $S_1 = T_1 = T_2 = S_2$, then the two reference trees were themselves isomorphic.
- **Root**: Here, the subtrees shared the roots with the reference trees, i.e., $root(S_1) = root(T_1)$ and $root(S_2) = root(T_2)$. 

![](Figure 6: The four different types of isomorphic subtrees, $S_1$ and $S_2$, that can be found in two reference trees, $T_1$ and $T_2$.)

---

183
prune these matches according to some objective cost function. With the CPSM algorithm can find all four different types of matches in a matrix.

Table 1: A running example of the algorithm for finding common substrings between \(-*(S,S), +(S,S)\) and \(-*(S,)\): (a) the \((P,\text{NEXT},Q,\text{NEXT})\) pairs, where \(P_{\text{NEXT}}(p,q) = p + \Delta P_{\text{NEXT}}(p,q)\) and \(Q_{\text{NEXT}}(p,q) = q + \Delta P_{\text{NEXT}}(p,q)\), and (b) the LCPST matrix.

- **Leaves**: These are maximal subtree matches, i.e., \(\text{Leaves}(S_1) \subseteq \text{Leaves}(T_1)\) and \(\text{Leaves}(S_2) \subseteq \text{Leaves}(T_2)\).

- **Inner**: Here, at least one of \(S_1\) or \(S_2\) does not share its root with its respective reference tree and there exists at least one leaf for each subtree that is not a leaf in the reference tree.

Previous techniques proposed for subtree isomorphism primarily finds matches in the **Leaves** category \([15, 1, 2]\). Typically, approaches similar to value numbering (used in common subexpression elimination) \([13]\) are used to provide unique ids to nodes such that two nodes with the same id will be isomorphic all the way down to the leaves.

Finding matches in the **Root** category will involve a lock-step, pre-order traversal down the reference trees. The traversal ends when a difference is found in the nodes and portions of the trees matched to that point form the match. Although this is a straightforward linear search, it involves a different algorithm than the one used to find **Leaves** matches \([15]\).

To our knowledge, no solutions have been proposed to efficiently find all matches belonging to the **Inner** category. The proposed CPSM algorithm can find all four different types of matches in a single sweep. The primary strength of our algorithm is its efficiency. Without explicitly enumerating all matches, it simply encodes all matches within the solution matrices as described in Section 4.3.

The results of the subtree matches found in mutually exclusive expression forests in our benchmarks are reported in Fig. 7. We have categorized the matches according to the four groups described above. All these matches were found with a single sweep of our algorithm through the source code. The results indicate that most of the matches fall in to the **Whole** and **Root** categories. However, there are a significant number of matches that also fall in the other two categories. The results are promising in that we have been able to efficiently generate new opportunities for performing combinational resource sharing within the system design tool. The next logical step for effective combinational resource sharing is to prune these matches according to some objective cost function.

This case study is only one example of the possible applications of the algorithm. Naturally, it can also be used for sequential resource sharing, which is the traditional approach where resources across different control steps in a given pipeline schedule are shared. Again, instead of sharing just a single node, we can explore the possibilities of sharing expression trees across control steps. Other applications include code re-use optimizations, custom instruction creation in ISE architectures \([5]\), JIT optimizations that seek out frequently executed code from the context call trees \([17]\), amongst many others.

6. CONCLUSIONS

In this paper, we formally described a general version of the subtree isomorphism problem. It is the first formulation to enumerate all possible matches in two reference trees irrespective of whether the isomorphism exists at the root, leaves, or internal sections of the trees. In this respect, it is a superset of all subtree isomorphism problems addressed so far. Although, it is a seemingly exponential search space, we show that the problem is structured and thus an optimal, quadratic-time solution exists for this generic subtree isomorphism problem. We have also performed an experimental case study by applying the algorithm to find opportunities for combinational resource sharing in a set of signal processing and control system benchmarks—our results indicate that the proposed algorithm can rapidly, optimally and efficiently mine common tree patterns vital to compiler analysis and optimizations. We believe that our algorithm will find significant uses in the embedded computing domain where tree-based pattern matching problems are abundant.

7. REFERENCES


Figure 7: (a) The number of subtree matches found (by category) in each test and (b) the median size (in terms of |V|) amongst all matches found in each test.


