Path Matching in Compressed Control Flow Traces

Youtao Zhang   Rajiv Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Due to its large size, a whole program path (WPP) is stored in compressed form generated using SEQUITUR. The occurrence of an intraprocedural path in the WPP cannot be carried out using existing algorithms due to the path interruption and path context problems. We present an algorithm that addresses the above problems. The complexity of the algorithm is analyzed and experimental data is presented to demonstrate that our algorithm is efficient in practice.

1 Introduction

Data streams with large amounts of data are often stored in compressed form and one of the commonly used compression algorithm is the SEQUITUR algorithm [4]. When the need to search for a pattern in the data stream arises, it is highly desirable to avoid uncompressing the data. Therefore researchers have been developing algorithms for pattern matching that operate on compressed data [1, 2, 5, 9, 10].

This paper addresses the problem of finding an occurrence of a *path* composed of a sequence of basic blocks, which is the pattern, in a *compressed control flow trace*. A control flow trace captures the complete execution path taken by a program on a given input in the form of the complete sequence of basic blocks executed by the program. Such traces are collected and mined for subpaths that are commonly followed by the program. This information is useful for both program understanding and optimization because it has been observed that while large programs contain millions of paths, only a few thousand are observed to be taken by the program in practice. Uncompressed control flow traces are very large ranging from hundreds of megabytes for a moderate run up to several gigabytes for a long run. Therefore Larus [3] proposed the use of the SEQUITUR algorithm to compress them. The compressed form of the control flow trace produced by SEQUITUR is referred to as the *whole program path* (WPP).

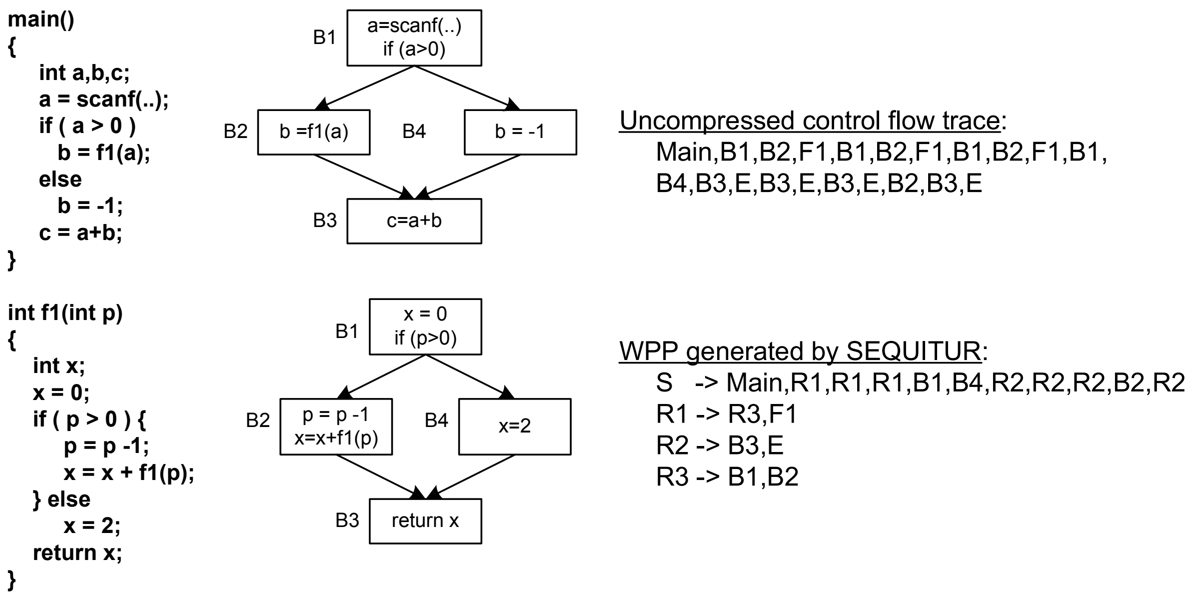
Searching for an occurrence of a path in the WPP poses unique challenges. The applications of WPPs require searching for an *intraprocedural path*. However, if an intraprocedural path contains procedure calls, it is *interrupted* by paths belonging to called procedures. Moreover, the same path may be generated in different *contexts*, that is, during the execution of different procedures. This is because each procedure reuses the same basic blocks identifiers.

Therefore due to the *path interruption* and *path context* problems, existing algorithms, such as one in [5], are not directly applicable for finding a path in a WPP. In this paper we develop a *path matching* algorithm that operates on a WPP. We analyze its complexity and present experimental data that shows that our algorithm is efficient in practice.

2 Problem Definition

Of all the different kinds of program traces that exist, the control flow trace is the most commonly used form because of the ease with which it can be collected and the variety of ways in which it can be used. Let us consider the form of the control flow trace.

Consider a program consisting of the `main` function and several other functions. Each function is represented in the form of a directed graph called the *control flow graph* (CFG). Each node in a CFG represents a *basic block* which is a straightline sequence of statements that can be entered only from the beginning and exited only from the end. The edges in a CFG capture the flow of control among basic blocks. The control flow trace consists of a sequence of basic block ids that are executed during one program run from the start to the end. In addition, it also contains indicators that identify entry and exit to a function. An entry is indicated by the appearance of the function name in the control flow trace while the exit point is universally identified by E. Using SEQUITUR a control flow trace is compressed into a WPP. Figure 1 shows an example program, its CFGs, a sample uncompressed control flow trace and the corresponding WPP. The WPP consists of a grammar which generates a single string which is the uncompressed control flow trace.



(a) Sample Code and its CFG. (b) Control Flow Trace and WPP.

Figure 1: An Example.

A WPP may be queried for different kinds of information during program analysis. Here are some examples of typical queries: *Does the path* $B_1B_2B_3$ *of* F_1 *appear in this trace?*; and *How many times does path* $B_1B_2B_3$ *of* F_1 *appear in the trace?*. In this paper we will present a path matching algorithm which can serve as the basis for answering different forms of queries. Specifically the algorithm that we present solves the following path matching problem:

Find the first occurrence of path P generated by function F in a given WPP.

The solution to the above path matching must tackle two main problems that are discussed next. These problems distinguish path matching from other pattern matching algorithms in the literature.

- *Path Interruption Problem.* Even if an intraprocedural path is executed, the sequence of block ids on the path may not appear in sequence in the control flow trace. In particular, if the blocks contain function calls, then the sequence will be interrupted by appearance of block ids corresponding to the called functions. For example, the path $B_1B_2B_3$ of function F_1 was executed 2 times, but no such sequence appears in the control flow trace.
- *Path Context Problem.* While globally unique block ids could be assigned to the basic blocks, this approach is not taken because the total number of basic blocks is very large and therefore globally unique ids will require a large number of bits. Instead the same block ids are reused within each function of the program. However, this also means that the same sequence of block ids that form a path can be generated by different functions. Therefore in addition to find an appearance of the block ids in a path we must also ensure that this sequence appears in the *context* of the function of interest. For example, the path $B_1B_2B_3$ can be generated by `main` or function F_1.

While it may appear that path interruption problem can be avoided by unwinding the traces, this is not a practical solution. Since unwinding must be performed before compression, the full online nature of SEQUITUR is lost. Moreover, we still must solve the path context problem. Therefore, in this paper, we do not alter the form of the control flow trace; but rather we develop solutions to the above problems.

3 Path Matching On Uncompressed Control Flow Trace

While our final objective is to develop an algorithm for path matching that operates on a WPP, we begin by considering path matching in context of an uncompressed control flow trace. By doing this, some insights in how to handle the *path interruption* and *path context* problems can be seen.

It is useful to view the complete control flow trace as being composed of control flow subtraces corresponding to individual function invocations. The complexity of path matching arises from the fact that while we are searching for the appearance of a path belonging to a specific function in the control flow subtrace corresponding to a function invocation, the control flow subtraces corresponding to other invocations of the same or different function may be encountered. In fact, at any given point in the complete control flow trace, multiple function invocations can be active. We refer to the control flow subtraces of these active invocations as *active subtraces.* At any point in the complete control flow trace, the trace preceding the point contains the *prefixes* of the active subtraces and the trace following the point contains the corresponding *suffixes* of these active subtraces.

To facilitate the discussion of active subtraces, we associate a *nesting level* with each entry in the control flow trace. The nesting levels of all members of a subtrace (i.e., the function name, block ids, and the end marker) are the same. Moreover this nesting level is the same as the nesting level of the corresponding function invocation in the dynamic call graph for the program run. Therefore while scanning a part of the control flow trace we can compute the relative nesting levels of the block ids encountered by incrementing the nesting level when a function name is encountered and decrementing it when an end marker is encountered. It should be noted that at any given point in the complete control flow trace the nesting levels of all active traces are distinct.

Based upon the above notion of nesting level we can state that a *control flow subtrace* of a function F in the complete control flow trace T has the following properties:

- A subtrace begins with an F and ends at the first $E \in T$ that appears after F such that the nesting levels of F and E are the same (say nl).
- A block id B between the above F and E belongs to the subtrace iff its nesting level is also nl.
- All occurrences of function names, end markers, and block ids that do not belong to the subtrace corresponding to F and E and appear between F and E in T have nesting levels greater than nl.

Traditional pattern matching algorithms (e.g., KMP algorithm) are based upon finite state automatons whose states indicate the matching status, that is, how much of the pattern has been seen so far. Path matching will also be based upon a finite state automaton. However, the form the automaton required is different since at each point we need to track the *matching status* of multiple *active subtraces*. Next we describe the form of the automaton appropriate for *path matching*.

Given an uncompressed control flow trace T, a path $P_{1..m}$, a function F_i, and the value of MNL which is the *maximum nesting level* that a control flow trace can reach, the path matching automaton for $P_{1..m}$ is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of states such that

  $$Q = \{(s_1, s_2, ..., s_l) \mid 0 \leq l \leq MNL \ and \ \forall i \in [1, l], -1 \leq s_i \leq m\}.$$

  Therefore a state in our automaton is a varying-numbered tuple where the number of entries in the tuple corresponds to the number of active traces. Each item in this tuple indicates the matching status of the corresponding active trace. In this way we solve the *path interruption problem*.
- $q_0 \in Q$ is the start state and $q_0 = (\epsilon)$
- $A \subseteq Q$ is a distinguished set of accepting states.
- Σ is a finite input alphabet, $\Sigma = \{B_1, ..., B_{max}, F_1, ..., F_{max}, E\}$ and B_j, F_j, E denote different basic blocks, function entry points and function return points.

- δ is a function from $Q \times \Sigma \rightarrow Q$, called the transition function of M. The state transition function has the following form:

  $$\begin{aligned}
  &\delta((s_1, s_2, ..., s_l), B) = (s_1, s_2, ..., \delta((s_l), B))\\
  &\delta((s_1, s_2, ..., s_l), F_i) = (s_1, s_2, ..., s_l, 0)\\
  &\delta((s_1, s_2, ..., s_l), F_j) = (s_1, s_2, ..., s_l, -1), \text{ where } F_i \neq F_j\\
  &\delta((s_1, s_2, ..., s_l), E) = (s_1, s_2, ..., s_{l-1})\\
  &\delta((-1), B) = (-1).
  \end{aligned}$$

  The state "-1" is used to handle the situation in which the function name does not match and therefore the elements of the subtraces need not be matched with P. In this way we solve the *path context* problem.

4 Compressed Path Matching Algorithm

A WPP obtained by compressing the control flow trace using SEQUITUR [4] can be represented as a set of rules denoted by R with the following form:

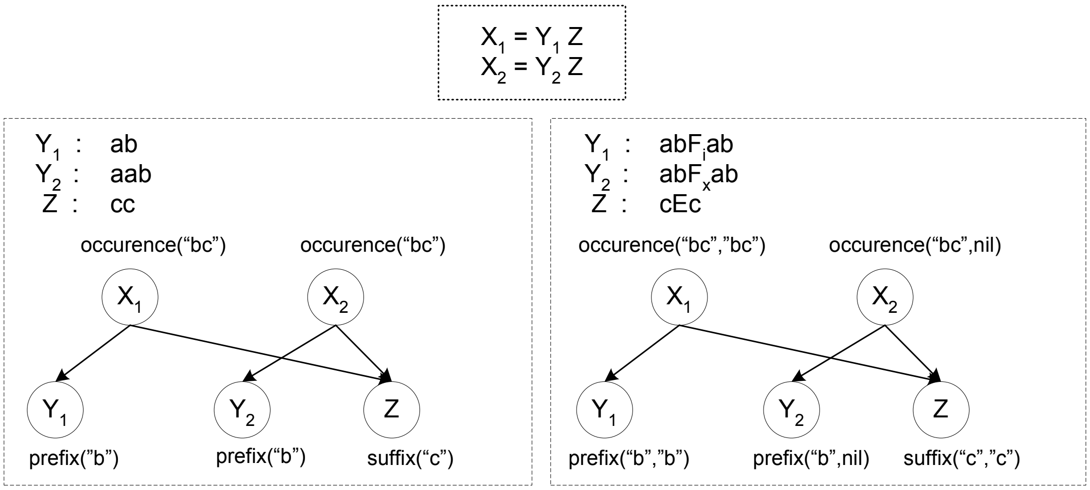
$$\begin{aligned}
&S \leftarrow X_{i_1} X_{i_2} ... X_{i_n}\\
&X_1 \leftarrow a_1; \quad X_2 \leftarrow a_2; \quad ..... \quad X_k \leftarrow a_k;\\
&X_{k+1} \leftarrow X_{l(1)} X_{r(1)}; \quad X_{k+2} \leftarrow X_{l(2)} X_{r(2)}; \quad ......\\
&X_{k+s} \leftarrow X_{l(s)} X_{r(s)};
\end{aligned}$$

where $\Sigma = \{a_i | 1 \leq i \leq k\}$, a_i can be one of B_j (block id j), F_j (entry to function F_j), and E (a function exit point).

In dictionary-based compressed matching algorithms (e.g., [1] and [2]) each non-terminal symbol in the dictionary has an associated *prefix flag*, a *suffix flag* and an *internal flag* (see example in Figure 2a). When combining two non-terminal symbols, their combination will decide the internal, prefix, and suffix flags of the combined node. However, this is not sufficient for path matching. As Figure 2b shows we can find that both "c" of Z may or may not be a part of a path ocurrence. The previous nested calling context must be considered. Instead of a single prefix flag, we associate a list of prefix flags to indicate the prefix at each different nesting level. Similarly we also must provide lists of internal and suffix flags. The size of the list is the nesting level of the associated node and it can be at most equal to the maximum nesting level MNL. Note that "nil" is different from the null string ϵ and they represent "-1" and "0" in our state automaton respectively.

Our algorithm has two main steps: a *preprocessing* step and a *path matching* step. The preprocessing is done in two parts. First several data structures defined in [1, 2] are created and second using these data structures certain *flags* associated with the non-terminals are computed. The path matching step searches through the right hand side of the starting rule and continues the search till an occurrence of the path is found.

Let us first discuss the preprocessing algorithm. As mentioned above, flags are associated with each non-terminal whose values are computed during preprocessing based upon the path P being considered. Consider a non-terminal X in the rule set R. We denote the fully expanded string corresponding to X by $XStr$. The following flags are maintained for X.



(a) Finding pattern "bc"

(b) Finding pattern ("bc",F_i)

Figure 2: A Motivating Example.

- *fnl flag:* $XStr$ may contain terminals corresponding to multiple nesting levels. These nesting levels represented in $XStr$ are always consecutive and have the following form: $(nl+1, nl+2, \cdots, nl+fnl)$. We associate a *fnl flag* with X which remembers the value of fnl. When X is being processed, we associate a nesting *level vector* with it which is of the form $(1, 2, \cdots, nl)$. In other words, the level vector only represents the *relative nesting levels* of the terminals in $XStr$.

- *Level flags:* There are three level flags associated with X – *fentry*, *fexit*, and *freturn*. $fentry$ and $fexit$ are the relative nesting levels of the first and last symbols of $XStr$. $freturn$ is the minimum relative nesting level of all E's in $XStr$. For example, if X represents $aEbEcFdFgFh$ then $fentry_X = 3$, $fexit_X = 4$, and $freturn_X = 2$. When the strings of two non-terminals are combined, the $fentry$ and $fexit$ level flags of the non-terminals are used to compute the relative nesting levels of symbols in the combined string. The $freturn$ flag is used to update the $internal$, $prefix$, and $suffix$ flags which are described next.

- *Prefix flags:* This is a list of flags with *fnl* items. Let $XStr(nl)$ denote the sequence of terminals from $XStr$ that have the relative nesting level nl. The prefix flag for level nl identifies the longest prefix of path P that is also the suffix of $XStr(nl)$.

- *Suffix flags:* This is also a list of flags. A suffix flag for level nl identifies the longest suffix of path P that is also the prefix of $XStr(nl)$.

- *Internal flags:* This is also a list of flags. An internal flag for level nl identifies the substring of path P from position i to j that is identical to $XStr(nl)$.

- *Position flag:* The position flag *foccur* gives the index in $XStr$ where the first occurence of path P ends. If P does not appear in $XStr$, then the value of $foccur_X$ is -1.

Preprocessing (P, WPP) {
    Preprocess P to enable $flag$ updates
    for each non-terminal X from R in bottom-up order **do**
        case $(X \leftarrow \alpha)$ **of**
            $\alpha == B_i$:
                if $B_i \in P$ **then**
                    $prefix_x = (B_i)$; $suffix_x = (B_i)$; $internal_x = (B_i)$;
                    $fentry_x = 1$; $fexit_x = 1$; $freturn_x = -1$;
                else
                    $prefix_x = (0)$; $suffix_x = (0)$; $internal_x = (0)$;
                    $fentry_x = 1$; $fexit_x = 1$; $freturn_x = -1$;
                endif
                if $(B_i == P)$ **then** $foccur_x = 1$;
                else $foccur_x = -1$ **endif**
            $\alpha == F_i$:
                $prefix_x = (\epsilon,\epsilon)$; $suffix_x = (\epsilon,\epsilon)$; $internal_x = (\epsilon,\epsilon)$;
                $fentry_x = 1$; $fexit_x = 2$; $freturn_x = -1$; $foccur_x = -1$;
            $\alpha == F_x$, where $x \neq i$:
                $prefix_x = (\epsilon,nil)$; $suffix_x = (\epsilon,nil)$; $internal_x = (\epsilon,nil)$;
                $fentry_x = 1$; $fexit_x = 2$; $freturn_x = -1$; $foccur_x = -1$;
            $\alpha == E$:
                $prefix_x = (\epsilon,\epsilon)$; $suffix_x = (\epsilon,\epsilon)$; $internal_x = (\epsilon,\epsilon)$;
                $fentry_x = 2$; $fexit_x = 1$; $freturn_x = 2$; $foccur_x = -1$;
            $\alpha == YZ$:
                Compute X's level vector from Y and Z's level vectors
                Y's vector: $(1, ..., y_{fexit}, ..., y_{max})$; Z's vector: $(1, ..., z_{fentry}, ..., z_{max})$
                Therefore X's vector is: $(1, ..., x_{fx}, ..., x_{max})$, where
                $x_{fx} = \max(\ y_{fexit},\ z_{fentry}\ )$, $x_{max} - x_{fx} = \max(y_{max} - y_{fexit},\ z_{max} - z_{fentry})$
                Create mapping functions $fmapY/fmapZ$ which map levels of Y/Z to levels in X
                $x_{fentry} = fmapY(y_{fentry})$; $x_{fexit} = fmapZ(z_{fexit})$;
                if $(fmapY(y_{freturn}) \leq fmapZ(z_{freturn})$ **then**
                    $x_{freturn} = fmapY(y_{freturn})$;
                    for each level $l < x_{freturn}$ **do**
                        Update X's level l flags from corresponding level flags of Y and Z
                    for each level $l >= fmapY(y_{freturn})$ and $<\ fmapZ(z_{freturn})$ **do**
                        Update X's suffix flag using suffix flag of Y
                        Update X's prefix flag suing prefix flag from Y and all flags from Z
                        Update X's internal flag with internal flags from Y and Z
                    for each level $l >= fmapZ(z_{freturn})$ **do**
                        Update X's prefix flag using prefix flag for Z
                        Update X's suffix flag using suffix flag for Y
                        Update X's internal flag to nil
                    if $foccur_y > 0$ **then** $foccur_x = foccur_y$
                    else
                        Examine prefix/suffix flags of Y/Z to see if P is in YZ.
                        if occurrence of P found in YZ **then** set $foccur_x$
                        elseif $foccur_z > 0$ **then** $foccur_x = |Y| + foccur_z$
                        else $foccur_x = -1$ **endif**
                else /* $(fmapY(y_{freturn}) > fmapZ(z_{freturn})$ */
                    processing is as above with minor modifications
                endif
        endcase
    endfor
}

Figure 3: Preprocessing Algorithm.

The preprocessing algorithm for a given R and P is presented in Figure 3. R can be represented by a directed acyclic graph (DAG). By reversing the edges we obtain R-DAG. The algorithm will update the flags of the nodes in the topological order generated from R-DAG. There are two types of nodes: leaves which are of the form $X \leftarrow a$; and internal nodes of the form $X \leftarrow YZ$. For leaf nodes the flags are initialized based upon a which is B_j, F_j or E. For an internal node $X \leftarrow YZ$, we first match level vectors of Y and Z, using $fexit$ flag of Y and $fentry$ flag of Z, and generate a new level vector for X. The lists of internal, prefix, and suffix flags of X are then updated using the flags from corresponding levels of Y and Z. The $foccur$ flag of X takes the value of $foccur$ flag of Y, if the latter is not -1. Otherwise we check if the concatenation of flags of Y and Z can form a new occurence of P. If this is not the case, then $foccur$ flag of X is computed from $foccur$ flag of Z.

The path matching algorithm is given in Figure 4. It searches through the right hand side of the starting rule and continues the search till an occurrence of the path is found. The previously computed flags of R are used as well as some additional flag computations are performed as shown in Figure 4.

PathMatching {
  Let $S \leftarrow X_{i_1}X_{i_r2}...X_{i_n}$.
  Initialize: $Z_{old} \leftarrow \epsilon$;
  for $(k = 1; k \leq n; k++)$ {
    Consider the rule $Z_{new} \leftarrow Z_{old}X_{i_k}$
    Compute the flags for Z_{new} from flags of Z_{old} and X_{i_k}
    if $foccur$ flag of Z_{new} is $\neq -1$ **then**
      PRINT(occurrence found at $foccur$); terminate;
    $Z_{old} \leftarrow Z_{new}$;
  }
  PRINT(no occorrence found);
}

Figure 4: Compressed control flow trace matching algorithm

5 The Cost of Path Matching

5.1 Complexity Analysis

Lets first consider the complexity of the preprocessing algorithm. First we perform the preprocessing needed to answer the queries used in updating the flags as described in [1, 2, 5]. As shown in prior work, the time complexity of this step is $O(m^2)$ where m is the length of path P. The preprocessing carried out to update the flags processes each rule (or the corresponding node in the WPP DAG) once and during the processing all the flags for the rule are updated. The update of each flag takes constant time as that in [2]; however flags are associated with each relevant nesting level. Since the number of entries in the level vector is at most MNL and the number of rules is $|R|$, the total time spent on updating the flags is $O(|R| \times MNL)$. Therefore the total preprocessing time is $O(m^2 + |R| \times MNL)$.

During path matching each symbol on the right hand side of the start production is processed and during the processing constant number of flag updates are performed. Thus the time spent on path matching is $O(|S| \times MNL)$. Combining the results of preprocessing step and path matching step we obtain the overall time complexity as $O((|R|+|S|) \times MNL+m^2)$.

It is also quite straightforward to show that the space complexity of the algorithm is also similar.

Theorem 1 *The time and space complexity of our compressed matching algorithm is*

$$O((|R| + |S|) \times MNL + m^2)$$

where MNL is maximum nesting level, $|S|$ is the length of the right hand side of the start rule, $|R|$ is the number of non-terminal symbols in R, and m is the length of path P.

5.2 Experimental Data

To study the effectiveness of our approach, we analyzed the characteristics of several programs from `SPECint95` benchmark suite [8]. The control flow traces were collected using the Trimaran compiler infrastructure [6] and compressed using SEQUITUR [4]. The second and third columns of Table 1 give the sizes of uncompressed control traces and WPPs respectively. As we can see, SEQUITUR is highly effective in compressing the sizes of the traces.

Benchmark	Control Flow Trace Size (MB)	WPP Size (MB)	$\|R\|$	$\|S\|$	MNL	% Symbols with Level ≤ 2
099.go	173.0	17.8	754786	667706	15	63 %
126.gcc	506.7	24.1	1018669	935995	91	48 %
130.li	81.6	1.2	50770	49385	424	60 %
132.ijpeg	267.8	12.8	486966	771220	11	99 %
134.perl	43.2	0.5	12590	58954	16	92 %

Table 1: Benchmark Characteristics.

To get a feel of the space and time complexity of our path matching algorithm, we also collected the characteristics of these traces including the values of $|R|$, $|S|$, and MNL. As expected the number of rules generated by SEQUITUR, that is $|R|$, is quite large since the WPP sizes are typically several megabytes in size. Similarly the values for $|S|$ are also large. We find that for most benchmarks the maximum nesting level MNL ranges from 11 to 91. Only for the lisp interpretor `130.li`, which is highly recursive, calls to `‘‘car’’/’’cdr’’` functions results in a higher maximum nesting level. While the worst case complexity is computed based upon MNL, in practice, the time spent on processing an internal node will depend on the size of its level vector. Therefore we also examined the lengths of these vectors. The last column in the table gives the percentage of nodes that have a level vector with length of at most 2. As we can see, on an average, this number is 72% which is very high.

6 Conclusion

In this paper we introduced a new pattern matching problem, path matching, which requires finding pattern occurrence over a semantically multi-level text string. We present a path matching algorithm over comrpessed text formats. We analyze its complexity and estimate its effiency in practice through experiments. During program analysis, path matching can provide useful information that can both help in program understanding and guide program optimization.

References

[1] A. Amir, G. Benson, and M. Farach, "Let Sleeping Files Lie: Pattern Matching in Z-compressed Files," *Journal of Computer and System Sciences*, Vol. 52, pages 299-307, 1996.

[2] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, "A Unifying Framework for Compressed Pattern Matching," *6th International Symposium on String Processing and Information Retrieval*, pages 89-96. IEEE Computer Society, 1999.

[3] J. Larus, "Whole Program Paths," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259-269, Atlanta, GA, May 1999.

[4] C. Nevill-Manning and I. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm," *Journal of Artificial Intelligence Research*, Vol. 7, pages 67-82, 1997.

[5] S. Mitarai, M. Hirao, T. Matsumoto, A. Shinohara, M. Takeda, and S. Arikawa, "Compressed Pattern Matching for SEQUITUR," *Data Compression Conference* (DCC), 2001.

[6] Trimaran Compiler Infrastructure. http://www.trimaran.org/.

[7] Y. Zhang and R. Gupta, "Timestamped Whole Program Path Representation and its Applications", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 180-190, Snowbird, UT, June 2001.

[8] http://www.spec.org/osg/cpu95/.

[9] G. Navarro, T. Kida, M. Takeda, A.Shinohara, and S.Arikawa, "Faster Approximate String Matching Over Compressed Text," *Data Compression Conference*, pages 459-467, 2001.

[10] A. Amir and G. Benson, "Efficient Two-dimensional Compressed Matching," *Data Compression Conference*, pages 279-288, 1992.