Representing Sequences in Description Logics Using Suffix Trees*

Daniel Kudenko and **Haym Hirsh**
lastname@cs.rutgers.edu
Department of Computer Science
Rutgers University, Piscataway, NJ 08855

Abstract

This paper describes an approach for representing and manipulating sequences in description logics (DLs). The key idea is to represent sequences using suffix trees, then represent the resulting trees in a DL using traditional (tractable) concept and role operators. This approach supports the representation of a range of types of information about a sequence, such as the locations and numbers of occurrences of all subsequences of the sequence. Moreover, subsequence testing and pattern matching reduces to subsumption checking in this representation, and computing the least common subsumer of two terms supports the application of inductive learning to sequences. Finally, we describe a simple addition to our approach, using the same set of DL operators, that extends our representation to handle additional types of information, such as sequence lengths and the existence and number of occurrences of palindromes in a sequence.

1 Introduction

The representation and manipulation of sequences has proven important across a wide range of tasks (such as sequences of characters in text processing, sequences of molecular compounds in biological sequences, and sequences of operators in a plan), and they have therefore received substantial attention in computer science for many years. In particular, how a sequence is represented has been found to have a significant impact on many aspects of the quality and tractability of sequence-manipulation tasks. However, although researchers in knowledge representation have accorded much study to how to conceptualize domains using description logics (DLs) — developing a well-defined semantics, thoroughly analyzed reasoning algorithms [Nebel, 1990], and many real applications ([Devanbu, 1993] and many more) — little attention has been given to representing and manipulating sequences in DLs. That is the problem addressed in this paper.

DLs support a range of reasoning tasks. The tasks that are central to this paper are *subsumption* and computation of *least common subsumer* (LCS) [Cohen *et al.*, 1992].[1] Our goal in this work is to provide a way to represent sequences in DLs that maintains a wide range of information about a sequence and supports various reasoning tasks over sequences while still preserving well-understood DL semantics and algorithms. Further, traditional DL reasoning tasks should have common-sense meanings for the represented sequences. Approaches such as Isbell's [Isbell, 1993] for representing sequences in the Classic DL [Borgida *et al.*, 1989] therefore do not meet our goals, in that such an approach requires the addition of new sequence-specific operators to a DL, resulting in an extended language without most DLs' well-defined semantics and tractable algorithms.

This paper describes a different approach to representing sequences in DLs that requires no new operators, and instead uses a traditional and tractable set of DL operators. Our basic idea is to represent strings using suffix trees [McCreight, 1976], and then represent the suffix trees in the DL, with the resulting DL expressions built only out of concept conjunction (**AND**) and roles with value and number restrictions (**ALL**, **ATLEAST**, and **ATMOST**). The resulting representation makes it possible to represent properties of sequences such as the locations and frequencies of all subsequences of a sequence. Further, a simple extension to this suffix-tree approach, using the same basic set of DL operators, allows the representation of additional sequence information such as sequence length and subsequence palindrome counts.

Note that, without loss of generality, for the remainder of this paper we will use the term "string" instead of "sequence", as well as other vocabulary arising in the string-matching literature, to maintain consistency with the terminology commonly used with suffix trees (whose

*We thank William Cohen and Martin Farach for many helpful discussions concerning this work.

[1] A concept expression C is a *least common subsumer* (LCS) of two concept expressions C_1 and C_2 iff C subsumes both C_1 and C_2 and there is no other $C' \neq C$ that also subsumes C_1 and C_2 and is subsumed by C.

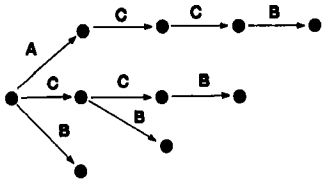

Figure 1: Suffix tree for the string "ACCB"

origins come from the theory of string matching).

2 Representing Strings in DLs Using Suffix Trees

Our approach to representing strings in DLs uses a data structure developed in the string matching literature known as "suffix trees". This section first describes suffix trees, then shows how they can be represented and manipulated in DLs.

2.1 Suffix Trees

Given a string S over an alphabet Σ, a suffix tree for S is the unique tree with the following properties:[2] (1) each edge is labeled with a single character from Σ (and a path is labeled with the sequence of the characters labeling the edges along the path); (2) no two outgoing edges from a single node are labeled with the same character; (3) each path from the root to a leaf is labeled with a suffix of S; and (4) for each suffix of S there is a corresponding path from the root to a leaf.

For example, the suffix tree for the string "ACCB" (over the three-letter alphabet $\{A, B, C\}$, which for exposition reasons serves as the alphabet for all examples in this paper) is given in Figure 1. Every suffix of the string corresponds to a path from the root of the tree to some leaf, and each path from the root to some leaf corresponds to a suffix of the string.

The suffix tree for a string S of length $|S|$ can be computed rather simply, in time $\Omega(|S|^2)$.

2.2 Representing Suffix Trees in DLs

Representing a suffix tree in a DL is fairly straightforward, given that the "topology" of a concept expression using roles is already often depicted in tree form. The task is simply to generate the appropriate expression whose tree is identical to the given suffix tree. The leaves of the tree are represented by a primitive concept, NODE, that is introduced solely for this purpose.

To do so, we create a role for every element of the string's alphabet Σ. The algorithm computes an expression arising from each successor of the root of the tree

[2]Note that our definition of suffix trees differs from that used in many string matching algorithms, in that rather than collapsing into a single edge each path all of whose nodes have single successors, we leave such paths intact to permit encoding a wider range of information than is typically represented with suffix trees. This will be clear once the representation extensions are presented.

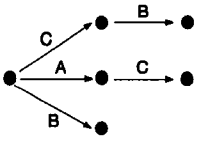

Figure 2: Example of a tree that corresponds to no string

and conjoins them together. Thus, for example, the DL definition corresponding to the suffix tree from Figure 1 is the following.

```
(and (all A (all C (all C (all B NODE))))
     (all C (and (all C (all B NODE))
                 (all B NODE)))
     (all B NODE))
```

Importantly, the DL subsumption relation has a practical use for reasoning about strings. If ST(S) is the suffix tree for string S, and DL(T) is the DL expression for a suffix tree T, the following is true:

Lemma 1 *If S_1 and S_2 are strings, then S_1 is a substring of S_2 iff DL(ST(S_2)) $\sqsubseteq$ DL(ST(S_1)).*

In other words, subsumption checking of the DL expressions generated from two strings is equivalent to substring testing. The proof of this lemma follows from the fact that DL(ST(S_1)) subsumes DL(ST(S_2)) iff all paths in the first appear in the second. Since every path from the root to an internal node of a suffix tree ST(S) corresponds to a substring of S, the set of all paths in DL(ST(S_1)) is a subset of those in DL(ST(S_2)) iff S_1 is a substring of S_2.

3 Pattern Matching Using Suffix Trees

The previous section showed that suffix trees can be used to represent strings. But what about trees that are labeled with characters yet do not correspond to strings, such as the tree in Figure 2? In this section we present a semantics for such trees that lets us go beyond substring checking to handle a range of forms of pattern matching.

3.1 Simple Pattern Matching

A suffix tree can be thought of as a data structure that maintains all substrings of a string — each path from the root to some node corresponds to a substring of the string. To enable pattern matching with suffix trees we associate an interpretation to *any* tree whose edges are labeled with characters (we call these trees *pattern trees*): a pattern tree denotes the set of all strings that contain all substrings that correspond to paths from the root to a node. More formally, each pattern tree T for an alphabet Σ is interpreted as a set of strings in the following way:

$$T^I := \bigcap_{v \in N(T)} (\Sigma^* L(v) \Sigma^*)$$

where $L(v)$ is the label of the path from the root to the node v ($L(root)$ is by definition the empty string) and

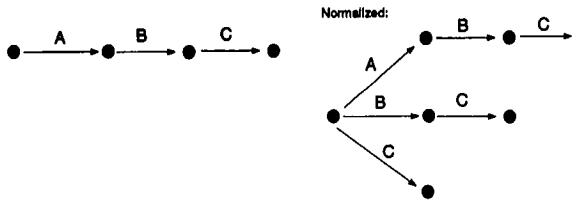

Figure 3: A pattern tree and its normalized version

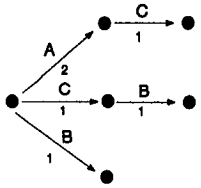

Figure 4: A pattern tree with substring frequencies

$N(T)$ is the set of nodes in T. Thus, for example, the tree shown in Figure 2 is interpreted as the set of strings containing the substrings "CB", "AC", and "B". Note that the suffix tree for a string is a special case of a pattern tree, where the interpretation of a suffix tree for a string S is the set of strings $\Sigma^* S \Sigma^*$, i.e., all strings that contain S as a substring.

Pattern trees make it possible to test whether a string contains some desired collection of strings. Given some set of strings, it is straight-forward to create a pattern tree that contains all the desired strings, with the DL suffix-tree creation algorithm still applying to such generated trees. Thus, for example, the pattern tree in Figure 2 is represented by the DL expression

```
(and (all C (all B NODE))
     (all A (all C NODE))
     (all B NODE))
```

As with Lemma 1, it is possible to state formally the relationship between the strings matched by a pattern and the subsumption relation applied to their corresponding DL representations. In order to do this, pattern trees have to be normalized in the following way. Make sure that for each path from a node to a leaf there is a path with the same labels from the root to a leaf. The resulting normalized pattern tree denotes the same set of strings as the original tree. Figure 3 contains an example for this normalization.

Let $Norm(T)$ be the normalized version of the pattern tree T. Then the following lemma holds:

Lemma 2 *If T_1 and T_2 are pattern trees, then $T_1^I \subseteq T_2^I$ iff $DL(Norm(T_1)) \sqsubseteq DL((Norm(T_2))$.*

In other words, the lemma states that subset relationships amongst sets of strings denoted by pattern trees are equivalent to subsumption relationships among the corresponding DL expressions.

The previous section showed how the suffix-tree string representation reduces substring testing to subsumption checking. Note that despite the semantics of a suffix tree now denoting a set of strings, this property still holds:

Corollary 1 *If S_1 and S_2 are strings, then $DL(ST(S_1)) \sqsubseteq DL(ST(S_2))$ iff S_2 is a substring of S_1.*

The proof of this corollary follows from the fact that $\Sigma^* S_1 \Sigma^* \subseteq \Sigma^* S_2 \Sigma^*$ iff S_1 is a substring of S_2.

Our final corollary shows the usefulness of pattern trees for pattern matching, namely that if a pattern tree T matches some string S then the proper relationship holds between their DL representations:

Corollary 2 *If S is a string and T is a pattern tree, then $DL(ST(S)) \sqsubseteq DL(T)$ iff all paths in T are labeled by substrings of S.*

Finally, we note that these semantics for pattern trees also makes the LCS operator have intuitively appealing meaning. If T_1 and T_2 are pattern trees, then the pattern tree corresponding to the LCS of $DL(T_1)$ and $DL(T_2)$ subsumes the DL expressions for exactly those strings that contain all the strings corresponding to the paths contained in both T_1 and T_2. For example, the pattern tree in Figure 2 corresponds to the LCS of the DL expressions for the suffix trees of the strings "ACBC" and "CBAC". Because of this, LCS can be used to perform inductive learning over strings. Indeed, our original motivation for this work was to find a way to represent strings in DLs so that existing DL learning approaches can be applied [Cohen and Hirsh, 1994].

3.2 Pattern Matching with Substring Counts

A simple pattern tree as presented in the previous section contains only information about the existence of patterns (i.e., substrings), but not on how often they can occur in a string. The representation of a pattern tree can be extended to encode such information by labeling edges with the minimum number of times a certain pattern may occur in a string. To be more precise, each edge e in the tree now carries a number that restricts the number of minimum occurrences of $L(target(e))$, the string that labels the path from root to $target(e)$. The pattern tree in Figure 4, for example, matches all strings with at least two "A", and at least one "C", one "B", one "CB", and one "AC" — number restrictions are displayed below the respective edge. It is straight-forward to extend the suffix-tree creation algorithm to handle such string-frequency counts, with the time and space complexity remaining quadratic in the length of the string.

The denotational semantics for pattern trees have to be modified for this extended pattern-matching language. The interpretation of a tree T is now the set of strings

$$T^{I_F} := \bigcap_{v \in N(T)} [(\Sigma^* L(v) \Sigma^*)^{F_{min}(v)}$$

where $F_{min}(v)$ is the minimum number restriction on the incoming edge of node v. Intuitively the interpretation

```
(and (all A (and (all C (and (all A (and (all C (atleast 1 min-occurs))
                                         (atleast 1 min-occurs)))
                             (atleast 2 min-occurs)))
                 (atleast 2 min-occurs)))
     (all C (and (all A (and (all C (atleast 1 min-occurs))
                             (atleast 1 min-occurs)))
                 (atleast 2 min-occurs))))
```

Figure 5: DL expression for the suffix tree for "ACAC"

defines the set of all strings that contain $L(v)$ at least $F_{min}(v)$ times.

Representing such extended pattern trees in DLs is straightforward by using a new role `min-occurs`. To represent the F_{min} value of some pattern-tree edge e we add an "`(atleast` $F_{min}(e)$ `min-occurs)`" restriction to the node in the DL expression that corresponds to the destination of e. Figure 5 gives the DL definition that results for the string "ACAC". It contains, for example, the information that "AC" occurs at least 2 times.

We conclude this section by noting that Lemma 2 and its corollaries still hold, i.e., subsumption still computes pattern matching and does substring testing. Further, the LCS of the DL definitions for two pattern trees keeps its meaning, representing the intersection of the two sets of strings described by the pattern trees.

3.3 Pattern Matching with Substring Locations

In cases when the position of these substrings is of importance, suffix trees can be extended with positional information. This can be done in a fashion analogous to that of the previous section, with each edge e having two new integer labels, one for the ending location of the first occurrence of $L(\text{target}(e))$ and a second for the last occurrence. Figure 6 shows the pattern tree that is generated for the string "ACBAC". This tree contains, for example, the information that the first occurrence of "AC" ends at position 2 and the last ends at position 5 in the string "ACBAC". It is again straight-forward to modify the pattern-tree creation algorithm so that it creates pattern trees with positional information and maintains its quadratic computational complexity.

The semantics of suffix trees with positional information can be defined as follows:

$$T^{I_P} := \bigcap_{v \in N(T)} \bigcup_{n=P_{min}(v)-|v|-1}^{P_{max}(v)-|v|-1} (\Sigma^n L(v) \Sigma^*)$$

where $P_{min}(v)$ and $P_{max}(v)$ are the first and last positions in a string where $L(v)$ may begin.

Pattern trees can be represented in DLs using a similar approach to the one used with substring counts. We use one additional role, `position`. Each P_{min} value on an edge e adds an "`(atleast` $P_{min}(e)$ `position)`" restriction to the node in the DL expression that corresponds to the destination of e, and each P_{max} value adds an "`(atmost` $P_{max}(e)$ `position)`" restriction. Finally, note that although Lemma 2 and Corollary 2 still hold for pattern trees with positional information, Corollary 1 does not. In other words, subsumption still implements pattern matching, but no longer supports substring checking. The LCS of two DL pattern trees, on the other hand, still computes the intersection of the substrings in the two pattern trees, unioning the interval of locations for each resulting string.

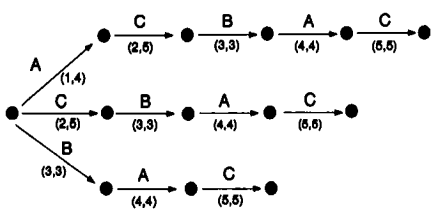

Figure 6: Suffix tree with positional information for "ACBAC"

3.4 Combining Frequency and Positional Information

We end this section by noting that the two extended forms of pattern matching based on suffix trees presented in this section are not mutually exclusive. The conjunction of the DL definitions generated from a string by each approach would give the suffix tree that carries both positional and pattern frequency information. In the resulting tree an edge has three numbers attached to it. Note that each extension can be independently used or ignored, depending on the application domain. The semantics for pattern trees with such combined positional and frequency information can be easily derived from the semantics of the individual constructs, by forming the intersection of the two sets of strings defined by the trees for positional and frequency information: $T^{I_{Comb}} := T^{I_F} \cap T^{I_P}$.

4 Representing Other String Properties

In addition to representing information about the desired substrings that should appear in a string and various properties of these substrings, it is often desirable to associate properties with an overall string. For example, one might want to reason about the length of a string,

the number of palindromes that it contains, or "extrinsic" information about the string (the author of some text, the functional role of a DNA sequence, etc.).

Such information can be easily added to the pattern-tree representation of the previous section. Each such property adds a new edge to the root of the tree labeled with the name of that property (length, palindrome-count, author, functional-role, etc.). When the value of the property is an integer the information can be stored as a number restriction in a manner analogous to how string counts or locations were handled in the previous section. To handle other, non-numeric-valued properties, each possible value must be defined as a concept and the information can be attached to the DL pattern tree by labeling the target node of the edge leaving the root with that property name. For example, the following is the DL expression for a tree with a hypothetical DNA label:

```
(and (all C (and (all C (all A (all A NODE)))
                 (all A (all A NODE))))
     (all A (all A NODE))
     (all Context PROMOTER-XY))
```

5 Final Remarks

This paper has proposed a representation formalism, based on the suffix-tree data structure, to represent and reason about sequences in description logics. Strings and patterns on strings are represented as a form of suffix trees, with the trees in turn represented as DL expressions. Using this representation subsumption supports substring checking as well as a (modular) range of forms of pattern matching. We also described a simple extension to this representation that allows representing global properties of a string.

A number of interesting issues still remain. The first is the question of broadening the class of patterns than can be represented in this suffix-tree-based formalism by, for example, introducing meta-characters such as the "*" character to denote a "don't care" position in a string. One obvious candidate for doing so is through the use of role hierarchies, although this can result in exponentially large normalized DL representations [Hollunder and Nutt, 1990]. A second issue is the question of using this representation to learn from sequences. Indeed, this work was originally motivated by our attempt to apply DL learning algorithms based on the LCS operation [Cohen and Hirsh, 1994] to sequences, and developing noise-tolerant methods suited to such tasks is a topic for future research.

References

[Borgida *et al.*, 1989] A. Borgida, R.J. Brachman, D.L. McGuinness, and L.Resnick. Classic: A structural data model for objects. In *Proceedings of SIGMOD-89*, 1989.

[Cohen and Hirsh, 1994] W. Cohen and H. Hirsh. Learning the classic description logic: Theoretical and experimental results. In *Principles of knowledge representation and reasoning : proceedings of the third international conference (KR '92)*, 1994.

[Cohen *et al.*, 1992] W. Cohen, A. Borgida, and H. Hirsh. Computing least common subsumers in description logics. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Devanbu, 1993] Prem Devanbu. Translating description logics into information server queries. In *Second International Conference on Information and Knowledge Management*, 1993.

[Hollunder and Nutt, 1990] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Technical Report RR-90-04, DFKI, 1990.

[Isbell, 1993] C.L. Isbell. Sequenced classic. Research Note, AT&T Bell Laboratories, 1993.

[McCreight, 1976] E.M. McCreight. A space economical suffix tree construction algorithm. *J. Assoc. Comput. Mach*, 23:262–272, 1976.

[Nebel, 1990] Bernhard Nebel. *Reasoning and revision in hybrid representation systems.* Springer-Verlag, Berlin, 1990.