Representing Sequences in Description Logics

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

Abstract

Representing and manipulating sequences in description logics (DLs) has typically been achieved through the use of new sequence-specific operators or by relying on host-language functions. This paper shows that it is not necessary to add additional features to a DL to handle sequences, and instead describes an approach for dealing with sequences as first-class entities directly within a DL without the need for extensions or extra-linguistic constructs. 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 variety 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 reduce to subsumption checking in this representation, while computing the least common subsumer of two terms supports the application of inductive learning to sequences.

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) — as yet no direct way has been developed for representing sequences as first-class entities within a DL. This paper describes such an approach, where sequences are represented using the traditional constructs found in most DLs, in contrast to typical approaches that use new sequence-specific constructs or that fall outside the logic by appealing to host-language functions.

In a DL a domain is modeled in terms of three kinds of entities: concepts (denoting subsets of a domain of individuals), roles (denoting relations between individuals), and features (also known as attributes, denoting functional roles). New concepts are defined using these building blocks together with combination operators such as conjunction. For example, Table 1 gives some basic concept-formation operators and their semantics over a domain of individuals Δ. Although many further operators in addition to this small set have been studied (e.g., CLASSIC (Borgida *et al.* 1989), KRIS (Baader & Hollunder 1992), and others), for this paper we will only need to focus on this core set of well-explored operators.

DLs support a range of reasoning tasks. One task that is central to this paper is that of *subsumption*: one concept expression C_2 *subsumes* a second concept expression C_1 (written $C_1 \sqsubseteq C_2$) iff $\mathcal{I}(C_1) \subseteq \mathcal{I}(C_2)$ for any model $\langle \mathcal{I}(), \Delta \rangle$ of the terminology, i.e., the set denoted by C_1 is a subset of the set denoted by C_2. Also relevant is the task of computing the *least common subsumer* of two terms (Cohen, Borgida, & Hirsh 1992), an operation that forms the basis for one approach to machine learning on DLs (Cohen & Hirsh 1994): 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. For most common DLs (such as the one given in Table 1) the least common subsumer of two expressions is always unique (Cohen, Borgida, & Hirsh 1992).

Our goal in this work is to provide a way to represent sequences directly within a DL, placing them on equal footing with other entities of interest that might

We thank Alex Borgida, William Cohen, and Martin Farach for many helpful discussions concerning this work.

Constructor	Semantics
AND	$\mathcal{I}(\texttt{(AND } \texttt{D}_1 \ \ldots \ \texttt{D}_n\texttt{)}) = \cap_{i=1}^{n} \mathcal{I}(\texttt{D}_i)$
ALL	$\mathcal{I}(\texttt{(ALL r D)}) = \{x \in \Delta : \forall y \ \langle x, y\rangle \in \mathcal{I}(\texttt{r}) \Rightarrow y \in \mathcal{I}(\texttt{D})\}$
ATLEAST	$\mathcal{I}(\texttt{(ATLEAST n r)}) = \{x \in \Delta : \lvert\{y : \langle x, y\rangle \in \mathcal{I}(\texttt{r})\}\rvert \geq n\}$
ATMOST	$\mathcal{I}(\texttt{(ATMOST n r)}) = \{x \in \Delta : \lvert\{y : \langle x, y\rangle \in \mathcal{I}(\texttt{r})\}\rvert \leq n\}$
THING	$\mathcal{I}(\texttt{THING}) = \Delta$

Table 1: Description Logic Constructors (from (Cohen & Hirsh 1994))

also be represented using the DL. Such a representation should maintain a wide range of information about a sequence and support 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 for which well-defined semantics and tractable algorithms are difficult to develop or even don't exist at all.

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, namely those given in Table 1. Our basic idea is to represent sequences 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 roles with value and number restrictions. The resulting representation makes it possible to represent properties of sequences such as the locations and frequencies of all subsequences of a sequence.

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 origins come from the theory of string matching).

Representing Strings and Substring Checking

This section begins the paper with a description of how strings can be represented within a DL using suffix trees and how substring checking can be achieved using subsumption testing over this representation. We begin by first describing suffix trees, then discussing their representation and manipulation within a DL.

Suffix Trees

Given some string S over an alphabet Σ, its *simple suffix tree (SST)* is a tree whose collection of all paths

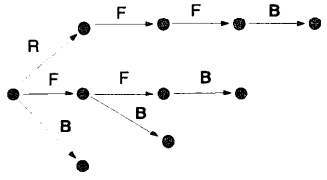

Figure 1: Suffix tree for the string "RFFB"

from the root to any terminal node gives the complete set of suffixes for the string S. More formally, the suffix tree for S is a tree where (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.

Consider, for example, robot movement plans, i.e., sequences of movement actions, where the robot is able to take one of four actions: L (turn left), R (turn right), F (move forward) and B (move back). Robot plans can be viewed as strings over the alphabet {L,R,F,B}, where each character represents one of the actions the robot may take. Figure 1 shows the suffix-tree representation for the string (i.e., plan) "RFFB" . 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.

A quadratic (that is, optimal) time algorithm for generating the unique suffix tree for a string can be easily developed:

Theorem 1 *Given a string S over an alphabet Σ, there exists an algorithm that computes the unique suffix tree for S, and that runs in time $\theta(|S|^2)$.*

One nice property of simple suffix trees is that if one string S_1 is a substring of S_2 then SST(S_1) is a subtree of SST(S_2) (where one tree T_1 is a subtree of a second T_2 if all paths in T_1 occur in T_2):

Lemma 1 *Given two strings S_1 and S_2, S_1 is a substring of S_2 iff SST(S_1) is a subtree of SST(S_2).*

This property of simple suffix trees is crucial for our representation of strings in DLs. In particular, the tra-

ditional suffix-tree representation for a string S (McCreight 1976) only requires $O(|S|)$ edges by collapsing into a single edge each path all of whose nodes have single successors (paths all of whose nodes have exactly one incoming edge and one outgoing edge). Such edges are labelled with a string (namely the string corresponding to the collapsed path), instead of a single character. However, to make it possible for subsumption to do substring checking and pattern matching as we describe below we are unable to do this and must instead use a version of suffix trees (which we call simple suffix trees) that does not do this compaction, and in the worst case may therefore require $O(|S|^2)$ edges. Otherwise, for instance, although "F" is a substring of "FB", the suffix tree for "F" would not be a subtree of the compact suffix tree for "FB".

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, as "description graphs" (Borgida & Patel-Schneider 1994). The task is simply to generate the appropriate expression whose description graph is identical to the given suffix tree, with nodes of the tree labeled by the primitive concept NODE (which is introduced solely for this purpose).

To do so, we create a role for every element of the string's alphabet Σ. The process of creating the desired expression from a given suffix tree is elementary. For example, the DL definition corresponding to the suffix tree from Figure 1 is the following.

```
(and NODE
     (all R (and NODE
                 (all F (and NODE
                             (all F (and NODE
                                         (all B NODE)))))))
          (all F (and NODE (all F (and NODE (all B NODE)))
                           (all B NODE)))
     (all B NODE))
```

Let SST be the function that maps a string S to its unique suffix tree and let DLST be the function that computes the DL expression for this suffix tree. We use DL(S) to refer to the function that takes a string S, computes its simple suffix tree using the SST function, and then applies the DLST function to the result, i.e., DL(S) = DLST(SST(S)).

Reasoning with DL Suffix Trees

It can be shown that for any string S the description-graph representation for the DL expression created by DL(S) is isomorphic to the simple suffix tree SST(S) (ignoring NODE labels in the description graph). Moreover, this leads to the following practical result for using the DL subsumption relation to reason about strings:

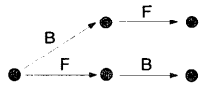

Figure 2: Example of a tree that corresponds to no string

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

In other words, subsumption checking of the DL expressions generated from two strings is equivalent to substring testing. Thus, for example, it is possible to check whether a robot movement plan contains the redundant action sequence "FB" by simply using subsumption testing on the DL expressions corresponding to "FB" and to the plan in question.

Pattern Matching Using Suffix Trees

The previous section showed that suffix trees can be used to represent strings, and that these, in turn, can be represented in a description logic. But what about trees that are labeled with characters yet do not represent the suffix tree for any 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.

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 $N(T)$ is the set of nodes in T. We say pattern tree T *matches* a string S if $S \in T^I$.

Thus, for example, the tree shown in Figure 2 is interpreted as the intersection of $\Sigma^*FB\Sigma^*$ and $\Sigma^*BF\Sigma^*$, i.e., the set of plans containing both the redundant subplans "BF"and "FB"; it thus would match the string "LFBFRL". Finally, note that this definition of pattern trees can also be used to give a pattern-matching semantics to traditional suffix trees, 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.

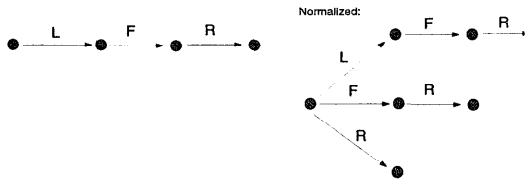

Figure 3: A pattern tree and its normalized version

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

```
(and NODE
     (all B (and NODE (all F NODE)))
     (all F (and 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 by making sure that for each path from an internal node to a leaf there is a path with the same labels from the root to a leaf. Note that a pattern tree and its normalized version denote the same set of strings. Figure 3 contains an example of this normalization.

Let $Norm(T)$ be the normalized version of the pattern tree T. Any two trees that have identical semantics have identical normal forms and vice versa, as is stated in the following lemma.

Lemma 2 *Let T_1 and T_2 be two pattern trees. Then $T_1^I = T_2^I$ iff $Norm(T_1) = Norm(T_2)$.*

We can now state the following lemma.

Lemma 3 *If T_1 and T_2 are pattern trees, then $T_1^I \subseteq T_2^I$ iff $DLST(Norm(T_1)) \sqsubseteq DLST(Norm(T_2))$.*

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. This can be seen by noting that the set of all strings in T^I is identical to strings obtained from all paths in $Norm(T)$.

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 (i.e., under this pattern-tree semantics for suffix trees $\mathrm{SST}(S)$ corresponds to *all* strings that *contain* S as a substring), this property still holds:

Corollary 2 *If S_1 and S_2 are strings, then $DL(S_1) \sqsubseteq DL(S_2)$ iff S_2 is a substring of S_1.*

Our last 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 3 *If S is a string and T is a pattern tree, then $DL(S) \sqsubseteq DLST(T)$ iff T matches S.*

In other words, subsumption checking over strings and trees represented in this fashion accomplishes the desired pattern-matching operation.

Finally, we note that this semantics for pattern trees also makes the LCS operator have intuitively appealing meaning. If T_1 and T_2 are pattern trees, then the (normalized) LCS of $\mathrm{DLST}(T_1)$ and $\mathrm{DLST}(T_2)$ subsumes the DL expressions for exactly those strings that are matched by either T_1 or T_2. For example, the pattern tree in Figure 2 corresponds to the LCS of the DL expressions for the suffix trees of the plans "BFB" and "FBF".

Lemma 4 *If S is a substring of S_1 and S_2 then $Norm(LCS(DL(S_1), DL(S_2))) \sqsubseteq DL(S)$.*

Because of the above property, LCS can be used to perform inductive learning over strings using existing LCS-based learning algorithms (Cohen & Hirsh 1994). 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 (similar to the way Cohen explored ways to represent strings in Horn-clause form to enable the use of inductive logic programming on strings (Cohen 1995)).

Note furthermore that the conjunction of two DL expressions for pattern trees T_1 and T_2 results in an expression corresponding to a pattern tree that contains all paths from the root to a leaf in either tree. Therefore the conjoined pattern tree matches all the strings that match both T_1 and T_2. Conjunction of the DL expressions gives a single pattern tree that is equivalent to requiring both pattern trees to match a string.

Extensions

The preceding sections described how strings and patterns on strings can be represented in description logics using suffix trees, and how various operations on them can be achieved using traditional DL operations. This section describes additional information that be can encoded and reasoned about using this basic suffix-tree-based DL representation.

Extension 1: Pattern Matching with Substring Counts

A simple pattern tree as presented in the previous section contains only information about the existence of

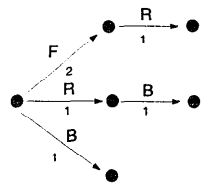

Figure 4: A pattern tree with substring frequencies

patterns (i.e., substrings), but not 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 must occur in a string. To be more precise, each edge e in the tree now carries a number that restricts the number of 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 plans with at least two "F"s, and at least one "R", one "B", one "RB", and one "FR" — number restrictions are displayed below the respective edge. The extension of the suffix-tree creation algorithm to handle such string-frequency counts is straight-forward. The time and space complexity remains 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 defines the set of all strings that contain each $L(v)$ 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 plan "FRFR". It contains, for example, the information that "FR" occurs at least 2 times.

We conclude this subsection by noting that Lemma 3 and its corollaries still hold, i.e., subsumption still computes pattern matching and does substring testing. Further, Lemma 4 still holds as well, i.e., 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. Figure 4 shows the pattern tree corresponding to the LCS of DLST("FRBR") and DLST("FBFR"), for example.

```
(and NODE
     (all F (and NODE
                 (atleast 2 min-occurs)
                 (all R (and NODE
                             (atleast 2 min-occurs)
                             (all F (and NODE
                                         (atleast 1 min-occurs)
                                         (all R (and NODE
                                                     (atleast 1 min-occurs)))
                                         ))
                             ))
                 ))
     (all R (and NODE
                 (atleast 2 min-occurs)
                 (all F (and NODE
                             (atleast 1 min-occurs)
                             (all R (and NODE (atleast 1 min-occurs)))
                             ))
                 )))
```

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

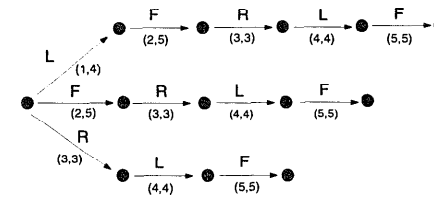

Figure 6: Suffix tree with positional information for "LFRLF"

Extension 2: Pattern Matching with Substring Locations

In cases when the position of 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 subsection, with each edge e now 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 plan "LFRLF". This tree contains, for example, the information that the first occurrence of "LF" ends at position 2 and the last ends at position 5 in the plan "LFRLF". As with substring counts, the pattern-tree generation algorithm can be modified easily to create pattern trees with positional information. The modification does not change the quadratic time and space 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 a DL 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 3 and Corollary 3 still hold for pattern trees with positional information, Corollary 2 does not. In other words, subsumption still implements pattern matching, but no longer supports substring checking. Lemma 4, on the other hand, still holds, i.e., the LCS of two DL pattern trees still computes the intersection of the substrings in the two pattern trees, unioning the interval of locations for each resulting string.

Combining Extensions 1 and 2

The two extended forms of pattern matching based on suffix trees presented thus far 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, one for count information and two for position information.

Further, each extension can be independently used or ignored, depending on the application domain. For example, if only the minimum number of occurrences of substrings is of importance, only F_{max} information need be maintained with a pattern tree. In addition, one can also use our approach for encoding string locations to include information about the location of just the first or just the final occurrence of the string. The LCS of a set of DL pattern trees containing such information would then represent the range of legal locations for the first or last occurrence of each substring present in all the DL trees.

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: $T^{I_{Comb}} := T^{I_F} \cap T^{I_P}$.

Final Remarks

This paper proposed a representation formalism, based on the suffix-tree data structure, to represent and reason about sequences in description logics. Using this representation, subsumption supports substring checking as well as a (modular) range of forms of pattern matching.

We are also exploring ways to support additional information about and operations on strings, such as querying about specific string locations and representing global properties of a string such as length. It would also be desirable for elements of a string to be representable as concepts themselves — for example, to have concepts representing each step in a plan. This would call for extensions to a description logic in which roles could be identified with concepts (similarly to Meta-SB-ONE (Kobsa 1991)). Such an extension would also allow the introduction of hierarchies over characters for strings, e.g., the "D" character to denote a "change of direction" action in a plan (i.e., turn left or right).

Our DL representation also supports the meaningful application of the LCS operation to represented strings. This makes it possible to apply LCS-based DL learning algorithms (Cohen & Hirsh 1994) to sequences. We are currently also exploring suffix-tree-based feature-generation methods to create Boolean features for propositional learning algorithms.

References

Baader, F., and Hollunder, B. 1992. A terminological knowledge representation system with complete inference algorithm. In *Proceedings of Int. Workshop PDK'91*, 67–86.

Borgida, A., and Patel-Schneider, P. 1994. A semantics and complete algorithm for subsumption in the classic description logic. *Journal of Artificial Intelligence Research* 1.

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

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

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

Cohen, W. 1995. Text categorization and relational learning. In *Proceedings of the Twelfth International Conference on Machine Learning.*

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

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

Kobsa, A. 1991. Reification in SB-ONE. In *International Workshop on Terminological Logics.*

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

Nebel, B. 1990. *Reasoning and revision in hybrid representation systems.* Berlin: Springer-Verlag.