ICARUS2 Corpus Query Processor Specification

Markus Gärtner

2020

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

The ICARUS2 Corpus Query Processor (ICQP) is a custom evaluation engine for corpus queries that follow the ICARUS2 Query Language (IQL) specification.

Chapter 1

Notations and Definitions

sequence xxx

tree xxx

search node a search node n is a tuple (c, δ_p, δ_s) where c is the local constraint predicate taking as argument a target node n_i from a sequence, δ_p is the minimum distance to the node's predecessor and

tree search node a tree search node v is a tuple $(c, l,)$

sequence xxx

sequence xxx

sequence xxx

sequence xxx

sequence xxx

1.1 Tree Inclusion

Let T be a rooted tree. We say that T is annotated if each node v in T is assigned a set of annotations $a_1(v)..a_n(v)$ where each annotation function a_i provides its own alphabet Σ_i of available annotation values. This definition is similar to the basic notion of a labeled tree where each node is assigned a single label or value from a common alphabet Σ, but to accommodate the nature of multi-layer annotations in linguistic corpora, we extend this definition. T is further ordered if for any node v its children $v_1..v_n$ follow a globally consistent ordering scheme. If not specified otherwise, all trees in this document are rooted and annotated.

A tree P is said to be included in T, denoted $P \sqsubseteq T$, if deleting nodes in T can yield P.

symbol from an alphabet$\Sigma$and we say that$T$isorderedif a left-to-right orderamong siblings in$T$is given. All trees in this paper are rooted, ordered, andlabeled. A tree$P$is included in$T$, denoted$P \sqsubseteq T$, if$P$can be obtained from$T$by deleting nodes of$T$. Deleting a node$v$in$T$means making the children of$v$children of the parent of$v$and then removing$v$. The children are inserted in theplace of$v$in the left-to-right order among the siblings of$v$. Thetree inclusionproblemis to determine if$P$can be included in$T$and if so report all subtreesof$T$that include$P$.

Chapter 2

Plain Matching

Chapter 3

Sequence Matching

Elements participating in sequence matching:

node singular and optionally quantified element

sequence sequence of elements that adhere to given arrangement

grouping sequence of elements that is optionally quantified as a whole

disjunction two or more alternative elements

3.1 Definitions

Let L be a list, then $N_L = |L|$ is its length and $l_i \in L$ denotes the element at position i of the list where $1 \leq i \leq N_L$. Let T be the list of target elements and S the tree of search nodes.

Utility procedures and functions used in the algorithms of this section:

atom(s) Returns the single wrapped child node for s.

child(s, i) Returns the child node s_i on index i for s.

eval(s, t) Evaluates the inner constraints of search node s on the target item t. The result is a cache entry with a boolean value indicating whether the evaluation was successful.

cacheGet(s, t) Retrieve a cached entry for the evaluation of search node s on target element t. If no entry exist, *nil* is returned.

cacheSet($s, t, entry$) Stores the given *entry* in the cache for the evaluation of search node s on target element t.

next(s) Returns the next search node in the sequence after s.

size(s) Returns the number of child nodes attached to s.

type(s) Returns the type of the given search node, one of the following:[1]

single A single atomic node that is existentially quantified.

sequence A group of adjacent search nodes.

[1]Nested nodes can be obtained with $atom(s)$ for single embeddings or $atom(s, i)$ for indexed elements where $atoms(s)$ returns the number of embedded nodes and $1 \leq i \leq atom(s)$.

group A group of ordered (but not necessarily adjacent) search nodes.

repetition A quantifier (c_{min}, c_{max}) and an associated *atom* search node.

negation An embedded node that must not match.

universal An embedded node that is expected to match all available target elements. Note that universal quantification is only allowed if the node is the only one in the (resolved) global context.

choice A group of nodes that represent logical alternatives.

region An embedded node and associated index interval $i_{start} = first(s)$ to $i_{end} = last(s)$ of legal positions for matching.

spot An embedded node and a single associated index for matching.

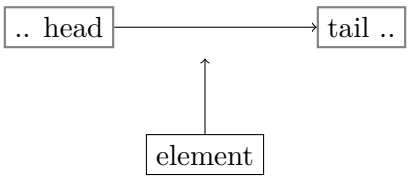
VALUE(*entry*) Extracts the actual boolean result value from a cache entry.

3.2 Rules

This section describes the recursive rules for constructing the object graph of search nodes from the original elements in the query. All rules assume the existence of already processed nodes that resulted in some head part to the left that is connected to a tail to the left, as illustrated below:

Initially the head node is just a generic entry point and the tail simply implements the final acceptance node that ensures that all actual search nodes in the query have been evaluated already.

The elements in the query are effectively processed left to right and for every encountered element a new node is inserted between the current head and tail:



This process is recursive and builds the graph of interconnected search nodes used in the algorithms described in Section 3.3.

3.3 Algorithm

Algorithm 3.1 Local constraint matching

```
1: procedure LOCAL-MATCH(s, t)
2:     entry ← cacheGet(s, t)
3:     if entry = nil then
4:         entry ← eval(s, t)
5:         cacheSet(s, t, entry)
6:     end if
7:     return value(entry)
8: end procedure
```

Algorithm 3.2 Single node matching

```
1: procedure MATCH(s, T, j_start, j_max)                    ▷ j_start ≤ j_max
2:     type_s ← type(s)
3:     if type_s = single then
4:         return j
5:     end if
6:     j ← j + 1
7:     return false
8: end procedure
```

Algorithm 3.3 Single node matching

```
1: procedure MATCH-NODE(s, T, j_start, j_max)               ▷ j_start ≤ j_max
2:     j ← j_start
3:     while j <= j_max do
4:         if LOCAL-MATCH(s, t_j) then
5:             tails ← next(s)
6:             return MATCH(tail, T, j, j_max)
7:         end if
8:         j ← j + 1
9:     end while
10:    return false
11: end procedure
```

Algorithm 3.4 Adjacent sequence matching

```
1: procedure MATCH-ADJACENT(C, T, j_start, j_max)           ▷ j_start ≤ j_max, C ⊆ S
2:     j ← j_start
3:     while j <= j_max do
4:         if LOCAL_MATCH(s, t_j) then
5:             return j
6:         end if
7:         j ← j + 1
8:     end while
9:     return −1
10: end procedure
```

Algorithm 3.5 Quantified matching

```
1: procedure MATCH-QUANTIFIED(s, T, j_start, j_max)          ▷ j_start ≤ j_max
2:     j ← j_start
3:     while j <= j_max do
4:         if LOCAL_MATCH(s, t_j) then
5:             return j
6:         end if
7:         j ← j + 1
8:     end while
9:     return −1
10: end procedure
```

Chapter 4

Tree Matching

Appendices