

Efficient Runtime Example Matching in a Translation System

Eugene Koontz
ekoontz@slt.sel.sony.com
Phone : (408) 955-4616
Fax : (408) 955-6848

February 17, 2000

Paper ID Code:

Topic Area(s): Machine Translation

Keywords: example based translation, grammar compilation

Submission to NAACL or ANLP (you must choose only one)? Workshop on Embedded Machine Translation Systems

Word Count: 2101

Under consideration for other conferences (specify)? None

Abstract

A method for efficiently finding a set of matching examples in an example-based machine translation system is shown. This allows use of a large example database to increase translation quality while mitigating the cost in run-time matching speed that such a large database would otherwise entail.

1 Introduction

Example-based translation uses a database of examples, each of which is a pair of source and target language expressions. Each pair is selected to exemplify some important feature of the source or target language (Furuse and Iida, 1992), (Watanabe, 1992). At run time, the system consults the example database to find the best candidate pair for the input, and this pair is used to generate an expression in the target language. This method allows translation quality to improve by adding relevant examples to the database.

However, an iterative search of the examples at run-time becomes the dominant factor in system performance for a realistically large example database. For embedded devices with limited processing power, a more efficient method of searching the database is needed.

This paper describes a technique for efficiently searching a large database of linguistic examples. It works by doing a compiling the iterative search into simple set operations that are performed at run-time.

2 Background

2.1 Fstructures

Every linguistic sign (a word or a phrase) is represented in the system by a fstructure, similar to the representations used in linguistic theories such as LFG or HPSG. It is a set of feature-value pairs, where each feature is an atomic symbol, and a value is either an atomic symbol or an fstructure. A path π is a list of features $\langle F_1, F_2, \dots \rangle$, where F_2 is a feature within the value of F_1 , and F_3 is a feature within the value of F_2 , and so on.

2.2 The Transfer Component

The input to the transfer component of the translation system is the input fstructure, which represents the syntactic and semantic structure of a string in the input language; the output is another fstructure which is passed to the Generation Component, which turns this fstructure into a string in the target language.

Within the transfer component, there are two subcomponents. The first is the match step. This eliminates from the example set all fstructures that do not satisfy certain boolean queries. The boolean queries to be applied to the example are determined by the transfer grammar, which provides a top-down control structure to the matching process. The transfer grammar is a set of context free rules. A rule can have zero or more boolean queries associated with it. The rule that corresponds to the input fstructure's immediate constituents will be used; next, the rules that correspond to those constituents' constituents, until the leaves of the input fstructure are reached.

The second step, (not discussed in this paper), is cost calculation, which quantitatively ranks examples in the set that satisfy the boolean queries.

3 The Syntax of Boolean Queries

Boolean tests are written by a human linguist and appear like the following example :

```
([$x TYPE] ?= ADV-SENT) AND
([$x ADV HEAD ROOT] ?= [$m INPUT ADV HEAD ROOT]) AND
([$x SENT TYPE] ?= [$m INPUT SENT TYPE]) AND
([$x SENT VP HEAD ROOT] ?= [$m INPUT SENT VP HEAD ROOT] OR
[$x SENT ACTION HEAD ROOT] ?= [$m INPUT SENT ACTION HEAD ROOT])
```

Figure 1: Example Query

Each query is a series of conjuncts; a conjunct in turn consists of one or more predicates. The left side of a predicate contains a path in the feature structure of an example; the right side is an atomic symbol or a path in the input feature structure. (The semantics of queries will be discussed further below in Sections 4 and 5.)

4 The Iterative Method

For a translation system using the iterative method, a query compiler syntactically rewrites each query (such as that in Figure 1) as a C function. These C functions are then executed at run time according to the control of the transfer grammar (Section 2.2). Each example fstructure in the database is instantiated as

one argument ($\$x$) and the input fstructure is instantiated as the other ($\$m$). Note that this interpretation of the query is done only at run time; there is no semantic interpretation of the query at compile-time.

5 The Fast Match Method

The fast match method, unlike the iterative method, considers the semantics of the boolean queries at compile time. We therefore now describe a set semantics for these queries.

These queries have a straightforward translation into Conjunctive Normal Form syntax, as shown in Figure 2.

$$\begin{aligned}
 & (type(X) = ADV-SENT) \wedge \\
 & (adv\ head\ root(X) = input\ adv\ head\ root(M)) \wedge \\
 & (sent\ type(X) = input\ sent\ type(M)) \wedge \\
 & ((sent\ type(X) = input\ sent\ type(M)) \vee \\
 & (sent\ action\ head\ root(X) = input\ sent\ action\ head\ root(M)))
 \end{aligned}$$

Figure 2: Conjunctive Normal Form for Example in Figure 1

The semantics is developed by classifying each boolean term as belonging to one of 3 types. Each type can then be given an interpretation as a set. The boolean operators $\{\wedge, \vee, \neg\}$ are then interpreted as the set operators $\{\cap, \cup, set - compl\}$, respectively.

A CNF statement is of the following form :

$$((t_{1,1} \vee t_{1,2} \dots \vee t_{1,n}) \wedge (t_{2,1} \vee t_{2,2} \dots \vee t_{2,n}) \wedge \dots)$$

where each term t is an atomic term, where an atomic term is as defined in Section 2.2.

Each t is one of the following types :

- example literal - test whether the value of a path function on the example is equal to a certain literal.

For example : $type(X) = ADV-SENT$

- variable - test whether the value of a path function on the example is equal to the value of another path function on the input f-structure. For example : $sent\ type(X) = input\ sent\ type(M)$
- input literal - test whether the value of a path function on the input is equal to a certain literal. For example : $type(M) = ADV-SENT$.

The following table shows the set interpretation for each type. (**E** means the set of all examples.)

| Type of Term | Syntax | Set Semantics |
|-----------------|------------------------|--|
| example literal | $P(ex) = v$ | $ex : P(ex) = v$ |
| variable | $P_1(ex) = P_2(input)$ | $runtime(P_1, input, S : (S_{v1}, S_{v2}...))$ |
| input literal | $P_2(input) = v$ | E if $P_2(input) = v$, \emptyset otherwise |

Table 1: Set interpretation for each type of term

Terms of type example literal can be handled completely at compile time. The expression compiler outputs a set of examples that satisfy the term.

Terms of type variable can be handled partially at compile time, but require the calling of the function *runtime* at run time. The *runtime* function takes three arguments : a path, an input fstructure, and a set of sets $\forall v : (S_{v1}, S_{v2}...)$, where each S_v is the set of examples such that $P_2(ex) = v$. At runtime, it chooses S_v such that $P_1(input) = P_2(ex \in S_v)$. The expression compiler outputs a set of a set of examples.

Terms of type input literal must be handled completely at run time. The expression compiler simply outputs the term.

With the atomic terms thus defined, we can define the set semantics of a conjunct $c = t_1 \vee t_2 \vee \dots$ as :

$$S(c) = \bigcup(t_1, t_2, \dots)$$

. Finally, we can define the semantics of an entire CNF expression $E = c_1 \wedge c_2 \wedge \dots$ as

$$S(E) = \bigcap (c_1, c_2, \dots)$$

. Note that at compile time, we cannot determine the sets corresponding to conjuncts and CNF expressions if there are any variable or input literal terms, since these terms require runtime information about the input.

5.1 Further Optimization

Two stages of optimization may be done on the interpreted expression. First, all example literal terms in a conjunct may be combined into a single set that is a union of all of the sets of the separate literal terms (see Figure 3). In the second stage, both example literal sets and variable sets of sets may be made smaller by intersecting each of them with the intersection of the union of each conjunct (see Figure 4).

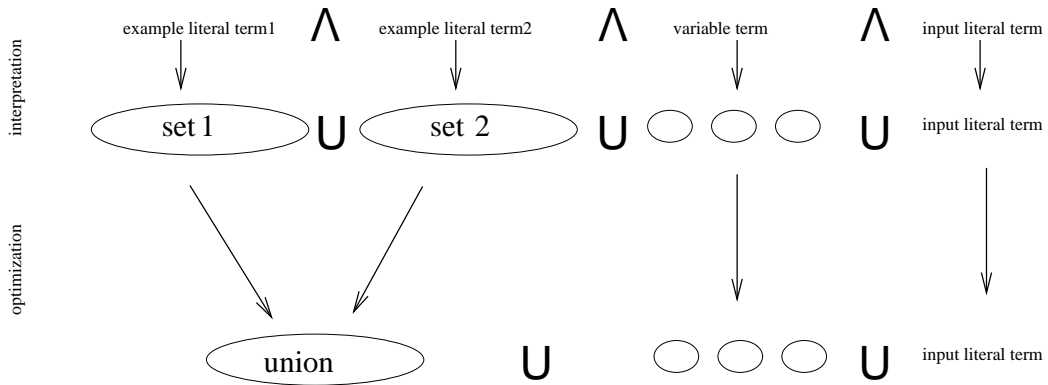


Figure 3: First stage optimization

6 Analysis

Here we compare the performance of the fast match method with the iterative method. The entire transfer process involves the processing of several transfer rules, and the example matching algorithm must be run separately for each transfer rule. The number of transfer rules depends proportionally on the number of nodes in the input phrase structure tree, but this is independent from whether we use the iterative or the fast example retrieval method.

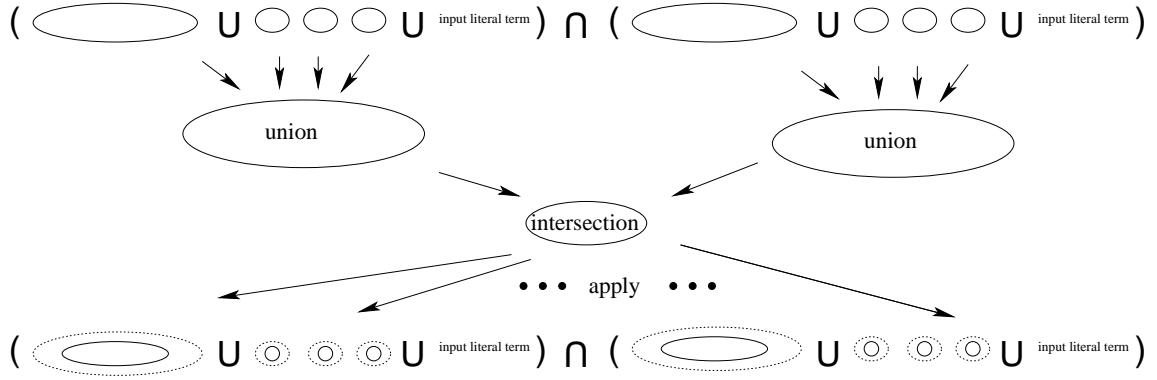


Figure 4: Second stage optimization

We therefore consider only a single transfer rule, having n conjuncts in its match query, where each conjunct has only a single predicate. For the iterative match method, we assume each boolean test takes unit time, and for the fast match, we assume that intersections are done between sorted sets, thus requiring as many operations as the size of the smaller of the two sets.

Note that for expository purposes, we here ignore the optimizations presented in section 5.1; considering these would further reduce the time needed for the fast match method.

6.1 Worst Case

The worst case occurs when every example returns True for every n conjunct. In the case of the iterative match, the number of atomic operations is equal number of boolean tests, which is nE , i.e., the number of terms times the number of examples.

For the fast match, every conjunct is a set, and since every example returns true for each conjunct, we have to perform $n - 1$ intersections on n sets, each of which consists of E members. Again, this means nE atomic operations, which is the same as the iterative match.

6.2 Average Case

For the average case, assume that each conjunct is true for half of the examples. For the iterative match, every example in \mathbf{E} must be checked for the first conjunct; but only $\frac{1}{2}\mathbf{E}$ examples will need to be checked in the next conjunct, since the other $\frac{1}{2}\mathbf{E}$ will have failed the first conjunct. Similarly, the for the third conjunct only $\frac{1}{4}\mathbf{E}$ examples will need to be checked, and so on. The total number of boolean tests is thus $\mathbf{E} \frac{2^{n+1}-1}{2^n}$, or $2\mathbf{E}$ as n approaches infinity.

For the fast match, each conjunct corresponds to a set of $\frac{1}{2}\mathbf{E}$ examples. Assume (pessimistically) that the two sets are identical; then their intersection will also be $\frac{1}{2}\mathbf{E}$ examples. Then n conjuncts will require $\frac{1}{2}\mathbf{E}n$ atomic operations, which will be worse than the iterative match average case. If we assume more optimistically, that the intersection between two adjacent sets is half of either set ($\frac{1}{4}\mathbf{E}$), the total number of operations becomes $\sum_{i=2}^n \frac{1}{i}$, which approaches \mathbf{E} (half of the number of operations as the iterative match).

6.3 Experimental Results

In fact, the boolean tests necessary during the iterative match are much more expensive than the comparing of integers during intersection computation. Our experiences using a run time profiler shows that the real advantage of the fast match compared to the iterative match is greater than the average case presented above; the percentage of time spent in the matching component of the translation system drops from about 67% to 28% (see Table 2)

| | Iterative Match | Fast Match |
|--------------------------------------|-----------------|------------|
| Total Run Time (seconds) | 1056 | 489 |
| Run Time spent in matching (seconds) | 710 | 138 |
| Match time as percent of whole | 67% | 28% |

Table 2: Percentage Time Spent In Matching Component

7 Previous Work

(Oi et al., 1994), taking advantage of the fact that each example lookup can be done independently from the others, uses a SIMD (Single Instruction Multiple Data) parallel hardware architecture to speed up the example retrieval process. The set of examples are partitioned into smaller subsets, each of which is iteratively matched against the input. While their performance results are impressive, their approach is limited to hardware that is too expensive for the consumer market for now. However, it would be interesting to combine the method described in this paper with a SIMD architecture and so realize the optimizations of both approaches.

References

- Osamu Furuse and Hitoshi Iida. 1992. An example-based method for transfer-driven machine translation. In *Proceedings of the Fourth International Conference on Theoretical and Methodological Issues in Machine Translation*.
- Kozo Oi, Eiichiro Sumita, Jared Saia, Osamu Furuse, and Hitoshi Iida. 1994. A massively parallel associative approach for real-time spoken language translation systems. In *JSAI Workshop Notes on Parallel Processing for Artificial Intelligence*, pages 7–12.
- Hideo Watanabe. 1992. A similarity driven transfer system. In *Proceedings of COLING-92*, pages 770–776.