

New Features in PARC's Finite-State Toolkit

Lauri Karttunen

Palo Alto Research Center,
3333 Coyote Hill Road, Palo Alto, CA 94304
karttunen@parc.com
<http://www.parc.xerox.com/ist1/members/karttune>

Abstract. This paper describes some previously undocumented features of FST, Finite-State Toolkit, the more powerful big sister of XFST that was distributed on the CD accompanying the Beesley and Karttunen 2003 book *Finite State Morphology* [1]. It documents a pattern matching algorithm that applies any number of user-defined patterns in parallel to an input text. The output can be in a number of formats including the original text marked up with XML tags. The patterns are defined by regular expressions and compiled into automata. Any number of pattern automata can be combined into a single network, hence the capability of matching several patterns in parallel. The paper also describes a number of previously undocumented optimization techniques that change the physical representation of a network. The user may choose to decrease the memory footprint of the network, possibly with some loss of speed. Alternatively, the speed of the runtime application can be significantly increased, possibly with an adverse effect on the size.

1 Introduction

Finite-state descriptions have been used very successfully to describe the phonology, orthography, and morphology of a large number of languages. Many other basic steps in language processing, ranging from tokenization to named-entity recognition and shallow parsing, can be performed efficiently by means of finite-state networks. FST and its sister application XFST are used for these tasks in many places in academia and in industry.

FST serves two purposes. It is an authoring tool, a compiler that creates finite-state networks (simple automata and transducers). It is also a runtime tool for applying networks to texts. As a compiler FST can be used to create finite-state networks from many types of sources such as word lists and regular expressions. It can perform calculus operations on networks such as concatenation, union, intersection and composition and the resulting networks can be determinized, minimized and optimized in various ways, as discussed in the second half of the paper.

As a runtime tool FST can be used to apply networks to string. For example, if the network is a LEXICAL TRANSDUCER [1] for English, FST can apply the network to an inflected form such as *leaves* to produce the lexical forms

leave+Verb+Pres+3sg, *leave+Noun+Pl*, *leaf+Noun+Pl*. Alternatively, FST can apply the same transducer in the opposite direction to map a lexical specification such as *leave+Noun+Pl* into the corresponding inflected form *leaves*.

2 Pattern Matching with Marking Transducers

FST can also be used for pattern matching. A pattern network can be defined in FST as a regular expression, compiled into a network and applied to texts marking each instance of the pattern. Marking transducers can be compiled with FST's replace operators. In this section we consider three cases in an increasing order of difficulty and show the limits of that approach.

2.1 Social Security Numbers

A Social Security Number contains nine digits and two dashes. The expressions in (1) define a transducer called `SSPat` that wraps the markers `<SocSec>`, `</SocSec>` around every instance of a Social Security number in the input text.¹

```
(1) define SSNo digit^3 "-" digit^2 "-" digit^4;
    define SSPat SSNo -> "<SocSec>" ... "</SocSec>" || Lim _ Lim;
```

For example, `SSPat` maps the input *Is it 898-97-2455?* to the string *Is it <SocSec>898-97-2455</SocSec>?*

2.2 Dates

Social Security numbers are an easy pattern to spot and mark because they are all of the same length. Thus it can never be the case the one valid `SS#` is an initial or final substring of another `SS#`. But many other types of named entity expressions do not have that property. One instance of a pattern may occur as a substring of another longer instance of the pattern. For example, `<Date>March 6</Date>`, `<Date>Thursday, March 6</Date>`, `<Date>March, 6, 2008</Date>` are valid dates but they should not be marked as such individually when they occur inside of a more complete date expression such as `<Date>Thursday, March 6, 2008</Date>`. With obvious definitions for `Day`, `Month` and `Year`, it might seem that a date parser could be defined as shown in (2).

```
(2) define Sep "," (space);2
    define Dates [Day | (Day Sep) Month space Date (Sep Year)];
    define DateParser Dates -> "<Date>" ... "</Date>" || Lim _ Lim;
```

¹ `SSNo` is a network for strings such as *898-97-2455*, `Lim` is defined as the union of the start- and end-of-string marker `.#.` and the union of all non-alphanumeric characters. The arrow is the simple replace operator. The three dots mark the place of the matched Social Security number.

² `(space)` is an optional whitespace character.

The problem with this definition of `DateParser` is that it is non-deterministic. An input such as *Friday, March 8*, produces two outputs: `<Date>Friday, March 8</Date>` and `<Date>Friday</Date>`, `<Date>March 8</Date>`. As shown in Karttunen *et al.* [2], we can solve this problem by substituting for `->` in (2) the left-to-right, longest-match replace operator, `@->`, first introduced in [3]. With this change, we only get the first output. Any potential match that is inside another larger match for the pattern does not count as a valid match. Even if we let the years range from 1 to 10000, the resulting `DateParser` transducer is quite small: 146 states, 9226 arcs.

2.3 Names

The use of the `@->` operator works well in cases where the pattern consists of a small number of elements. To parse dates we only need in our lexicon the names of the seven days of the week, the twelve months, the ten digits, and the comma and whitespace characters. The year component in (2) contains 10000 year strings but they are encoded by a tiny network with just 5 states and 39 arcs. But a lexicon of ten thousand first or last names, while much too small for practical purposes, is much too large for pattern matching with a marking transducer of the type introduced in the preceding two subsections. Consider the simple definitions in (3).

```
(3) define Name FirstName | (FirstName space) LastName;
    define NameParser Name @-> "<Name>" ... "</Name>";
```

Here `Name` is defined as consisting either a first name or a last name preceded by an optional first name. The `NameParser` transducer maps, for example, *George* to `<Name> George</Name>`, *Bush* to `<Name> Bush</Name>` and *George Bush* to `<Name> George Bush</Name>` following the left-to-right, longest-match principle.

If `FirstName` and `LastName` are compiled from lists containing around a couple of hundred names, the compilation goes quickly and the size of the resulting network is small. However, if the size of the word list grows, this approach to name recognition quickly becomes impractical. For real applications we would have lists of tens or thousands of names. It is evident that another approach is needed.

Table 1 tracks the developments as the size of the lexicon increases.

First names	Last names	Compilation time	States/Arcs	Size
100	100	1 sec	479/2916	359 Kb
1000	1000	22 sec	3609/211821	2.6 Mb
5000	5000	8 min	12799/791859	9.7 Mb

Table 1. Compilation Statistics for `NameParser`

3 A New Algorithm

The lesson of the experiments in section 2 is that the left-to-right, longest match principle becomes computationally very expensive when the size of the pattern components increases. Using the state and arc space of the marking transducer to enforce the matching regimen is not practical except in cases such as the date parser where all the components of the pattern are small. In this section we present an algorithm that achieves the same result as marking transducers but scales up much better. Let us call it the `PMATCH ALGORITHM`. It is based on three ideas.

1. The left-to-right longest-match principle is implement in the apply algorithm.
2. Only the end of a pattern needs to be marked.
3. The same string is a match for more than one pattern.

To illustrate these ideas, let us consider a very simple case. We know that *Sara Lee* is a name of a company but there are also many people named *Sara Lee*. We would like to recognize all instances of *Sara Lee* and tag them both as a person and as a company. The definitions are given in (4).

```
(4) define CTag "</Company>":0;
    define PTag "</Person>":0;
    define Persons {Sara Lee};
    define Companies {Sara Lee};
    define Patterns Companies CTag | Persons PTag;
```

The resulting `Patterns` network consist of a linear path for the string *Sarah Lee* leading to a penultimate state that has to arcs, one has the label `</Company>:0`, the other is labeled `</Person>:0`.³ Both of them lead to a final empty state.

3.1 The basic *pmatch* routine

Let us assume that the *pmatch* algorithm is applying the `Patterns` network to the input *He works for Sara Lee*. The algorithm starts at the beginning of the input and at the start state of the `Patterns` network and tries to match the first symbol, *H* of the input against the arcs of the current pattern state. If it finds a match, it advances to the arc's destination state and to the next symbol in the input string. If it fails to find a match, as the case is here, it writes the *H* into the output buffer and advances to the next symbol, *e*. Before starting the matching process, *pmatch* checks the left context at its current position. The default requirement is that a pattern should start from the beginning of a string or after a non-alphanumeric symbol. Because the *e* and the following space do not meet the starting condition, they are appended to the output buffer without any attempt to match them. In the case at hand, the matching attempts fail until the

³ The 0 on the lower (right) side of the double label represents ϵ , the empty string.

process reaches the letter *S*. From there on, the input matches the path leading to the penultimate state. At that point we are at the end of the input string but both of the tag arcs yield match because they have an epsilon on the input side. Having reached a final state over a tag arc with no input left to check, *pmatch* now takes note of the fact that the next input symbol, the period, satisfies the default ending condition.⁴ It reads off the tag on the output side of the label, creates the corresponding initial XML tag on the fly and inserts it into the output buffer. Depending on the order of the tag arcs in the penultimate state, the start tag is now either `<Company>` or `<Person>`. Let us assume the former. In that case *pmatch* first inserts the initial tag into the output buffer and copies the string that it has match into the output buffer terminating with the closing tag. At that point the output buffer contains the string *He works for <Company>Sarah Lee</Company>*. When *pmatch* processes the second closing tag, it takes note of the fact that it already has one analysis for the string and wraps the second pair of initial and final tags around the first pair. The final output of the process is *He works for <Person><Company>Sara Lee</Company></Person>*.

Assume now that we add two new names, *Sara* and *Lee* to the **Persons** list in (4). Given the same input as before, we now get a successful match at the point where the output buffer contains the string *He works for <Person>Sara</Person>*. But in this case the final state of the pattern network has an outgoing arc for space. Because the *pmatch* algorithm always looks for the longest match, it ignores this preliminary result and tries for a longer match. At the point where it comes to the `<Company>` and `<Person>` tags, the preliminary output gets overwritten and the final output is what we just saw. Because the *pmatch* algorithm never starts a search in the middle of another search, it will not try to find a match for *Lee* in this case. Consequently, strings that are substrings of some successfully matched longer string are always passed over.

3.2 Observations on the *pmatch* algorithm

Compared to the marking approach described in 2 the *pmatch* algorithm has many advantages. Enforcing the longest match regimen in the apply routine is much more efficient than hardwiring the constraint into the transducer when the pattern is made up of large subcomponents. As Table 1 shows The marking approach become impractical when the number of first and last names goes beyond 5000 each. With *pmatch* we have no problem at all. For comparison we used the formula in (3) to build a simple pattern network for *pmatch*.

```
(5) define Name FirstName | (FirstName space) LastName;
    define NameParser Name </Name>:0;
```

Table 2 shows the compilation statistics for 9029 first names and 28810 last names. Table 1 tracks the developments as the size of the lexicon increases. Because the *pmatch* networks are often small, it is possible to combine several

⁴ The starting and ending conditions play the same role as the **Lim** constraints in (1) and (2).

First names	Last names	Compilation time	States/Arcs	Size
9029	28810	1 sec	22918/ 81348	1.3 Mb

Table 2. Compilation Statistics for `NameParser`

patterns into a single network and run them in parallel. This is an important advantage over systems that require a separate run for each pattern.

3.3 Output Options

The default output mode in FST for pattern matching is to wrap XML tags around each match of the pattern. In this section we cover briefly the other output options. The output mode is controlled by five interface variables, defaults in parentheses: `mark-patterns` (on), `locate-patterns` (off), `delete-patterns` (off), `extract-patterns` (off).

default : Wrap XML tags around the strings that match a pattern, for example, `<Actor>Grace Kelly</Actor>`.

stand-off markup : Leave the original text unmodified. Produce an output file that indicates for each match its beginning byte position in the file, for example, `78|11|Grace Kelly|<Actor>`. Set `locate-patterns` on.

extraction : Extract from the file all the strings that match some pattern. Output them with their tags. For example, `<Actor>Grace Kelly</Actor>`. Ignore all the rest. Set `locate-patterns` off, Set `extract-patterns` on.

redaction : Ignore strings that match some pattern. Print the rest. Set `extract-patterns` off, set `delete-patterns` on.

4 Optimizations

The internal representation of states and arcs in FST can be modified in several ways to make reduce the size or to improve the speed of applications. The effect of any of the techniques described in this section are highly variable. Networks that are very dense in terms of the arcs/state ratio can sometimes be dramatically reduced in size by encoding arc space in a different way even if the state count goes up. Sparse network tend to receive less benefit if any at all. In this section I will briefly describe some of these optimizations. All the operations described here are lossless and reversible.

4.1 Optimizations for size reduction

FST includes four operations that are designed to reduce the number of arcs.

label set reduction : Compute equivalence classes of arc labels and represent each class by one label only. Two labels belong to the same class if they always occur together and and their arcs always have the same destination state.

arc optimization : A heuristic method for reducing the number of arcs, typically at the cost of adding more states.

arc sharing : Let some arcs be shared by more than one state. Arc sharing presupposes arc optimization.

compaction : Minimize the size of state and arc structures. Achieves the best compression but slows down the application speed significantly.

4.2 Optimizations for speed

vectorization : In the standard representation, a state consists of a list of arcs. An arc has a label, a pointer to a destination state and a pointer to the next arc. Most FST algorithms work only on the standard representation. A vectorized state has no arcs. Instead it has a vector of destination pointers that accessed directly by symbol labels. In typical applications, dense states (states with long arc lists) get vectorized while sparse states remain in the standard representation.

4.3 Size vs. speed

Table 3 illustrates the effect of the manipulations discussed above on the size of a network and the speed of the application. The network in question is a “sentence breaking” transducer that introduces sentence boundaries. Its large size is due to the long lists of abbreviations, numerical expressions and other expressions that are exceptions to general punctuation rules. The application speed was measured on a 710K text file.

Representation	States/Arcs	Size	Speed
Standard, no optimization	5302/596017	7.2 Mb	23 sec.
Label set reduction	5302/28033	3.5 Mb	11 sec.
Arc optimization	8364/83301	1.1 Mb	12 sec
Arc sharing	5302/75679	994 Kb	11 sec
Compaction	5302/596017	518 Kb	3 min 3 sec
Vectorization	5302/5312	6,9 Mb	2 sec

Table 3. Comparison of the Effect of Various Optimizations to Size and Speed

In this particular case, Arc optimization with Arc sharing are the best options if size is important. Vectorization is the best option if speed is more important than space.

5 Conclusion

The basic design of the PARC’s finite-state toolkit goes back to the late 1980s and early 1990s. FST and XFST have been used successfully in many research projects

and industrial applications. The original domain of application of PARC's finite-state technology was spell-checking and morphology. In recent years the focus has shifted to other applications such as pattern matching. This article highlights and documents for the first time some of the features developed recently and some of old features for which there is little previous documentation.

References

1. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI Publications, Palo Alto, CA (2003)
2. Karttunen, L., Chanod, J.P., Grefenstette, G., Schiller, A.: Regular expressions for language engineering. *Journal of Natural Language Engineering* **2**(4) (1996) 305–328
3. Karttunen, L.: Directed replacement. In: ACL'96, Santa Cruz, CA (1996) `cmp-lg/9606029`.