# Finite state applications with Javascript

*Mans Hulden*[1] *Miikka Silfverberg*[1] *Jerid Francom*[2]
(1) University of Helsinki
(2) Wake Forest University
`mans.hulden@email.arizona.edu`, `miikka.silfverberg@helsinki.fi`, `francojc@wfu.edu`

ABSTRACT
In this paper we present a simple and useful Javascript application programming interface for performing basic online operations with weighted and unweighted finite-state machines, such as word lookup, transductions, and least-cost-path finding. The library, *jsfst*, provides access to frequently used online functionality in finite-state machine-based language technology. The library is technology-agnostic in that it uses a neutral representation of finite-state machines into which most formats can be converted. We demonstrate the usefulness of the library through addressing a task that is useful in web and mobile environments—a multilingual spell checker application that also detects real-word errors.

KEYWORDS: Finite-state technology, Javascript, spell checking, perceptrons.

## 1 Introduction

Finite-state machine (FSM) technology is widely used in language-processing and FSMs are applied in many contexts ranging from text processing to more complex tasks. Currently, several high-quality toolkits exist for the manipulation and construction of weighted and unweighted finite-state machines, such as *foma* (Hulden, 2009), *HFST* (Lindén et al., 2009), *OpenFST* (Allauzen et al., 2007), *SFST* (Schmid, 2006), and *xfst* (Beesley and Karttunen, 2003), to name a few. These toolkits, once compiled, also provide programming interfaces for the real-time use of automata in language processing tasks. However, for online and mobile applications, the use of finite-state machines is complicated by the unavailability of generic application interfaces for applying the finite-state machines built with any of the above toolkits. In the following, we demonstrate a simple Javascript API that allows one to take advantage of finite-state technology in a toolkit-agnostic environment. The API can take advantage of any FSM represented in the AT&T toolkit format (to which all of the above tools can export). We demonstrate its real-world usefulness with a Javascript spell checking application that offers both traditional spell checking functionality—marking of incorrectly spelled words using a simple unweighted automaton—and more advanced real-word error (RWE) correction, implemented through a perceptron algorithm and encoded as weighted finite automata.

## 2 The *jsfst* API

The *jsfst* API allows basic operations on FSMs, both weighted and unweighted. These include: checking whether a word (string) is accepted by an automaton, performing a string transduction by a transducer, and finding, given an input string, the least-cost path through a weighted automaton.

It is assumed that the user has stored the required FSMs as a compressed string representation (see below), after which the FSM may be expanded in memory and subsequently used in applications through the API. We also provide a tool for converting FSMs from the widely-used AT&T textual format (Mohri et al., 1997) into a compressed Javascript format discussed below. The workflow is simple; the user defines one or several compressed FSM variables, after which operations on the FSMs can be performed through the API:

```
<script type='text/javascript' src='./jsfst.js'></script>
<script type='text/javascript' src='./my_automaton.js'></script>
// my_automaton.js: var mycompressedFSM = "moomoom4se8us7loo9s7goo9|4FEBC|9280" ;
...
// Uncompress
var myNet = jsfst_uncompress_unweighted(mycompressedFSM);
...
// Check if a word is accepted by the machine
var accept = jsfst_apply_unweighted_automaton(myNet, myString);
...
```

### 2.1 Compression of FSMs for online use

When dealing with online and mobile applications, memory restrictions tend to be more severe than under other environments. This calls for compression of finite-state networks during transmission and storing. Often, Javascript code is transferred from server to client in a *gzip*-compressed format which offers some compaction; however, this can be much improved upon by judicious encoding of the finite-state machines. In the current work, we rely on a

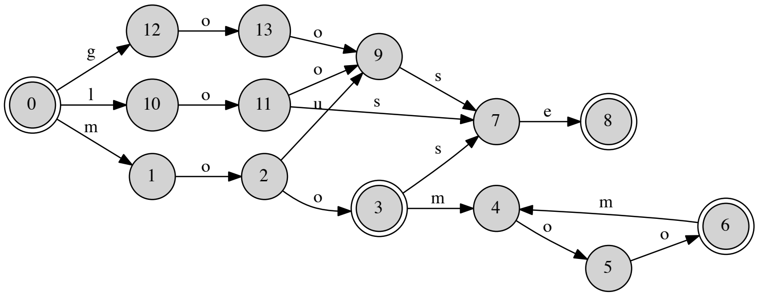


Figure 1: *Finite state machine represented as the string:* `moomoom4se8us7loo9s7goo9|4FEBC|9280`.

string representation of FSMs that, together with standard compression methods (*gzip*), yields compression ratios of about 1:10 compared with completely uncompressed material, and about 1:3 compared with other textual formats which are compressed, such as the widely used AT&T FSM format followed by *gzip*.

We represent the FSMs as a string in three parts, **A|B|C**. The string encodes the result of printing out a depth-first-search through an FSM, where states are only mentioned when strictly necessary for its later reconstruction. Similar methods have been used successfully in other circumstances where space has been at a premium (Karttunen, 1990). The parts encode the following information:

- Part A: represents a depth-first-search through the automaton. Each arc is mentioned once whenever it is encountered. A state number is given every time a backtrack occurs during the DFS, or at the end. States not mentioned are assumed to be constructed in ascending numerical order, one for each transition that is not followed by a backtrack.
- Part B: a sequence of bits, 1 for each arc that is the last one in the source state, otherwise 0, converted into hexadecimal symbols, 4 bits in each symbol.
- Part C: a sequence of bits, 1 for each state that is final, otherwise 0, converted into hexadecimal, 4 bits in each symbol.

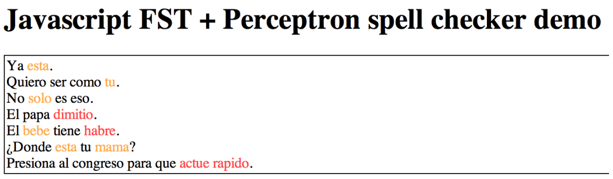
For example, in our compressed format, the FSM in figure 1, which encodes the regular expression `/(moo)*|(mo(o|u)se|goose|loo?se)/`, is represented as the concatenation of the three strings:

| | | |
|---|---|---|
| (A) | `moomoom4se8us7loo9s7goo9` | (is a DFS path) |
| (B) | `4FEBC` | (is 0100 1111 1110 1011 11(00) in binary) |
| (C) | `9280` | (is 1001 0010 1000 00(00) in binary) |

Transducers are encoded similarly, but differing input-output pairs are separated by a colon. Weighted machines additionally have the weight specified in parentheses after each transition. Likewise, weighted machines have their final weight (in **C**) given explicitly, instead of in the bit-encoded format. Note that the original state numbering of any given FSM may change when this encoding is performed by a DFS, as states are assigned running numbers in the order they

| Language | .cmp.gz | .cmp | .att.gz | .att |
|---|---|---|---|---|
| **Basque** | **797K** | 2.1M | 2.3M | 8.2M |
| **English** | **632K** | 1.3M | 1.9M | 6.1M |
| **Finnish** | **1.3M** | 2.9M | 3.7M | 13M |
| **German** | **8.2M** | 23M | 22M | 84M |
| **Spanish** | **194K** | 415K | 591K | 2.0M |
| **Swedish** | **433K** | 994M | 1.3M | 4.0M |

Table 1: *File sizes of some morphologies from which word-surface forms have been extracted in different formats for use in a online spell-checking application. Here* ***.cmp*** *represents the string format FSM compression,* ***.gzip*** *a gzipped file, and* ***.att*** *the AT&T file format.*



**Javascript FST + Perceptron spell checker demo**

Ya esta.
Quiero ser como tu.
No solo es eso.
El papa dimitio.
El bebe tiene habre.
¿Donde esta tu mama?
Presiona al congreso para que actue rapido.

Figure 2: *Screenshot of combined standard and RWE spell checker in Spanish: incorrectly spelled words are marked interactively in red, while real-word errors are marked in orange.*

are encountered. This allows for a very compact encoding of sets of words and FSMs in general. For example, the basic acyclic Spanish morphological dictionary used in subsequent examples contains 571,129 words. Represented as an FSM encoded in our compressed format, and then gzipped, it occupies 198,904 bytes. This is equivalent to 2.79 bits per word. Table 1 shows the size of various morphological transducers, converted into automata by extracting only the valid word-forms, and then encoded in various compressed and uncompressed formats.

## 3 Example application: spell checking

A classic byproduct of encoding a morphological analyzer as an FSM is the ability to quickly implement spell checking tools. This involves extracting the domain or range of the morphological transducer, yielding an automaton that (presumably) contains only valid surface forms of words. This automaton can then be consulted for checking correctness of words—usually with much larger coverage than word lists can provide. For our experiments, we have done so using morphological transducers for Basque (Agirre et al., 1992), English, Finnish (Pirinen, 2011), German (Schmid et al., 2004), Spanish (Carreras et al., 2004), and Swedish. As is seen from table 1, except for the German transducer which encodes many circumfixation phenomena and is rather large to begin with, the compressed sizes of the automata are small enough to be used and integrated into, for instance, web-based text editing environments. [1]

### 3.1 Real-word errors with perceptrons encoded as weighted automata

To demonstrate more advanced usage, we have implemented a real-word-error-aware spell checker using weighted automata. Catching real-word spelling errors is a difficult problem

[1]Performance is also reasonable. When using a determinized automaton, we can check correctness of an average of 49,000 words/s using the Javascript API (vs. 1,400,000 words/s in a C implementation using the *foma* API).

(see Hirst and Budanitsky (2005) for an overview). However, in restricted domains, fairly high accuracies can be achieved (Yarowsky, 1994). One of these is the detection of diacritic placement errors in Romance languages. We focus here on Spanish, as it is arguably at the easier end of the spectrum, and a comfortable level of certainty in error marking can be achieved.

When restoring diacritics, we divide words into three classes: (1) words whose correct orthographic form has no diacritics, for example "para" (for); (2) words whose correct form has diacritics, but there is only one possible form, for example "según" (according to); (3) words that have two orthographically correct forms with and without diacritics, for example "que" (that) and "qué" (what).

In the unambiguous cases, a list of correct forms is sufficient for restoring the diacritics. For ambiguous words, one of the forms for example "qué" is the diacritized form and the other one, "que", is the non-diacritized form. It is possible to train a binary classifier based on contextual features, which decides whether the form should be diacritized or not. The contextual features that are used in the classifier are: (1) the word form itself, for example *WORD* = *que*; (2) the neighboring words, for example *NEIGHBOR* = *por*; and (3), whether the word occurs at a sentence boundary *NEIGHBOR* = *##*. These features are usually sufficient to choose between alternative forms. For example, in a case like '... para él.' the ambiguity of 'el' (article) vs. 'él' (he/him) is resolved by the presence of a sentence boundary.

We use a logistic classifier to classify ambiguous words. Model parameters are estimated using the averaged perceptron algorithm (Collins, 2002) and all features whose parameters are non-zero are compiled into a weighted FSM. Using the word lists and the classifier, the system achieves 99.4% accuracy when determining whether a form should be diacritized or not.[2]

To reduce the size of the resulting FSM, it is useful to filter features. For training data consisting of approximately 14M words, we filtered out features for words occurring less than 60 times. This reduced the accuracy of the system from 99.4% to 99.1%, while the size of the feature FSM was reduced from 4.7M to 1.4M in AT&T text format. The size is further reduced to 141K by representing the transducer in compressed format as explained in section 2.1. The performance of the RWE detection is similar to that of checking a word against an automaton, as the perceptron is encoded as a deterministic weighted automaton (30,000 lookups/s). However, for each word to be checked, three separate lookups are needed (one for each feature). The combined use of both a basic spell checker and an RWE error checker is shown in figure 2.

## 4 Conclusion

Javascript-based tools have already demonstrated their usability: there are now Javascript-based WYSIWYG text editors and PDF readers, for example. As we have demonstrated, it is also possible to create Javascript-based language technology tools that are sufficiently fast and compact for heavy use even in, for example, mobile devices.

It is possible to encode a variety of machine learning and language modeling paradigms as FSMs, which means that a general, efficient Javascript API for FSMs provides access to a broad swathe of cutting-edge technology. Since Javascript is widely supported, tools using such an API are also easy to insert in a variety of applications. This, in turn, offers the potential to promote the creation of tools with a much wider coverage of languages, including less-resourced minority languages.

[2]We use a section of the Spanish GigaWord corpus (Graff, 2011) for training data, from which 10% was held out for tuning, and another 10% set aside to be used as evaluation data.

## References

Agirre, E., Alegria, I., Arregi, X., Artola, X., de Ilarraza, A. D., Maritxalar, M., Sarasola, K., and Urkia, M. (1992). Xuxen: A spelling checker/corrector for Basque based on two-level morphology. In *Proceedings of the third conference on Applied natural language processing*, pages 119–125. Association for Computational Linguistics.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata*, pages 11–23. Springer.

Beesley, K. R. and Karttunen, L. (2003). *Finite state morphology*. CSLI, Stanford.

Carreras, X., Chao, I., Padró, L., and Padró, M. (2004). Freeling: An open-source suite of language analyzers. In *Proceedings of the 4th LREC*, volume 4.

Collins, M. (2002). Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *Proceedings of EMNLP '02*, pages 1–8.

Graff, D. (2011). Spanish Gigaword Third Edition (LDC2011T12). *Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA*.

Hirst, G. and Budanitsky, A. (2005). Correcting real-word spelling errors by restoring lexical cohesion. *Natural Language Engineering*, 11(01):87–111.

Hulden, M. (2009). Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, pages 29–32. Association for Computational Linguistics.

Karttunen, L. (1990). Binary encoding format for finite-state networks. *Technical Report, Palo Alto Research Center*, (P90-00019).

Lindén, K., Silfverberg, M., and Pirinen, T. (2009). HFST tools for morphology–an efficient open-source package for construction of morphological analyzers. In *State of the Art in Computational Morphology*, pages 28–47. Springer.

Mohri, M., Pereira, F., Riley, M., and Allauzen, C. (1997). AT&T FSM library-finite state machine library. *AT&T Labs-Research*.

Pirinen, T. (2011). Modularisation of Finnish finite-state language description—towards wide collaboration in open source development of morphological analyser. In *Proceedings of NoDaLiDa*, volume 18.

Schmid, H. (2006). A programming language for finite state transducers. *Lecture Notes in Computer Science*, 4002.

Schmid, H., Fitschen, A., and Heid, U. (2004). SMOR: A German computational morphology covering derivation, composition and inflection. In *Proceedings of LREC 2004*, pages 1263–1266. Citeseer.

Yarowsky, D. (1994). Decision lists for lexical ambiguity resolution: Application to accent restoration in Spanish and French. *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 88–95.