OpenFST library extension
for String-to-Dependency
Statistical Machine Translation

F. Flego, and B. Byrne
CUED/F-INFENG/TR.694
July 11, 2014

Cambridge University Engineering Department
Trumpington Street
Cambridge. CB2 1PZ
England
E-mail: {ff257,wjb31}@eng.cam.ac.uk
## Contents

1 Introduction ........................................... 2

2 Description of dependencies rules: phrase, glue and hiero rules 2

3 Operation examples: combining dependency structures ..... 5
   3.1 Hiero rule example .................................. 5
   3.2 Glue rule example .................................. 7

4 Dependency tags as weights in FST .......................... 7
   4.1 Differences and limitations with respect to previous work 8
   4.2 “Controlled FST topology” and dependency weights encoding 9
      4.2.1 Weight pushing on “Controlled FST topology” ........ 9
      4.2.2 Division between POSs .......................... 11
      4.2.3 Division between dependency vectors ................ 11
      4.2.4 Composition with a rho matcher ........................ 12
      4.2.5 Dependency weights encoding and FST optimisation .... 13

5 Example of hiero rule application .......................... 15
   5.1 Division between dependency vectors: two non-terminals case 16

A List of glue rules ........................................ 23

B POS mapping .............................................. 25
1 Introduction

This document describes an OpenFST library which implements the string-to-dependency algorithm [3].

In [3] Part Of Speech (POS) and word dependencies information (obtained from a parser) are added to the target side of rules in the translation hiero grammar as additional features to improve translation quality.

When a set of translation hypotheses is represented using a FST, as in HiFST [1], arc weights are obtained from the cost of each grammar rule applied during decoding (other components as, for example, the LM hypothesis score, can also contribute to the final weights). FSTs provide a compact representation of the translation space so that operations such as partial hypotheses concatenation, pruning, etc. can be efficiently applied by means of standard FST operations as, for example, concatenation and pruning.

The library described here encodes POS and word dependencies information using FST string weights. Although in general such weights do not represent a valid semiring, it is possible to set some constraints on the FST topology so that the weight class with the necessary (extended) binary operations (times, plus, etc.) will have the semiring properties, allowing the above operations to be applied.

In the following sections the basic concepts of the string-to-dependency algorithm are reviewed. An extended hiero grammar to handle the dependency information is presented and examples of grammar rule application in terms of operations on FSTs are given.

Next, the limitations of using such dependency semiring are discussed and a simplified approach is proposed to overcome them.

2 Description of dependencies rules: phrase, glue and hiero rules

A grammar phrase rule for machine translation can be written as (not showing the left hand side field used to distinguish between different rule types)

\[
\text{SRC TGT W}
\]

where \(\text{SRC}\) is a sequence of symbols representing the source language, and \(\text{TGT}\) is the translation in the target language. A set of weights \(\text{W}\) is associated to this rule and will contribute to the cost of the final hypothesis.

The above rule can be extended with the POS and word dependencies information by adding them as additional fields to provide

\[
\text{SRC TGT W POSs DEPs}
\]

where \(\text{DEPs}\) is an integer vector describing the word dependency tree of the target side\(^1\), and \(\text{POSs}\) is used for the POS information.

\(^1\)DEPs[i] = j means that TGT word in position i depends on TGT word on position j.
Shen’s string-to-dependency algorithm uses three types of rules: phrase, hiero and glue rules (showing only TGT, POSs, and DEPs):

- **phrase**: A dependency phrase rule example and the related target side dependency tree is shown below.

\[
\begin{align*}
\{g \ h \ i\} & \text{ POS5 } \{1, \ -1, \ 1\} \\
\end{align*}
\]

where target word h has dependency index -1 to indicate that it is the tree head and was tagged as POS5 by the parser\(^2\). A distinction is made in [3] between a fixed structure, where DEPs describes a tree with a single head (as in the example above), and a float structure which is a dependency tree with more than one head (e.g. Figure 3, box R5). In the latter case the POS field holds a special code to indicate whether the float structure is a left or a right structure (see [3] for details).

- **hiero**: The SRC and TGT of a hiero rule represent sequences of terminals (actual words) and one or more non-terminals, indicated here with \(X_i\), which behave as pointers to other rules previously applied (or a set of them represented by another FST). An example of a hiero rule with two non terminals is

\[
\{a \ X1 \ f \ X2 \ j \ k\} \ (\text{POS1}\text{?POS2}\text{?POS3}) \ (2, \ 2, \ -1, \ 4, \ 2, \ 4)
\]

In this case the first value in the POSs tag (POS1) is the POS of the tree head (f) for the target side. The following values separated by "?" are the expected POS values of the head nodes of the trees pointed to by the non-terminals. These values are obtained from the training parallel corpus during rule extraction. In the example of Figure 1 POS2 and POS3 values are expected for words c and h, respectively.

![Figure 1: Hiero rule with two non-terminals pointing to two phrase rules.](image)

\(^2\)For simplicity, POS\(i\) variables (with index \(i\)) are used in the following examples to indicate a given POS value.
The result of applying the above rule is shown in Figure 2. A cost penalty is added to the final cost for each mismatch between a non-terminal expected POS value and that of the structure the non-terminal points to.

![Dependency tree](image)

**Figure 2:** Dependency tree resulting from application of hiero rule in Figure 1.

- **glue**: this rule behaves similarly to the hiero rule above and is responsible for the Left/Right Adjoining/Concatenation combinatory category operations (LA, RA, LC, RC) described in [3].

In this case SRC and TGT only have two non-terminals, $X_1$ and $X_2$. Depending on the operation performed, the tree described by the DEPs vector can be: two isolated nodes (e.g. Figure 6, box R6) which implies that a concatenation is applied; or one node which depends on the other one for the adjoining operation (e.g. LA in Figure 3, box R4).

![Glue rule applied to two phrases](image)

**Figure 3:** Glue rule applied to two phrases. Top left: LA, Top right: LC.

When LC or RC is applied, a *float* dependency structure is obtained, which is assigned a special POS tag to indicate whether it is a left or a right *float* structure. In the case of LC between a left *float* and a *fixed* structures for example (e.g. Figure 3, box R6, R5 and R7), the POSs tag of the related glue rule is given by

$$\text{POSs} = \text{LEFT?LEFT?HEAD}$$
For glue rules no POS mismatch is computed. Instead the expected values in the POS tag specify what type of structures the non-terminals can point to (only matching rules are therefore allowed).

The POS tag above specifies that the POS of the resulting structure is LEFT (a left float structure), the first non-terminal is expected to point to a LEFT structure (float), and the second non-terminal is expected to point to a fixed (HEAD) structure.

When an adjoining operation is applied, the POS of the resulting structure will be that of the structure pointed to by \( X_1 \) (for RA) or that of the structure pointed to by \( X_2 \) (for LA). Therefore the final POS cannot be determined in advance, before the actual structures pointed to are known. For example, the LA operation between a LEFT (float) and a HEAD (fixed) structures has the following POS tag

\[
\text{POS} = \text{HEAD?LEFT?HEAD}
\]

In this case the first HEAD specifies that the final POS is unknown until the non-terminals are substituted with the structures they point to. For the LA operation shown in Figure 3 box R4, R5 and R7 the final POS will be that of the structure pointed to by \( X_2 \), that is, the POS of letter h.

A list of all possible glue rules is given in Appendix A.

3 Operation examples: combining dependency structures

The result of applying a hiero or a glue rule, other than implying a particular combination of the dependency structures involved, can result in a penalty being added to the resulting score. This happens when the expected POS tags for the non-terminals in a hiero rule do not match those of the structures pointed to or, in the case of hiero rules, when an invalid operation (see [3]) is performed and the resulting structure is a NULL structure (see list in Appendix A). Two simple examples are described here.

3.1 Hiero rule example

The first example, shown in Figure 4, is the application of a hiero rule, \( R1 \), with two non-terminals \( X_1 \) and \( X_2 \), pointing to two phrase rules, \( R2 \) and \( R3 \) defined as
The POS tag of the hiero rule specifies that the final POS will be POS1, while the expected POS tags for the tree head nodes pointed to by $X_1$ and $X_2$, are POS2 and POS3, respectively.

![Figure 4: Hiero rule with two non-terminals pointing to two phrase rules.](image)

The resulting DEPs vector is given by $\{5, 2, 5, 2, 2, -1, 7, 9, 7, 5, 9\}$ which describes the tree in Figure 5.

![Figure 5: Dependency tree resulting from application of hiero rule in Figure 4.](image)

In the above example the structures pointed to by the hiero rule non-terminals are, respectively, string “bcde” with POS4 and string “ghi” with POS5. For any
mismatch occurring between POS2 and POS4, and between POS3 and POS5, a penalty is added to the final score.

### 3.2 Glue rule example

Two examples of glue rule application are shown in Figure 6, where R4 is used for the LA operation, and R6 for LC. Their respective POS tags are defined as follows:

- **R4**: \( \text{POSS} = \text{HEAD?LEFT?HEAD} \)
- **R6**: \( \text{POSS} = \text{LEFT?LEFT?HEAD} \)

Applying R4 produces the result in the left box of Figure 7 where the tree head POS is given by the head node in R7.

![Figure 6: Glue rule applied to two phrases. Top left: LA, Top right: LC.](image)

When R6 is applied instead, a LEFT (float) structure is obtained, which is shown in the right box of Figure 7.

![Figure 7: Dependency trees resulting after applying glue rules in Figure 6. Left: LA. Right: LC.](image)

### 4 Dependency tags as weights in FST

This section discusses how FST weights can be extended to incorporate the dependency information described in Section 2, and how the string-to-dependency
algorithm can be implemented using FST operations such as Determinization, Min-
imization, etc. provided by a modified version of the standard OpenFST library.

4.1 Differences and limitations with respect to previous work

The approach is similar to that used in [2] where weights based on a categorial
semiring are used on FST arcs to represent POS tags only. In that work the (left)
categorial and the tropical semiring are joined into a Lexicographic semiring whose
properties (weak divisibility, path property, etc) permit operations such as Deter-
minization, Minimization, etc.

However, when dependency information is encoded as a weight, the related semir-
ing does not satisfy, in general, the necessary properties for applying such opera-
tions. In particular, the path property, which is necessary for almost all FST oper-
ations, does not apply. This can be better understood looking at the RTN at the
top of Figure 8 labelled “A” which shows the application of the hiero rule R1 in
Figure 4, and where w1 represents the tropical weight and the POS and dependency
tree information3.

\[
\begin{aligned}
\text{RTN: } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
& a & X1 & f & X2 \\
\text{X1: } & 0 & 1 & 2 & 3 \\
& b & c & & & \\
& \varepsilon/W2 & & & \\
& \varepsilon/W3 \\
\text{Exp. RTN: } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\
& a & b & c & f & & & & \\
& \varepsilon/W1 & & & \\
& \varepsilon/W2 & & & \\
& \varepsilon/W3 \\
\end{aligned}
\]

Figure 8: FST example with dependency weights where Determinization fails.

In the example in the figure, the RTN first non-terminal X1 points to the FST
labelled “X1”, which represents two possible dependency representations (weights
w2 and w3) for string “b c”. The second non-terminal X2 is pointing to some other
FST not shown.

Expanding the above RTN produces the FST labelled “Exp. RTN” in the figure.
Although not evident in the above simple example, the expanded RTN can contain
redundant information in terms of duplicated sub-paths and nodes. Therefore a
common procedure is to determinize (no state is allowed two transitions with the
same input label) and minimize it to optimise its size. However, the FST labelled
“Exp. RTN” cannot be determinized because it is not possible to establish which
arc between those with weights w2 and w3 has the higher cost.

3Note that, as it will be clearer in the following, for dependency hiero or glue rules the weight need to be placed
on an arc preceding the first non-terminal. The first arc is used in this work. Though for phrase rules the weight
can be placed on any arc, the last arc is used here to distinguish them from hiero and glue rules.
In fact, while this would be possible if only tropical weights are involved, for the dependency and POS weights the results cannot be known until the results of $W1 \otimes W2$ and $W1 \otimes W3$ are compared.

The result of the above “times” ($\otimes$) operations between weights is twofold: penalties are added in the case of POS mismatches, and different dependency relationships are generated for the two paths, which will provide different dependency LM scores\(^4\). Both information are necessary to establish which path is the best scoring one.

An approach where dependency information and related operations are coded into a string tag for delayed processing as in [2] could also be possible. However, the authors believe that this approach would have several drawbacks:

- very complicated tags representing nested complex operations will be generated
- FSTs representing partial hypotheses cannot be optimised unless these tags are resolved to compute penalties and dependency LM scores

The second issue above occurs when for a given sentence span, all possible glue rules (combinatory operations) and hiero rules are applied. The result is likely to provide some paths which result in identical target symbols, final POS tags and dependency vectors, but with different weights because of the penalties introduced. In such case only the path with smallest cost need be retained. This is only possible if dependency information processing is not delayed. A similar problem arises if local pruning needs to be applied.

For the above reasons, to overcome the limitations discussed earlier, a “controlled FST topology” in combination with “dependency weights encoding” are used. These are described in next section.

### 4.2 “Controlled FST topology” and dependency weights encoding

As discussed in the previous section, implementing the string-to-dependency algorithm using FST operations poses two contrasting requirements: weights are needed so that combinatory operations can be carried out by the “times” operation; and paths with same target symbols but different dependency structures should be preserved when operations to optimise the FST size are applied. A “controlled FST topology” is used in combination with the weight pushing operation to satisfy the first requirement, while to meet the second requirement weights are encoded into labels.

#### 4.2.1 Weight pushing on “Controlled FST topology”

The term “Controlled FST topology” is used here to refer to an FST as the expanded RTN in Figure 8 where each path has a hiero or glue rule weight on the first epsilon

\(^4\)Note that even if a look-ahead version of the Determination function could be implemented, resolving more complicated networks might not be trivial.
arc, followed by one or two weights (one for each non-terminal) of the type described in the phrase rule section. Before this FST can be pointed to by another hiero or glue rule which will provide the partial hypotheses for a higher sentence span, it needs to be transformed into an FST where each path is a sequence of target symbols with a final “phrase-like” (dependency) weight, as the FST labelled “X1” in Figure 8. The required operations to obtain such transformation are illustrated in the following starting from the expanded RTN in Figure 9, which represents a hiero rule with a single non-terminal pointing to three different phrases.

![Figure 9: Hiero rule application with one non-terminal.](image)

The format of each arc label in the above FST is

\[
\text{output\_symbol/tropical\_weight,dependency\_weight}
\]

where the string deps is used to separate the (coded) POS information (values from table in Appendix B) from the dependency vector (\(G, A, L,\) and \(F\)).

The hiero rule in the FST above has target side \(<a X>\), with POS 12 and expecting the pointed structure to have POS 24. Vector \(G\) specifies the word dependencies of the target side. This rule is concatenated to three alternative phrases, \(<b c>\), \(<b>\), and \(<c>\). Each phrase is described by a \{POS, dependency vector\} pair which is placed on a final epsilon arc (\{18, A\}, \{24, L\}, and \{2, F\}).

The weight pushing (to the right) operation can be applied to this network. This operation first computes the shortest distance from the start node to each node of the network, then uses the result to recompute each arc weight. The result is shown in Figure 10. A POS mismatch occurred between the expected POS (24) and the POSs (18 and 2) of arcs between nodes \{4,5\} and \{7,5\}. A penalty was therefore applied to these arcs (a value of 50 is used here). As a result of the penalty application the best path (considering the tropical weight only) provides string \(ab\) with POS 12 and final dependency vector given by \(G\times L\), which indicates that the “times” operation\(^5\) was applied between vectors \(G\) and \(L\). This operation is defined so that the non-terminal node in the dependency tree specified by \(G\) is substituted with the tree specified by \(L\) (similarly to the example of Figure 5).

The weight pushing operation applies division (\(\oslash\)) between each arc weight (all tropical, POS and dependency weights) and the shortest distance value previously computed for the node pointed to by the arc.

A \texttt{div} label is used by the library to separate the dividend from the divisor.

\(^5\)Note x stands for symbol \(\otimes\) here, and \(\).
Figure 10: Push weights to the right applied to network of Figure 9.

After division is applied the general form of the weight is given by

\[
\text{POS1} \div \text{POS2} \_\text{deps} \_A \div \_B
\]

which specifies that POS1 is divided by POS2 and the dependency vector A is divided by vector B. Differently that the times operation, whose definition is simple and whose result is obtained at the time the operation is applied, division is a more complex operation whose result cannot always be computed immediately. If the dividend and the divisor are identical they cancel each other providing an empty label “\(< >\)”, as it was the case for arc between nodes 6 and 5. Otherwise the following is applied.

### 4.2.2 Division between POSs

When \( \text{POS1} \div \text{POS2} \) has to be computed, the “controlled FST topology” guarantees that there will always be a following arc (not considering arcs with empty POS tags) with a POS2 cancelling out the POS2 divisor\(^6\).

For hiero rules, as in the above example, the POS division always gives a null label since the initial hiero rule POS (12 in this case) will also be the final POS for each path. When a glue rule is applied instead, and weight pushing applied as shown in Figure 19, POS division occurs as, for example, in the first arc between nodes 11 and 16 (18 \( \div \) 24). In this case the POS divisor (24) will be subsequently cancelled out by the POS value in the final node.

### 4.2.3 Division between dependency vectors

Division between dependency vectors follows a similar process. In the case of a hiero rule with a single non-terminal, as in the example above, the divisor is always cancelled out by the following dependency vector. An example is given by the arc between nodes 4 and 5 (GxAA \( \div \) GxL) and the final node 5 (Gxl). Applying the \( \otimes \) operation between these two weights cancels out the divisor GxL providing GxAA. The more complicated case with two non-terminals is discussed in Section 5.1.

\(^6\)An error is thrown by the library otherwise.
4.2.4 Composition with a rho matcher

Once the FST in Figure 10 is obtained, the next step requires to modify the network topology so that each path from start to end node has a unique sequence of dependency weights\(^7\). Although an ad-hoc operation can be implemented to perform this cloning procedure, the same result is easily obtained applying the OpenFST standard composition between the above FST and a “rho matcher”. For this composition to work, dependency weights need to be coded into labels first (weights encoding and rho-matcher design are discussed later). After weights encoding, rho-composition, and decoding, the resulting FST is shown in Figure 11.

Figure 11: Composition of FST of Figure 10 with a “rho matcher”.

It is now possible to apply to the above FST the weight pushing operation which will provide the final POS and dependency structure for each path. The result is shown in Figure 12.

Figure 12: Push weights operation applied to FST of Figure 11.

During decoding a set of hiero/glue rules need to be evaluated for each given sentence span. For each of these rules the approach described above is applied and all the resulting FSTs (as that of Figure 12) can be joined through the union operation. The resulting FST would be similar in topology to that in the square box of Figure 9 (a single dependency weight at the end of each possible path) and can therefore be used as a source set of phrases pointed to by the non-terminal of another rule in a higher CYK cell. The process is repeated until the whole sentence is translated.

\(^7\)Different symbol sequences can share the same dependency weights combination, as shown in Figure 20.
The above simplified example was used to show the basic steps of the proposed approach. But, in a more realistic case, more complicated networks are produced after each single rule application. Optimisation of such intermediate results is therefore necessary to avoid generating too large lattices. This can be done after dependency weights encoding is applied, as it is described in next section.

4.2.5 Dependency weights encoding and FST optimisation

The FSTs produced by the process described in the previous section can easily become very large. The effect of the rho-composition is to duplicate paths adding a lot of redundant information. This is particularly true when rules with two non-terminals are applied, resulting in a number of possible independent paths equal to the product of the number of independent paths in the first FST by that of the second FST, pointed to by the first and the second non-terminal, respectively.

Considering all the rules that apply at each CYK cell and that the process is iterated until the whole sentence is decoded, this quickly become unfeasible. Determinization and Minimization are common operations applied to FSTs to optimise their size. However, as discussed in Section 4.1, it is not possible to apply such operations on FSTs with POS and dependency weights. A workaround for this issue is to encode the dependency weights into labels as is shown in Figure 13 where the FST from Figure 12 was encoded, generating the input/output (dependency) labels \textit{TAG9}, \textit{TAG11}, \textit{TAG12}.

![Figure 13: Dependency weights encoding of FST in Figure 12.](image)

The procedure is similar to the standard weights encoding operation provided by the OpenFST library, with the difference that only POS and dependency (not tropical) weights are encoded, and that the arc (or node) to be encoded must not have input/output labels. The first requirement is needed so that when FST optimisation operations are applied the (tropical) cost of each path is available and identical hypotheses can be removed\(^8\). Though the second requirement is not fundamental for the approach to work, if not met will make the optimisation less efficient\(^9\). These two requirements are always satisfied by the procedure described

\(^8\)If two hypotheses are identical after POS and dependency encoding it implies that they also have identical final dependency tag. Therefore discarding all but the lowest (tropical) cost one can be done without losing any information.

\(^9\)If the arc to be encoded has non null input/output labels, these would have to be encoded together with the POS and dependency weights into the final tag. If this happens, arcs with identical original final input/output labels will not be merged if the respective POS and dependency information is different. Or, the encoded tags will be different for arcs with identical POS and dependency weights but different final input/output labels, preventing the arcs with identical dependency information to be merged.
in the previous section.

After applying the Remove epsilon, Minimisation and Determinization operations to the FST in Figure 13 the FST of Figure 14 is obtained. Encoded weights need to be decoded before this FST is involved into another weight pushing operation.

![Figure 14: Remove epsilon, Minimisation and Determinization operations applied to FST of Figure 13.](image)

Encoding, FST optimisation, and decoding can be applied at any step of the process described in the previous section. This allows to reduce the size of the lattices exploiting paths which share labels/dependency information.

Local pruning is also possible. Usually the LM is applied prior to local pruning to retain only a given set of best scoring hypotheses. Applying a standard LM in this case is straightforward, through composition of an encoded FST with a LM FST. The only requirement is that the dependency tags are ignored by the LM (this is easily obtained in HiFST employing a multi-epsilon matcher which “Treats specified non-0 labels as non-consuming labels”). If the dependency LM is also applied for local pruning, than each sequence of target labels along with its dependency information need to be retrieved and scored. A possible efficient implementation would store this intermediate dependency LM scores into the dependency weight (along with neighbouring information, [3]), so that the dependency LM score for the current partial hypothesis does not need to be re-computed at each further step of the decoding process.

As mentioned in the previous section, a useful side effect of applying this special weight encoding operation is that we can use a “rho matcher” to duplicate nodes prior to applying the weight pushing operation. The “rho matcher” used to obtain the FST in Figure 11 is shown in Figure 15.

The “rho” label in the matcher will match any arc with no dependency tags and a path for each possible sequence of encoded tags in the original FST must exist\(^{10}\).

\(^{10}\)A double nested loop through all possible existing TAGS was used to obtain this matcher which has several paths that are not actually needed. A much simpler version is built by the extended library looking at the position of the encoded tags in the original FST.
5 Example of hiero rule application

In this section an example of LA between fixed structures is shown. The glue rule used is as R4 in Figure 6 with the difference that the POS tag is now defined as \( \text{POSs} = \text{HEAD}\text{?HEAD}\text{?HEAD} \), that is, the final structure will be a fixed structure (HEAD), while each of the structures pointed to by the non-terminals need to be fixed (\( \text{?HEAD}\text{?HEAD} \)).

In this example two FSTs with different hypotheses and final POS and dependencies are pointed to by the non-terminals in the RTN representing the glue rule. The first FST is shown in Figure 16 which, after coding and optimisation, is transformed to the FST of Figure 17 (a similar FST is obtained for the second non-terminal).

Figure 16: FST pointed to by the first non-terminal of the glue rule.

Figure 15: “rho matcher” used to obtain FST in Figure 11.
Expanding the RTN gives the FST shown in Figure 18, where the first arc has the following POS information

1004?4?4

which represents HEAD?HEAD?HEAD coded using the table in Appendix B with the additional convention that the final POS value is added 1000 to distinguish this rule from hiero rules.

Now it is possible to apply the weight pushing operation to the FST of Figure 18, which yield the FST of Figure 19. Note that after this operation all the original paths which had a non-HEAD POS will have infinite (Inf) weights. This is because, for glue rules, a POS mismatch (a HEAD structure is expected in this example by both non-terminals) results in an infinite weight so that that particular path will be removed.

5.1 Division between dependency vectors: two non-terminals case

This glue rule example also shows how actual division between dependency vectors is obtained (see discussion in Section 4.2.1). The best path for each node computed during the weight pushing operation up to node 7 on the FST in Figure 18 is \{0, 1, 2, 3, 4, 6, 7\} with a partial dependency tree given by GxA. This will be the divisor for all the partial results obtained up to node 7 (Figure 19). Similarly, the best path up to the final node in Figure 18 is given by nodes \{0, 1, 2, 3, 4, 6, 7, 8, 10, 13, 16\} with a final dependency vector GxAxL. All the dependency vectors obtained through the different paths up to the final node are then divided by GxAxL (Figure 19).

After coding, rho-composition and decoding is applied, the FST of Figure 20 is obtained. Now it is possible to apply weight pushing to this FST to obtain a final POS and dependency values for each path. Considering for example the path through nodes \{0, 1, 2, 3, 4, 7, 10, 13, 19, 25, 37\} the following needs to be computed.

\{ \{GxB\} \ {GxA}\} \times \{ \{GxAxB\} \ {GxAxL}\} \}

Note that the GxB, GxA, GxAxB and GxAxL terms are expressed here in terms of their factors for clarity. However, the value actually stored as the arc weight is the resulting (unique) dependency vector. To compute the above operation the OpenFST library extension first computes the following special division operation

\{ \ \{GxA\}\} \times \{GxAxB\} = B
which, substituted in the expression above gives

\[ \{ GxB \} \setminus \{ GxA \} \times \{ GxAxB \} \setminus \{ GxAxL \} = \{ GxBxB \} \setminus \{ GxAxL \} \]

The last divisor \( GxAxL \) will then be cancelled out by the dependency weight in node 16, providing \( GxBxB \) as the final dependency tree for the path considered. The “controlled FST topology” guarantees that the operations above can always be resolved. The result of this second weight pushing application is showed in Figure 21.

After weights encoding and optimisation the final FST in Figure 22 is obtained.
Figure 18: Expanded RTN representing glue rule application for left adjoining (LA) of two fixed structures.
Figure 19: Push weights applied to the FST of Figure 18.
Figure 20: FST obtained after encoding, rho-composition, and decoding of FST in Figure 19.
Figure 21: Weight pushing applied to the FST of Figure 20.
Figure 22: FST obtained after coding and optimisation of FST in Figure 21.
A List of glue rules

Glue rules for implementing left/right adjoin/concatenation operations are listed below. Their format is as follows

\[ X_1 \_X_2 \quad X_1 \_X_2 \quad W \quad \text{POSs} \quad \text{DEPs} \]

where both source and target are just two non-terminals, \( X_1 \_X_2 \), and \( \text{POSs} \) is of the form \( \text{POS}1?\text{POS}2?\text{POS}3 \) as explained in Section 2.
<table>
<thead>
<tr>
<th>POSs</th>
<th>DEPs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left Adjoining:</strong></td>
<td></td>
</tr>
<tr>
<td>HEAD?LEFT?HEAD</td>
<td>1, -1</td>
</tr>
<tr>
<td>HEAD?HEAD?HEAD</td>
<td>1, -1</td>
</tr>
<tr>
<td>HEAD?NULL?HEAD</td>
<td>1, -1</td>
</tr>
<tr>
<td>NULL?RIGHT?HEAD</td>
<td>1, -1</td>
</tr>
<tr>
<td><strong>Left Concatenate:</strong></td>
<td></td>
</tr>
<tr>
<td>LEFT?LEFT?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>LEFT?LEFT?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?LEFT?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>LEFT?HEAD?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>LEFT?HEAD?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?HEAD?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?RIGHT?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?RIGHT?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>LEFT?LEFT?NULL</td>
<td>-1, -1</td>
</tr>
<tr>
<td>LEFT?NULL?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td><strong>Right Adjoining:</strong></td>
<td></td>
</tr>
<tr>
<td>HEAD?HEAD?HEAD</td>
<td>-1, 0</td>
</tr>
<tr>
<td>HEAD?HEAD?RIGHT</td>
<td>-1, 0</td>
</tr>
<tr>
<td>HEAD?HEAD?NULL</td>
<td>-1, 0</td>
</tr>
<tr>
<td>NULL?HEAD?LEFT</td>
<td>-1, 0</td>
</tr>
<tr>
<td><strong>Right Concatenate:</strong></td>
<td></td>
</tr>
<tr>
<td>NULL?LEFT?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?LEFT?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?HEAD?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?HEAD?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?HEAD?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>NULL?RIGHT?LEFT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?RIGHT?HEAD</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?RIGHT?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?NULL?RIGHT</td>
<td>-1, -1</td>
</tr>
<tr>
<td>RIGHT?RIGHT?NULL</td>
<td>-1, -1</td>
</tr>
</tbody>
</table>
B POS mapping

The following mapping (only first 30 entries shown) is used for the POS values. Note that actual POSs start from index 10 while indexes from 1 to 4 are used to indicate a LEFT, RIGHT, NULL and HEAD structures, respectively. When POS matching is checked, a HEAD POS will match any of the actual POS values from index 10 above.

1 LEFT
2 RIGHT
3 NULL
4 HEAD
10 ,
11 :
12 .
13 ( 
14 )
15 $ 
16 CC
17 CD
18 DT
19 EX
20 FW
21 IN
22 IN/that
23 JJ
24 JJR
25 JJS
26 LS
27 MD
28 NN
29 NNP
30 NNPS
...

References
