DAGGER: A Toolkit for Automata on Directed Acyclic Graphs

Daniel Quernheim
Institute for Natural Language Processing
Universität Stuttgart, Germany
Pfaffenwaldring 5b, 70569 Stuttgart
daniel@ims.uni-stuttgart.de

Kevin Knight
University of Southern California
Information Sciences Institute
Marina del Rey, California 90292
knight@isi.edu

Abstract

This paper presents DAGGER, a toolkit for finite-state automata that operate on *directed acyclic graphs* (dags). The work is based on a model introduced by (Kamimura and Slutzki, 1981; Kamimura and Slutzki, 1982), with a few changes to make the automata more applicable to natural language processing. Available algorithms include membership checking in bottom-up dag acceptors, transduction of dags to trees (bottom-up dag-to-tree transducers), k-best generation and basic operations such as union and intersection.

1 Introduction

Finite string automata and finite tree automata have proved to be useful tools in various areas of natural language processing (Knight and May, 2009). However, some applications, especially in semantics, require graph structures, in particular *directed acyclic graphs* (dags), to model reentrancies. For instance, the dags in Fig. 1 represents the semantics of the sentences "The boy wants to believe the girl" and "The boy wants the girl to believe him." The double role of "the boy" is made clear by the two parent edges of the BOY node, making this structure non-tree-like.

Powerful graph rewriting systems have been used for NLP (Bohnet and Wanner, 2010), yet we consider a rather simple model: finite dag automata that have been introduced by (Kamimura and Slutzki, 1981; Kamimura and Slutzki, 1982) as a straightforward extension of tree automata. We present the toolkit DAGGER (written in PYTHON) that can be used to visualize dags and to build dag acceptors and dag-to-tree transducers similar to their model. Compared to those devices, in order to use them for actual NLP tasks, our machines differ in certain aspects:

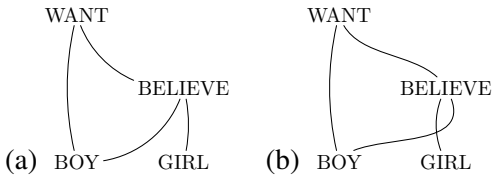

Figure 1: (a) "The boy wants to believe the girl." and (b) "The boy wants the girl to believe him." First edge represents `:agent` role, second edge represents `:patient` role.

- We do not require our dags to be planar, and we do not only consider derivation dags.
- We add weights from any commutative semiring, e.g. real numbers.

The toolkit is available under an open source licence.[1]

2 Dags and dag acceptors

DAGGER comes with a variety of example dags and automata. Let us briefly illustrate some of them. The dag of Fig. 1(a) can be defined in a human-readable format called PENMAN (Bateman, 1990):

```
(1 / WANT
    :agent (2 / BOY)
    :patient (3 / BELIEVE
                  :agent 2
                  :patient (4 / GIRL)))
```

[1] http://www.ims.uni-stuttgart.de/~daniel/dagger/

```
s
s -> (WANT :agent i :patient s)
s -> (BELIEVE :agent i :patient s)
i -> (0)
s -> (0)
i -> (GIRL)
i -> (BOY)
s -> (GIRL)
s -> (BOY)
i i -> (GIRL)
i i -> (BOY)
i s -> (GIRL)
i s -> (BOY)
s s -> (GIRL)
s s -> (BOY)
```

Figure 2: Example dag acceptor `example.bda`.

In this format, every node has a unique identifier, and edge labels start with a colon. The tail node of an edge is specified as a whole subdag, or, in the case of a reentrancy, is referred to with its identifier.

Fig. 2 shows a dag acceptor. The first line contains the final state, and the remaining lines contain rules. Mind that the rules are written in a top-down fashion, but are evaluated bottom-up for now. Let us consider a single rule:

```
s -> (WANT :agent i :patient s)
```

The right-hand side is a symbol (`WANT :agent :patient`) whose tail edges are labeled with states (`i` and `s`), and after applying the rule, its head edges are labeled with new states (`s`). All rules are height one, but in the future we will allow for larger subgraphs.

In order to deal with symbols of arbitrary head rank (i.e. symbols that can play multiple roles), we can use rules using special symbols such as `2=1` and `3=1` that split one edge into more than one:

```
i s -> (2=1 :arg e)
```

Using these state-changing rules, the ruleset can be simplified (see Fig. 3), however the dags look a bit different now:

```
(1 / WANT
    :agent (2 / 2=1
              :arg (3 / BOY))
    :patient (4 / BELIEVE
                :agent 2
                :patient (5 / GIRL)))
```

Note that we also added weights to the ruleset now. Weights are separated from the rest of a rule by the `@` sign. The weight semantics is the usual one, where weights are multiplied along derivation steps, while the weights of alternative derivations are added.

```
s
s -> (WANT :agent i :patient s) @ 0.6
s -> (BELIEVE :agent i :patient s) @ 0.4
i -> (0) @ 0.2
s -> (0) @ 0.4
i -> (GIRL) @ 0.3
s -> (GIRL) @ 0.3
i -> (BOY) @ 0.2
s -> (BOY) @ 0.2
i i -> (2=1 :arg e) @ 0.3
i s -> (2=1 :arg e) @ 0.3
s s -> (2=1 :arg e) @ 0.3
e -> (GIRL) @ 0.4
e -> (BOY) @ 0.6
```

Figure 3: Simplified dag acceptor `simple.bda`.

2.1 Membership checking and derivation forests

DAGGER is able to perform various operations on dags. The instructions can be given in a simple expression language. The general format of an expression is:

(command f_1 .. f_m p_1 .. p_n)

Every command has a number of (optional) features f_i and a fixed number of arguments p_i. Most commands have a short and a long name; we will use the short names here to save space. In order to evaluate a expression, you can either

- supply it on the command-line:

  ```
  ./dagger.py -e EXPRESSION
  ```

- or read from a file:

  ```
  ./dagger.py -f FILE
  ```

We will now show a couple of example expressions that are composed of smaller expressions. Assume that the dag acceptor of Fig. 2 is saved in the file `example.bda`, and the file `boywants.dag` contains the example dag in PENMAN format. We can load the dag with the expression `(g (f boywants.dag))`, and the acceptor with the expression `(a w (f example.bda))` where `w` means that the acceptor is weighted. We could also specify the dag directly in PENMAN format using `p` instead of `f`. We can use the command `r`:

```
(r (a w (f example.bda)) (g (f
boywants.dag)))
```

to check whether `example.bda` recognizes `boywants.dag`. This will output one list item

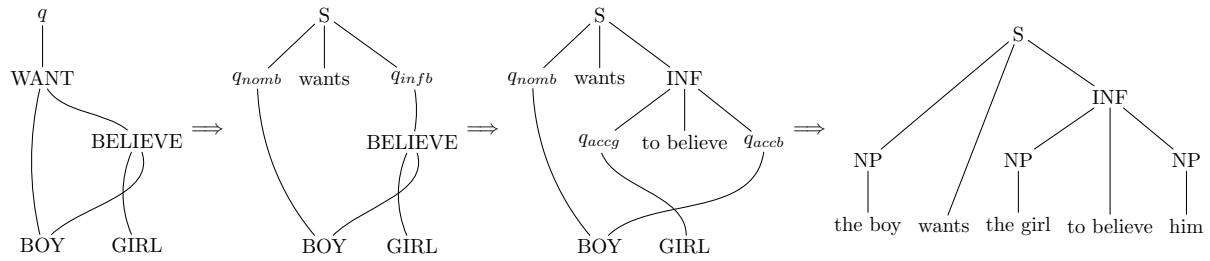

Figure 4: Derivation from graph to tree "the boy wants the girl to believe him".

```
q
q.S(x1 wants x2)) -> (WANT :agent nomb.x1 :patient inf.x2)
inf.INF(x1 to believe x2) -> (BELIEVE :agent accg.x1 :patient accb.x2)
accg.NP(the girl) -> (GIRL)
nomb.NP(the boy) accb.(him) -> (BOY)
```

Figure 5: Example dag-to-tree-transducer `example.bdt`.

for each successful derivation (and, if the acceptor is weighted, their weights), in this case: `('s', '0.1', 0, '0')`, which means that the acceptor can reach state `s` with a derivation weighted 0.1. The rest of the output concerns dag-to-tree transducers and will be explained later.

Note that in general, there might be multiple derivations due to ambiguity (non-determinism). Fortunately, the whole set of derivations can be efficiently represented as another dag acceptor with the `d` command. This *derivation forest acceptor* has the set of rules as its symbol and the set of configurations (state-labelings of the input dag) as its state set.

```
(d (a w (f example.bda)) (g f
boywants.dag)))
```

will write the derivation forest acceptor to the standard output.

2.2 k-best generation

To obtain the highest-weighted 7 dags generated by the example dag acceptor, run:

```
(k 7 (a w (f example.bda)))

(1 / BOY)
(1 / GIRL)
(1 / BELIEVE :agent (2 / GIRL) :patient 2)
(1 / WANT :agent (2 / GIRL) :patient 2)
(1 / 0)
(1 / BELIEVE :agent (2 / BOY) :patient 2)
(1 / WANT :agent (2 / BOY) :patient 2)
```

If the acceptor is unweighted, the smallest dags (in terms of derivation steps) are returned.

```
(1 / 0)
(1 / BOY)
(1 / GIRL)
(1 / BELIEVE :agent (2 / GIRL) :patient 2)
(1 / BELIEVE :agent (2 / BOY) :patient 2)
(1 / BELIEVE :agent (2 / GIRL) :patient
    (3 / 0))
(1 / BELIEVE :agent (2 / GIRL) :patient
    (3 / GIRL))
```

2.3 Visualization of dags

Both dags and dag acceptors can be visualized using GRAPHVIZ[2]. For this purpose, we use the `q` (query) command and the `v` feature:

```
(v (g (f boywants.dag)) boywants.pdf)
(v (a (f example.bda)) example.pdf)
```

Dag acceptors are represented as hypergraphs, where the nodes are the states and each hyperedge represents a rule labeled with a symbol.

2.4 Union and intersection

In order to construct complex acceptors from simpler building blocks, it is helpful to make use of union (`u`) and intersection (`i`). The following code will intersect two acceptors and return the 5 best dags of the intersection acceptor.

```
(k 5 (i (a (f example.bda)) (a (f
someother.bda))))
```

Weighted union, as usual, corresponds to sum, weighted intersection to product.

[2] available under the Eclipse Public Licence from `http://www.graphviz.org/`

	string automata	tree automata	dag automata
compute	... strings (sentences)	... (syntax) trees	... semantic representations
k-best	... paths through a WFSA (Viterbi, 1967; Eppstein, 1998)	... derivations in a weighted forest (Jiménez and Marzal, 2000; Huang and Chiang, 2005)	✓
EM training	Forward-backward EM (Baum et al., 1970; Eisner, 2003)	Tree transducer EM training (Graehl et al., 2008)	?
Determinization	... of weighted string acceptors (Mohri, 1997)	... of weighted tree acceptors (Borchardt and Vogler, 2003; May and Knight, 2006a)	?
Transducer composition	WFST composition (Pereira and Riley, 1997)	Many transducers not closed under composition (Maletti et al., 2009)	?
General tools	AT&T FSM (Mohri et al., 2000), Carmel (Graehl, 1997), OpenFST (Riley et al., 2009)	Tiburon (May and Knight, 2006b), ForestFIRE (Cleophas, 2008; Strolenberg, 2007)	DAGGER

Table 1: General-purpose algorithms for strings, trees and feature structures.

3 Dag-to-tree transducers

Dag-to-tree transducers are dag acceptors with tree output. In every rule, the states on the right-hand sides have tree variables attached that are used to build one tree for each state on the left-hand side. A fragment of an example dag-to-tree transducer can be seen in Fig. 5.

Let us see what happens if we apply this transducer to our example dag:

```
(r (a t (f example.bdt)) (q (f
boywants.dag)))
```

All derivations including output trees will be listed:

```
('q', '1.0',
S(NP(the boy) wants INF(NP(the girl)
to believe NP(him))),
'the boy wants the girl to believe
him')
```

A graphical representation of this derivation (top-down instead of bottom-up for illustrative purposes) can be seen in Fig. 4.

3.1 Backward application and force decoding

Sometimes, we might want to see which dags map to a certain input tree in a dag-to-tree transducer. This is called *backward application* since we use the transducer in the reverse direction: We are currently implementing this by "generation and checking", i.e. a process that generates dags and trees at the same time. Whenever a partial tree does not match the input tree, it is discarded, until we find a derivation and a dag for the input tree. If we also restrict the dag part, we have *force decoding*.

4 Future work

This work describes the basics of a dag automata toolkit. To the authors' knowledge, no such implementation already exists. Of course, many algorithms are missing, and there is a lot of room for improvement, both from the theoretical and the practical viewpoint. This is a brief list of items for future research (Quernheim and Knight, 2012):

- Complexity analysis of the algorithms.
- Closure properties of dag acceptors and dag-to-tree transducers as well as composition with tree transducers.
- Extended left-hand sides to condition on a larger semantic context, just like extended top-down tree transducers (Maletti et al., 2009).
- Handling flat, unordered, sparse sets of relations that are typical of feature structures. Currently, rules are specific to the rank of the nodes. A first step in this direction could be gone by getting rid of the explicit `n=m` symbols.
- Hand-annotated resources such as (dag, tree) pairs, similar to treebanks for syntactic representations as well as a reasonable probabilistic model and training procedures.
- Useful algorithms for NLP applications that exist for string and tree automata (cf. Table 1). The long-term goal could be to build a semantics-based machine translation pipeline.

Acknowledgements

This research was supported in part by ARO grant W911NF-10-1-0533. The first author was supported by the German Research Foundation (DFG) grant MA 4959/1–1.

References

John A. Bateman. 1990. Upper modeling: organizing knowledge for natural language processing. In *Proc. Natural Language Generation Workshop*, pages 54–60.

L. E. Baum, T. Petrie, G. Soules, and N. Weiss. 1970. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Ann. Math. Statist.*, 41(1):164171.

Bernd Bohnet and Leo Wanner. 2010. Open source graph transducer interpreter and grammar development environment. In *Proc. LREC*.

Björn Borchardt and Heiko Vogler. 2003. Determinization of finite state weighted tree automata. *J. Autom. Lang. Comb.*, 8(3):417–463.

Loek G. W. A. Cleophas. 2008. *Tree Algorithms: Two Taxonomies and a Toolkit*. Ph.D. thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology.

Jason Eisner. 2003. Learning non-isomorphic tree mappings for machine translation. In *Proc. ACL*, pages 205–208.

David Eppstein. 1998. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673.

Jonathan Graehl, Kevin Knight, and Jonathan May. 2008. Training tree transducers. *Comput. Linguist.*, 34(3):391–427.

Jonathan Graehl. 1997. Carmel finite-state toolkit. `http://www.isi.edu/licensed-sw/carmel`.

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proc. IWPT*.

Víctor M. Jiménez and Andrés Marzal. 2000. Computation of the n best parse trees for weighted and stochastic context-free grammars. In *Proc. SSPR/SPR*, pages 183–192.

Tsutomu Kamimura and Giora Slutzki. 1981. Parallel and two-way automata on directed ordered acyclic graphs. *Inf. Control*, 49(1):10–51.

Tsutomu Kamimura and Giora Slutzki. 1982. Transductions of dags and trees. *Math. Syst. Theory*, 15(3):225–249.

Kevin Knight and Jonathan May. 2009. Applications of weighted automata in natural language processing. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*. Springer.

Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. 2009. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39(2):410–430.

Jonathan May and Kevin Knight. 2006a. A better n-best list: Practical determinization of weighted finite tree automata. In *Proc. HLT-NAACL*.

Jonathan May and Kevin Knight. 2006b. Tiburon: A weighted tree automata toolkit. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *Proc. CIAA*, volume 4094 of *LNCS*, pages 102–113. Springer.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 2000. The design principles of a weighted finite-state transducer library. *Theor. Comput. Sci.*, 231(1):17–32.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Fernando Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, pages 431–453. MIT Press.

Daniel Quernheim and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proc. SSST*. (to appear).

Michael Riley, Cyril Allauzen, and Martin Jansche. 2009. OpenFST: An open-source, weighted finite-state transducer library and its applications to speech and language. In *Proc. HLT-NAACL (Tutorial Abstracts)*, pages 9–10.

Roger Strolenberg. 2007. ForestFIRE and FIREWood. a toolkit & GUI for tree algorithms. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology.

Andrew Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.