hyp: A Toolkit for Representing, Manipulating, and Optimizing Hypergraphs

Markus Dreyer*
SDL Research
6060 Center Drive Suite 150
Los Angeles, CA 90045
markus.dreyer@gmail.com

Jonathan Graehl
SDL Research
6060 Center Drive Suite 150
Los Angeles, CA 90045
graehl@sdl.com

Abstract

We present hyp, an open-source toolkit for the representation, manipulation, and optimization of weighted directed hypergraphs. hyp provides compose, project, invert functionality, k-best path algorithms, the inside and outside algorithms, and more. Finite-state machines are modeled as a special case of directed hypergraphs. hyp consists of a C++ API, as well as a command line tool, and is available for download at github.com/sdl-research/hyp.

1 Introduction

We present hyp, an open-source toolkit that provides data structures and algorithms to process weighted directed hypergraphs.

Such hypergraphs are important in natural language processing and machine learning, e.g., in parsing (Klein and Manning (2005), Huang and Chiang (2005)), machine translation (Kumar et al., 2009), as well as in logic (Gallo et al., 1993) and weighted logic programming (Eisner and Filardo, 2011).

The hyp toolkit enables representing and manipulating weighted directed hypergraphs, providing compose, project, invert functionality, k-best path algorithms, the inside and outside algorithms, and more. hyp also implements a framework for estimating hypergraph feature weights by optimization on forests derived from training data.

*Markus Dreyer is now at Amazon, Inc., Seattle, WA.

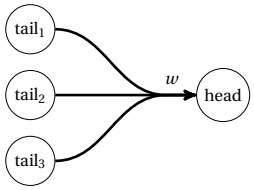

Figure 1: An arc leading from three tail states to a head state, with weight w.

2 Definitions

A weighted directed hypergraph (hereinafter *hypergraph*) is a pair $H = \langle V, E \rangle$, where V is a set of vertices and E a set of edges. Each edge (also called *hyperedge*) is a triple $e = \langle T(e), h(e), w(e) \rangle$, where $T(e)$ is an ordered list of tails (i.e., source vertices), $h(e)$ is the head (i.e., target vertex) and $w(e)$ is the semiring weight (see Section 3.4) of the edge (see Figure 1).

We regard hypergraphs as *automata* and call the vertices *states* and edges *arcs*. We add an optional start state $S \in V$ and a final state $F \in V$.

Each state s has an input label $i(s) \in (\Sigma \cup \{\varnothing\})$ and output label $o(s) \in (\Sigma \cup \{\varnothing\})$; if $o(s) = \varnothing$ then we treat the state as having $o(s) = i(s)$. The label alphabet Σ is divided into disjoint sets of nonterminal, lexical, and special $\{\epsilon, \phi, \rho, \sigma\}$ labels. The input and output labels are analogous to those of a finite-state transducer in some hyp operations (Section 3.3).

The set of incoming arcs into a state s is called the *Backward Star* of s, or short, $BS(s)$. Formally, $BS(s) = \{a \in E : h(a) = s\}$. A *path* π is a sequence of arcs $\pi = (a_1 \ldots a_k) \in E^*$ such that $\forall a \in \pi, \forall t \in T(a), (\exists a' \in \pi : h(a') = t) \vee BS(t) = \emptyset$. Each tail state

Proceedings of NAACL-HLT 2015, pages 11–15,
Denver, Colorado, May 31 – June 5, 2015.

t of each arc on the path must be the head of some arc on the path, unless t is the start state or has no incoming arcs and a terminal (lexical or special) input label, in which case we call t an *axiom*. The rationale is that each tail state of each arc on the path must be *derived*, by traveling an arc that leads to it, or given as an *axiom*. If the hypergraph has a start state, the first tail of the first arc of any path must be the start state. The head of the last arc must always be the final state, $h(a_k) = \mathrm{F}$. Paths correspond to trees, or proofs that the final state may be reached from axioms.

Hypergraph arcs have exactly one head; some authors permit multiple heads and would call our hypergraphs *B-hypergraphs* (Gallo et al., 1993).

3 Representing hypergraphs

Text representation. `hyp` uses a simple human-readable text format for hypergraphs. For example, see the first two lines in Figure 2. Each hypergraph arc has the following format:

```
head <- tail1 tail2 ... tailn / weight
```

Head and tail states are non-negative integers followed by an optional label in parentheses (or a pair of `(input output)` labels). If it is lexical (i.e., a word), then it is double-quoted with the usual backslash-escapes; nonterminal and special symbols are unquoted. Special symbols like ϵ, ϕ, ρ, σ are written with brackets, as `<eps>`, `<phi>`, `<rho>`, `<sigma>`. Each arc may optionally have a slash followed by a weight, which is typically a negative log probability (i.e., the cost of the arc). A final state `n` is marked as `FINAL <- n`. Figure 2 shows the text and visual representation of a hypergraph with only one arc; it represents and accepts the string *he eats rice*.

Visual representation. A provided `Draw` command can render hypergraphs using Graphviz (Gansner and North, 2000). Small gray numbers indicate the order of arc tails. Axiom nodes are filled gray.[1] The final state is drawn as a double circle, following finite-state convention.

[1] Gray is used analogously in graphical models for *observed* nodes.

```
0(S) <- 1("he") 2("eats") 3("rice") / 0.693
FINAL <- 0(S)
```

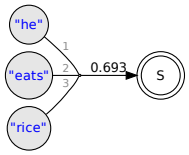

Figure 2: The text and visual representation of a hypergraph with a single arc, similar to Figure 1. The visual representation leaves out the state IDs of labeled states.

```
0(S) <- 1(NP) 2(VP)
1(NP) <- 3(PRON)
2(VP) <- 4(V) 5(NP) 6(PP)
3(PRON) <- 10("He")
4(V) <- 11("eats")
5(NP) <- 7(N)
6(PP) <- 8(PREP) 9(N)
7(N) <- 12("rice")
8(PREP) <- 13("with")
9(N) <- 14("sticks")
FINAL <- 0(S)
# These added arcs
# make it into a forest:
15(NP) <- 7(N) 6(PP)
2(VP) <- 4(V) 15(NP)
```

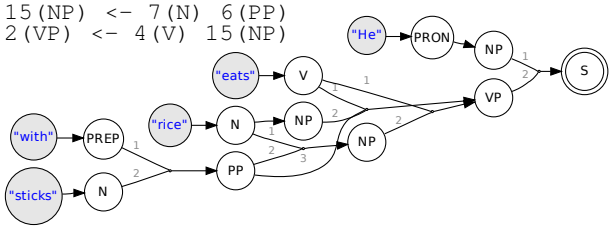

Figure 3: A packed forest.

Reducing redundancy. State labels need not be repeated at every mention of that state's ID; if a state has a label anywhere it has it always. For example, we write the label S for state 0 in Figure 2 only once:

```
0(S) <- 1("he") 2("eats") 3("rice") / 0.693
FINAL <- 0
```

Similarly, state IDs may be left out wherever a label uniquely identifies a particular state:

```
0(S) <- ("he") ("eats") ("rice") / 0.693
FINAL <- 0
```

`hyp` generates state IDs for these states automatically.

3.1 Trees and forests

A forest is a hypergraph that contains a set of trees. A forest may be *packed*, in which case its trees share substructure, like strings in a lattice. An example forest in `hyp` format is shown in Figure 3. Any two or more arcs pointing into one state have OR semantics; the depicted forest compactly rep-

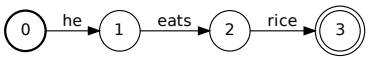

Figure 4: A one-sentence finite-state machine in OpenFst.

```
START <- 0
1 <- 0 4("he")
2 <- 1 5("eats")
3 <- 2 6("rice")
FINAL <- 3
```

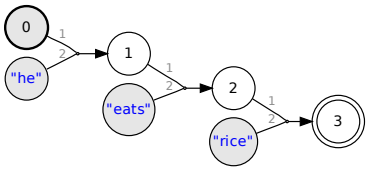

Figure 5: A one-sentence finite-state hypergraph in `hyp`.

resents two interpretations of one sentence: (1) he eats rice using sticks OR he eats rice that has sticks. Hypergraphs can represent any context-free grammar, where the strings in the grammar are the lexical yield (i.e., leaves in order) of the hypergraph trees.

3.2 Strings, lattices, and general FSMs

In addition to trees and forests, hypergraphs can represent strings, lattices, and general finite-state machines (FSMs) as a special case. A standard finite-state representation of a string would look like Figure 4, which shows a left-recursive bracketing as `(((he) eats) rice)`, i.e., we read "he", combine it with "eats", then combine the result with "rice" to accept the whole string (Allauzen et al., 2007).

We can do something similar in `hyp` using hypergraphs—see Figure 5. The hypergraph can be traversed bottom-up by first reading start state 0 and the "he" axiom state, reaching state 1, then reading the following words until finally arriving at the final state 3. The visual representation of this left-recursive hypergraph can be understood as an unusual way to draw an FSM, where each arc has an auxiliary label state. If a hypergraph has a start state and all its arcs are *finite-state arcs*, `hyp` recognizes it as an FSM; some operations may require or optimize for an FSM rather than a general hypergraph. A finite-state arc has two tails, where the first one is a structural state and the second one a terminal label state.[2] Adding additional arcs to the simple sentence hypergraph of Figure 5, we could arrive at a more interesting lattice or even an FSM with cycles and so infinitely many paths.

3.3 Transducers

A leaf state s with an output label $o(s) \neq i(s)$ rewrites the input label. This applies to finite-state as well as general hypergraphs. The following arc, for example, reads "eats" and an NP and derives a VP; it also rewrites "eats" to "ate":

```
(V) <- ("eats" "ate") (NP)
```

If a state has an output label, it must then have an input label, though it may be `<eps>`. The start state conventionally has no label.

3.4 Semirings and features

Each hypergraph uses a particular semiring, which specifies the type of weights and defines how weights are added and multiplied. `hyp` provides the standard semirings (Mohri, 2009), as well as the expectation semiring (Eisner, 2002), and a new "feature" semiring. The feature semiring pairs with tropical semiring elements a sparse feature vector that adds componentwise in the semiring product and follows the winning tropical element in the semiring sum. Features 0 and 8 fire with different strengths on this arc:

```
(V) <- 11("eats" "ate") / 3.2[0=1.3,8=-0.5]
```

By using the expectation or the feature semiring, we can keep track of what features fire on what arcs when we perform compositions or other operations. Using standard algorithms that are implemented in `hyp` (e.g., the inside-outside algorithm, see below), it is possible to train arc feature weights from data (see Section 6).

4 Using the `hyp` executable

The `hyp` toolkit provides an executable that implements several commands to process and manipulate hypergraphs. It is generally called as `hyp <command> <options> <input-files>`, where `<command>` may be `Compose`, `Best`, or others. We now describe some of these commands.

[2] Some operations may efficiently transform a generalization of FSM that we call a "graph", where there are zero or more label states following the structural or "source" state, rather than exactly one.

Compose `hyp Compose` composes two semiring-weighted hypergraphs. Composition is used to parse an input into a structure and/or rewrite its labels. Composition can also rescore a weighted hypergraph by composing with a finite-state machine, e.g., a language model.

Example call:

```
$ hyp Compose cfg.hyp fsa.hyp
```

Since context-free grammars are not closed under composition, one of the two composition arguments must be finite-state (Section 3.2). If both structures are finite-state, `hyp` uses a fast finite-state composition algorithm (Mohri, 2009).[3] Otherwise, we use a generalization of the Earley algorithm (Earley (1970), Eisner et al. (2005), Dyer (2010)).[4]

Best and PruneToBest. `hyp Best` prints the k-best entries from any hypergraph. `hyp PruneToBest` removes structure not needed for the best path.

Example calls:

```
$ hyp Best --num-best=2 h.hyp > k.txt
$ hyp PruneToBest h.hyp > best.hyp
```

For acyclic finite-state hypergraphs, `hyp` uses the Viterbi algorithm to find the best path; otherwise it uses a general best-tree algorithm for CFGs (Knuth (1977), Graehl (2005)).

Other executables. Overall, `hyp` provides more than 20 commands that perform hypergraph operations. They can be used to concatenate, invert, project, reverse, draw, sample paths, create unions, run the inside algorithm, etc. A detailed description is provided in the 25-page `hyp` tutorial document (Dreyer and Graehl, 2015).

5 Using the hyp C++ API

In addition to the command line tools described, `hyp` includes an open-source C++ API for constructing and processing hypergraphs, for maximum flexibility and performance.[5] The following code snippet creates the hypergraph shown in Figure 2:

```
typedef ViterbiWeight Weight;
typedef ArcTpl<Weight> Arc;
MutableHypergraph<Arc> hyp;
StateId s = hyp.addState(S);
hyp.setFinal(s);
hyp.addArc(new Arc(Head(s),
                   Tails(hyp.addState(he),
                         hyp.addState(eats),
                         hyp.addState(rice)),
                   Weight(0.693)));
```

The code defines weight and arc types, then constructs a hypergraph and adds the final state, then adds an arc by specifying the head, tails, and the weight. The variables `S`, `he`, `eats`, `rice` are symbol IDs obtained from a vocabulary (not shown here). The constructed hypergraph `hyp` can then be manipulated using provided C++ functions. For example, calling

```
reverse(hyp);
```

reverses all paths in the hypergraph. All other operations described in Section 4 can be called from C++ as well.

The `hyp` distribution includes additional C++ example code and `doxygen` API documentation.

6 Optimizing hypergraph feature weights

`hyp` provides functionality to optimize hypergraph feature weights from training data. It trains a regularized conditional log-linear model, also known as conditional random field (CRF), with optional hidden derivations (Lafferty et al. (2001), Quattoni et al. (2007)). The training data consist of observed input-output hypergraph pairs (x, y). x and y are non-loopy hypergraphs and so may represent string, lattice, tree, or forest. A user-defined function, which is compiled and loaded as a shared object, defines the search space of all possible outputs given any input x, with their features. `hyp` then computes the CRF function value, feature expectations and gradients, and calls gradient-based optimization methods like L-BFGS or Adagrad (Duchi et al., 2010). This may be used to experiment with and train sequence or tree-based models. For details, we refer to the `hyp` tutorial (Dreyer and Graehl, 2015).

[3] If the best path rather than the full composition is requested, that composition is lazy best-first and may, weights depending, avoid creating most of the composition.

[4] In the current `hyp` version, the Earley-inspired algorithm computes the full composition and should therefore be used with smaller grammars.

[5] Using the C++ API to perform a sequence of operations, one can keep intermediate hypergraphs in memory and so avoid the cost of disk write and read operations.

7 Conclusions

We have presented `hyp`, an open-source toolkit for representing and manipulating weighted directed hypergraphs, including functionality for learning arc feature weights from data. The `hyp` toolkit provides a C++ library and a command line executable. Since `hyp` seamlessly handles trees, forests, strings, lattices and finite-state transducers and acceptors, it is well-suited for a wide range of practical problems in NLP (e.g., for implementing a parser or a machine translation pipeline) and related areas. `hyp` is available for download at `github.com/sdl-research/hyp`.

Acknowledgments

We thank Daniel Marcu and Mark Hopkins for guidance and advice; Kevin Knight for encouraging an open-source release; Bill Byrne, Abdessamad Echihabi, Steve DeNeefe, Adria de Gispert, Gonzalo Iglesias, Jonathan May, and many others at SDL Research for contributions and early feedback; the anonymous reviewers for comments and suggestions.

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer.

Markus Dreyer and Jonathan Graehl. 2015. Tutorial: The `hyp` hypergraph toolkit. `http://goo.gl/O2qpi2`.

J. Duchi, E. Hazan, and Y. Singer. 2010. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Christopher Dyer. 2010. *A formal model of ambiguity and its applications in machine translation.* Ph.D. thesis, University of Maryland.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending datalog for modern AI. In *Datalog Reloaded*, pages 181–220. Springer.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Practical weighted dynamic programming and the Dyna language. In *In Advances in Probabilistic and Other Parsing*.

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–8, Philadelphia, July.

Giorgio Gallo, Giustino Longo, and Stefano Pallottino. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201.

Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233.

Jonathan Graehl. 2005. Context-free algorithms. arXiv:1502.02328 [cs.FL].

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64. Association for Computational Linguistics.

Dan Klein and Christopher D. Manning. 2005. Parsing and hypergraphs. In *New developments in parsing technology*, pages 351–372. Springer.

Donald E. Knuth. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Shankar Kumar, Wolfgang Macherey, Chris Dyer, and Franz Och. 2009. Efficient minimum error rate training and minimum bayes-risk decoding for translation hypergraphs and lattices. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1*, pages 163–171. Association for Computational Linguistics.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA.

Mehryar Mohri. 2009. Weighted automata algorithms. In *Handbook of weighted automata*, pages 213–254. Springer.

A. Quattoni, S. Wang, L. P. Morency, M. Collins, and T. Darrell. 2007. Hidden conditional random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(10):1848–1852.