Tutorial: The `hyp` hypergraph toolkit

Markus Dreyer
SDL
6060 Center Drive Suite 150
Los Angeles, CA 90045
mdreyer@sdl.com

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

June 12, 2015

Abstract

We present `hyp`, a general toolkit for the representation, manipulation, and optimization of weighted hypergraphs. Finite-state machines are modeled as a special case.

Contents

1 Introduction **2**

2 Definitions **3**

3 Representing hypergraphs **4**
3.1 Constructing a one-arc toy hypergraph 4
3.2 Trees and forests 4
3.2.1 Trees 5
3.2.2 Reducing redundancy in the text format 6
3.2.3 Forests 7
3.3 Constructing strings, lattices, and FSMs 7
3.3.1 Strings 7
3.3.2 Lattices and general finite-state machines 8
3.4 Transducers 9
3.5 Semirings and features 10

4 Using the hyp executable **10**
4.1 Compose . . . 10
4.1.1 Natural language parsing . . . 11
4.1.2 Hypergraph rescoring . . . 12
4.2 PruneToBest, Best . . . 12
4.3 Inside . . . 12
4.4 Invert, Project . . . 15
4.5 Union . . . 16
4.6 Concat . . . 17
4.7 Other `hyp` commands . . . 17

5 Using the C++ API **18**
5.1 Creating a forest . . . 18
5.2 Hypergraph properties . . . 19
5.2.1 Arc storage . . . 19
5.2.2 Canonical lexical states . . . 19
5.3 Operating on a hypergraph . . . 19

6 Optimizing hypergraph feature weights **20**
6.1 Model . . . 20
6.2 Defining the search space . . . 21
6.3 Example . . . 22
6.4 Running the demo . . . 22

1 Introduction

The `hyp` toolkit provides data structures and algorithms to process weighted directed hypergraphs.

Such hypergraphs are important in natural language processing and machine learning. Their use arises naturally in parsing (Klein and Manning, 2005; Huang and Chiang, 2005), syntax-based machine translation and other tree-based models, as well as in logic (Gallo et al., 1993) and weighted logic programming (Eisner and Filardo, 2011).

The `hyp` toolkit allows for the representation and manipulation of weighted directed hypergraphs. It provides data structures for hypergraphs as well as many algorithms: `compose`, `project`, `invert`, the shortest path algorithm,

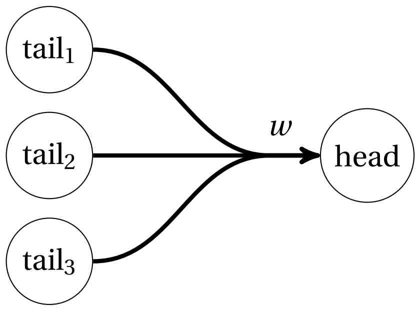

Figure 1: An arc leading from three tail states to a head state, with weight w.

the inside and outside algorithm, and more. In addition, it provides functionality to optimize hypergraph feature weights from training data.

2 Definitions

A weighted directed hypergraph (henceforth just *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 of the edge (see Figure 1). Semirings are described in Section 3.5.

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 $F \in V$.

The set of incoming arcs into a state s is called the *Backward Star* of s, or short, BS(s). Formally, $\text{BS}(s) = \{a \in E : h(a) = s\}$.

A *path* π is a sequence of arcs $\pi = (a_1 \ldots a_k) \in E^*$ such that $\forall a \in \pi, \forall t \in T(a), (\exists a' \in \pi : h(a') = t) \vee \text{BS}(t) = \emptyset$. In words, each tail state t of each arc on the path must be the head of some arc on the path, unless t has no incoming arcs (i.e., is a leaf state). 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, $h(a_k) = F$.

The hypergraphs we consider, in which each arc has exactly one head, have been called *B-hypergraphs* (Gallo et al., 1993), as opposed to a more general version in which multiple head states are allowed.

```
0 <- 1 2 3 / 0.693
FINAL <- 0
```

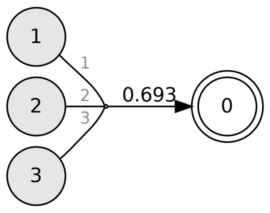

Table 1: The text and visual representation of a simple hypergraph with only a single arc.

3 Representing hypergraphs

`hyp` uses a simple text format for hypergraphs, which makes it easy to write down and construct toy hypergraphs and play with them.

3.1 Constructing a one-arc toy hypergraph

Let's construct our first hypergraph! It has only a single arc, which is written as `0 <- 1 2 3 / 0.693`, denoting the head state 0, the tail states 1, 2, 3, and the weight 0.693. The text and visual representations are shown in Table 1. Each state has an ID (see number in circle). State 0 is the final state and is marked as such: `FINAL <- 0`. Similar to finite-state machine convention, the final state is drawn as a double circle. The small gray numbers denote the ordering of the tails of the arc. We draw any leaf node gray.[1] The visual representation has been created with the `hyp Draw` command.

That's nice, but usually we also need labels. As described in Section 2, our hypergraph arcs do not have labels, but states do. Table 2 shows how labels are added. The result is similar to the typeset example in Figure 1.

3.2 Trees and forests

Hypergraphs can represent trees and (packed or non-packed) forests, but they can also—as special cases—represent strings, lattices, or finite-state machines. Let's start with trees and forests; we'll come back to strings, lattices, etc., later.

[1]This is in analogy to graphical models, where *observed* nodes are drawn gray, denoting that they do not need to be derived or inferred.

```
0(head) <- 1(tail1) 2(tail2) 3(tail3) / 0.693
FINAL <- 0(head)
```

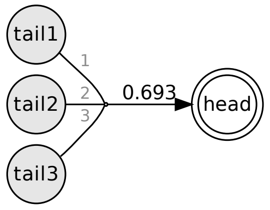

Table 2: The same hypergraph as in Table 1, but here the states are labeled. For simplicity, we leave out the state IDs of labeled states in the visual representation.

```
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
```

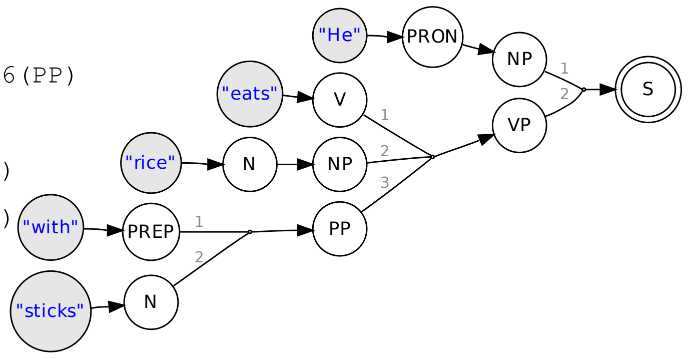

Table 3: A simple tree.

3.2.1 Trees

A tree is a typical, simple hypergraph. Table 3 shows a labeled, unweighted tree and the corresponding text format. Again, the gray numbers 1, 2, 3, ... in the figure denote the ordering of the tails of an arc. State IDs are not shown wherever labels are given. The text format contains one arc per line. Each arc has the following format:

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

Each head or tail state consists of a state ID and a label, `ID(label)`. The label is enclosed in double-quotes if it is lexical (i.e., a word). Each arc may have a weight (optional):

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

A weight is typically a negative log number (i.e., the cost of the arc). A final state `n` is marked as `FINAL <- n`.

```
(S) <- (NP1) (VP1)
(NP1) <- (PRON)
(VP1) <- (V1) (NP2) (PP)
(PRON) <- ("He")
(V1) <- ("eats")
(NP2) <- (N1)
(PP) <- (PREP) (N2)
(N1) <- ("rice")
(PREP) <- ("with")
(N2) <- ("sticks")
FINAL <- (S)
```

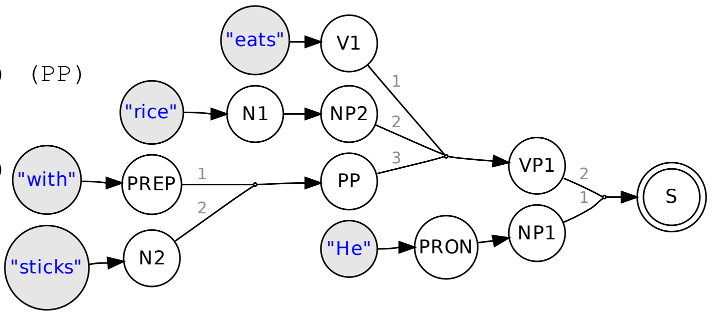

Table 4: Leaving out all state IDs in the text format.

3.2.2 Reducing redundancy in the text format

To reduce redundancy, we can leave out labels wherever they can be inferred:

```
0(S) <- 1 2
1(NP) <- 3
2(VP) <- 4 5 6
3(PRON) <- 10("He")
4(V) <- 11("eats")
5(NP) <- 7
6(PP) <- 8 9
7(N) <- 12("rice")
8(PREP) <- 13("with")
9(N) <- 14("sticks")
FINAL <- 0
```

To further reduce redundancy, we can also leave out state IDs wherever they can be inferred:

```
0(S) <- 1 2
1(NP) <- 3
2(VP) <- 4 5 6
3(PRON) <- ("He")
4(V) <- ("eats")
5(NP) <- 7
6(PP) <- 8 9
7(N) <- ("rice")
8(PREP) <- ("with")
9(N) <- ("sticks")
FINAL <- 0
```

We can leave out all state IDs. In the above example, however, states `1` and `5` both have the same label `NP`, so if we just leave out the state labels these states will be seen as identical. We give them different labels and can leave out all state IDs, see Table 4.

```
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)
```

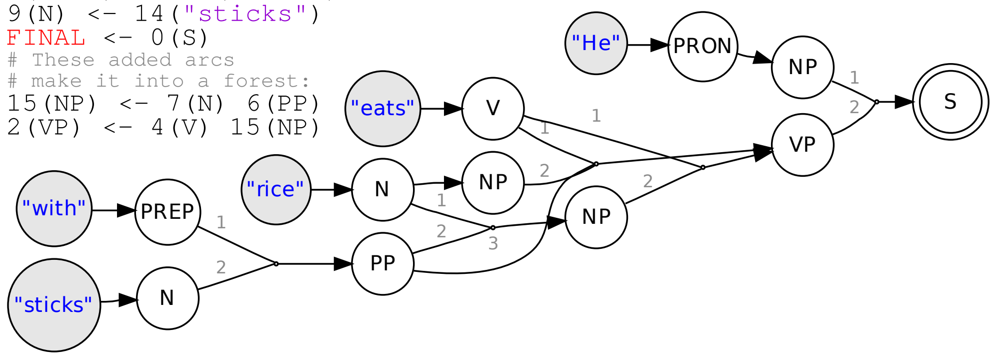

Table 5: A packed forest.

3.2.3 Forests

A forest is a hypergraph that contains more than one tree. The forest may be *packed*, in which case the trees share substructure, just like strings in a lattice. We can turn the tree from Section 3.2.1 into a forest by adding some arcs, see Table 5. That forest represents two interpretations of the sentence, where he (1) eats rice that has sticks or (2) eats rice using sticks as a tool.

3.3 Constructing strings, lattices, and FSMs

3.3.1 Strings

In the finite-state world, a string is often encoded as a simple, flat-line finite-state machine. It is a left-recursive split of the string as `( ( ( He ) eats ) rice )`, i.e., we first have “He”, then combine it with “eats”, then combine the result with “rice” to accept the whole string, see Table 6.

We can do something similar using hypergraphs, see Table 7. All leaves in this lattice-like hypergraph are “axioms”; they are given. The hypergraph can be traversed bottom-up as follows: First we have (start) state 0 and the “he” state. Given those, we reach state 1. Given state 1 and the axiom “eats” we reach state 2. Given state 2 and axiom “rice”, we reach final state 3. (The start state

```
0 1 he
1 2 eats
2 3 rice
3
```

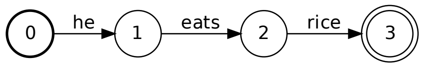

Table 6: A one-sentence finite-state machine. The text here is in the OpenFst text format.

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

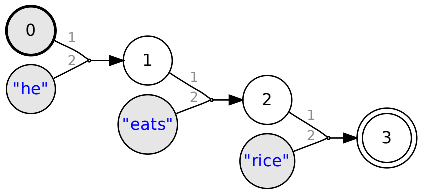

Table 7: A sentence expressed as a simple hypergraph.

```
START <- 0
1 <- 0 ("he")
2 <- 1 ("eats")
2 <- 1 ("likes")
3 <- 2 ("rice")
FINAL <- 3
```

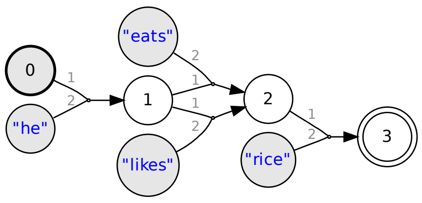

Table 8: A simple lattice hypergraph.

0 is optional here.) The figure can be understood as an unusual way to draw a lattice/finite-state machine: The arc from 0 to 1 has its label drawn as a balloon hanging off the arc. Similarly, for the arcs `1 -> 2` and `2 -> 3`.

3.3.2 Lattices and general finite-state machines

We can add more arcs to the string to turn it into a lattice, see Table 8.

Or we can turn it into a general finite-state machine by adding a loop arc:

```
START <- 0
1 <- 0 ("he")
2 <- 1 ("eats")
2 <- 2 ("rice")
FINAL <- 2
```

We call these special, restricted hypergraphs, in which each arc has a structural tail state followed by a lexical tail state, *regular* or *finite-state* hypergraphs.

Note to avoid confusion: An arc like `2 <- 1 ("likes")` has two tails:

State 1, for which no label is specified, and the state with label `("likes")`, for which no state ID is specified.

3.4 Transducers

Any (leaf) state may have "rewrite information", i.e., a specific output label, in addition to the input label. For example, the following machine rewrites "he eats rice" to "he ate rice"; it is a finite-state transducer (in `hyp` hypergraph format):

```
START <- 0
1 <- 0 ("he")
2 <- 1 ("eats" "ate")
2 <- 1 ("likes" "liked")
3 <- 2 ("rice")
FINAL <- 3
```

General hypergraphs (trees, forests, etc.) may have output labels as well. The following hypergraph rewrites "he eats rice with sticks" to "he ate pasta"; it also parses the input string into a structure that (unlike the example above) is not just left-recursive:

```
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" "ate")
5(NP) <- 7(N)
6(PP) <- 8(PREP) 9(N)
7(N) <- 12("rice" "pasta")
8(PREP) <- 13("with" <eps>)
9(N) <- 14("sticks" <eps>)
FINAL <- 0(S)
```

Special symbols like epsilon, phi, rho, sigma are written with brackets, as `<eps>`, `<phi>`, `<rho>`, `<sigma>`.

When a state has no explicit output label the output label is assumed to be identical to the input label—a convention from the finite-state world.

In order to rewrite a string, we compose it with the input (see Section 4.1). Our code uses a generalized composition algorithm that can compose a (weighted) general hypergraph with a (weighted) finite-state hypergraph. It is a generalization of the Earley algorithm, see (Earley, 1970), (Stolcke, 1995), (Eisner et al., 2005), (Dyer, 2010). Like with finite-state machines, we can project to the output after composing, to obtain the resulting acceptor (see Section 4.4).

3.5 Semirings and features

Each hypergraph uses a particular semiring. Semirings are used to specify the type of the weights and define how weights are added and multiplied. We provide the two standard semirings log and Viterbi, as well as the expectation semiring (Eisner, 2002). The semiring can be specified when using the command line tools (see below) or when constructing a hypergraph in C++.

Using the expectation semiring enables us to place features on the arcs. Example:

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

where the arc has weight 3.2 and feature 0 and feature 8 fire once, respectively (so their expected values are both 3.2).

We also added a new semiring called feature semiring, which behaves exactly like the Viterbi semiring but allows for a feature vector on each arc, like the expectation semiring.

Using the expectation or the feature semiring, we can keep track of what features fire on what arcs when we compose several hypergraphs (or perform other operations). Using standard algorithms that we implemented (e.g., the inside-outside algorithm, see below), it is possible to train hypergraphs as a CRF (see Section 6), for example. See Section 4.3 for an example that demonstrates the behavior of different semirings.

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> <inputfiles>`, where the input files are hypergraphs in the text format described above, and the `<command>` may be `Compose`, `Best`, or others. We now describe some of these commands.

4.1 Compose

As an example, consider composing a CFG hypergraph with an input sentence hypergraph:

```
$ cat cfg.hyp
FINAL <- (S)
(S) <- (NP) (VP)      / 0.1823216 # = -log(5/6)
```

```
FINAL <- 0(S)
0(S) <- 2(NP) 1(VP) / 0.182322
0(S) <- 2(NP) 6(V) 5(NP) / 1.79176
2(NP) <- 3(N) / 0
3(N) <- 4("he") / 0.693147
1(VP) <- 6(V) 5(NP) / 0
6(V) <- 7("eats") / 0
5(NP) <- 8(N) / 0
8(N) <- 9("rice") / 1.20397
```

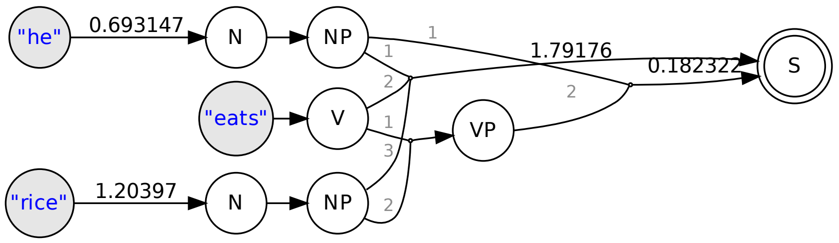

Table 9: Composition result: A packed forest.

```
(S) <- (NP) (V) (NP) / 1.791759  # = -log(1/6)
(NP) <- (N)
(VP) <- (V) (NP)
(V) <- ("eats")
(N) <- ("he")   / 0.6931472 # = -log(5/10)
(N) <- ("rice") / 1.203973  # = -log(3/10)
(N) <- ("fish") / 1.609438  # = -log(2/10)

$ cat sent.hyp
START <- 0
1 <- 0 4("he")
2 <- 1 5("eats")
3 <- 2 6("rice")
FINAL <- 3

$ hyp Compose cfg.hyp sent.hyp
```

The result is a packed parse forest, see Table 9.

4.1.1 Natural language parsing

We can do weighted lattice parsing: A weighted context-free grammar (CFG) can be encoded as a hypergraph and can be composed with finite-state input (e.g., a string or a weighted lattice), using our composition algorithm. The result is the parse of the whole input structure. But since the CFG composition algorithm in `hyp` currently does not perform pruning it can only be used with small grammars. If both hypergraphs are *finite-state*, however, `hyp` switches to a fast finite-state composition algorithm.

4.1.2 Hypergraph rescoring

We can also rescore a weighted hypergraph by composing (or similarly, intersecting) with a finite-state machine, e.g., a language model.

4.2 PruneToBest, Best

To get an *n*-best list from any hypergraph, use `hyp Best`. For example, the two entries in the composition result (Table 9) are:

```
$ hyp Compose cfg.hyp sent.hyp | hyp Best --num-best=2
  n=1 2.07944 "he" "eats" "rice"
  n=2 3.68888 "he" "eats" "rice"
```

To print the best tree, use `PruneToBest`. Example:

```
$ hyp Compose cfg.hyp sent.hyp | hyp PruneToBest
  FINAL <- 0(S)
  0(S) <- 1(NP) 2(VP) / 0.182322
  1(NP) <- 5(N) / 0
  2(VP) <- 3(V) 4(NP) / 0
  3(V) <- 8("eats") / 0
  4(NP) <- 7(N) / 0
  5(N) <- 6("he") / 0.693147
  7(N) <- 9("rice") / 1.20397
```

Remember that all arc weights are interpreted as negative log numbers, i.e., costs, and the best parse is the one with the lowest cost.

4.3 Inside

The inside algorithm computes the cost of reaching each state starting from the leaves.

In Table 9, for example, the final state 0 can be reached via two different paths, which have weights 2.07944 and 3.68888, respectively. The sum in negative log space is $-\log(\exp(-2.07944) + \exp(-3.68888)) = 1.897117$. `Inside`, called on the hypergraph from Table 9, gives exactly that result for state 0 and lists the costs for reaching the other states as well:

```
$ hyp Inside foo.hyp
  0       1.89712
  1       1.20397
  2       0.693147
  3       0.693147
  4       0
  ...
```

If the Viterbi semiring is used, then the costs of any alternative paths are not added up, but the best among them is picked, so we have:

```
$ hyp Inside --arc-type=viterbi foo.hyp
  0       2.07944
  1       1.20397
  2       0.693147
  3       0.693147
  4       0
  ...
```

The feature semiring computes arc weights just like the Viterbi semiring, but in addition it accumulates features, where feature values are not in log space. Consider the hypergraph in Table 9, but with real-valued features added on some arcs:

```
$ cat feats.hyp
FINAL <- 0(S)
0(S) <- 2(NP) 1(VP) / 0.182322[0=1.3,1=2]
0(S) <- 2(NP) 6(V) 5(NP) / 1.79176[0=1.5,2=3]
2(NP) <- 3(N) / 0
3(N) <- 4("he") / 0.693147
1(VP) <- 6(V) 5(NP) / 0
6(V) <- 7("eats") / 0
5(NP) <- 8(N) / 0
8(N) <- 9("rice") / 1.20397[1=4]
```

On the first arc, feature 0 fires 1.3 times and feature 1 fires twice. The output of `Inside` looks like the Viterbi result above, but with the accumulated feature vectors added:

```
$ hyp Inside --arc-type=feature feats.hyp
  0       2.07944[0=1.3,1=6]
  1       1.20397[1=4]
  2       0.693147
  3       0.693147
  4       0
  ...
```

If the expectation semiring is used on a hypergraph with features, the inside algorithm computes the expected value of each feature for each state. For the final state, we get the expected value of each feature in the overall hypergraph.[2]

For the expectation semiring, each arc weight is a tuple of the weight w and the feature vector v times w (Eisner, 2002). We take the negative log of w and of vw. For example, in the following example, the first arc has $w = 5/6$ and $v = [1.3, 2]$, so $vw = [1.08, 1, 67]$. The numbers on the arc are the negative log of these, so we have (rounded) $0.1823[0 = -0.0800, 1 = -0.5108]$, see here:

```
$ cat expectation.hyp
FINAL <- 0(S)
0(S) <- 2(NP) 1(VP) / 0.182322[0=-0.08004271,1=-0.5108256]
0(S) <- 2(NP) 6(V) 5(NP) / 1.79176[0=1.386294,2=0.6931472]
2(NP) <- 3(N) / 0
3(N) <- 4("he") / 0.693147
1(VP) <- 6(V) 5(NP) / 0
6(V) <- 7("eats") / 0
5(NP) <- 8(N) / 0
8(N) <- 9("rice") / 1.20397[1=-0.1823216]
```

`Inside` lists the (negative log of the) feature expectations at each state:

```
$ hyp Inside --arc-type=expectation expectation.hyp
  0       1.89712[0=1.60943,1=0.510826,2=2.59026]
  1       1.20397[1=-0.182322]
  2       0.693147
  3       0.693147
  4       0
  ...
```

Observe that the inside cost of the final state 0 is listed as 1.89712, i.e., $-\log(0.15)$, just like when the log semiring is used (see above). The expected value of fea-

[2]This property of the expectation semiring is remarkable because the feature expectations can otherwise only be computed by running the inside *and* the outside passes. Note, however, that the expectation semiring accumulates feature expectations in all its inside weights, so those vectors become denser for states closer to the final state. It is often more efficient to compute feature expectations by running the inside and outside passes in the log semiring, then passing over the hypergraph again and accumulating the feature expectations in just one vector. We use this method in `sdl/Hypergraph/FeatureExpectations.hpp`.

ture 0 is listed as −1.60943, i.e., $-\log(0.2) = -\log(0.125 \times 1.3 + 0.025 \times 1.5)$, because that feature fires 1.3 times on the best path and 1.5 times on the other path.

4.4 Invert, Project

The `Invert` operation swaps input and output labels of all arcs. For example, the following toy transducer changes *eat* or *sleep* to their past tense forms (while accepting any other word without change via the special `<rho>` symbol):

```
$ cat past_tense.hyp
FINAL <- 0
START <- 0
0 <- 0 ("eats" "ate")
0 <- 0 ("sleeps" "slept")
0 <- 0 (<rho>)
```

The inverted version changes those past tense forms back to *eat* or *sleep*:

```
  cat past_tense.hyp | hyp Invert
FINAL <- 0
START <- 0
0 <- 0 1("ate" "eats") / 0
0 <- 0 2("slept" "sleeps") / 0
0 <- 0 3(<rho>) / 0
```

The `project` operation projects to the output labels by removing the input labels, or vice versa. This is often useful for obtaining the output after composing. Example:

```
$ cat sent.hyp
START <- 0
1 <- 0  ("he")
2 <- 1  ("eats")
3 <- 2  ("rice")
FINAL <- 3

$ hyp Compose sent.hyp past_tense.hyp | hyp Project
START <- 0
1 <- 0 6("he") / 0
```

```
0 <- 1("a") 2("b") / 0
3 <- 4("x") 5("y") 6("z") / 0
8 <- 0 7(<eps>) / 0
8 <- 3 7(<eps>) / 0
9 <- 10("a") 11("b") / 0
FINAL <- 13
13 <- 8 12(<eps>) / 0
13 <- 9 12(<eps>) / 0
```

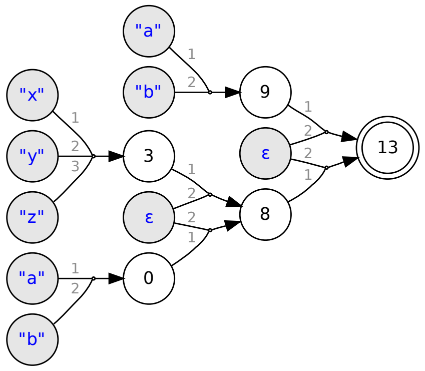

Table 10: Union result.

```
2 <- 1 5("ate") / 0
FINAL <- 3
3 <- 2 4("rice") / 0
```

4.5 Union

The `hyp Union` command takes the union of two or more hypergraphs. The union operation implements an OR logic, so that in the resulting hypergraph, one can follow the paths of any of the input hypergraphs. Example:

```
$ cat ab.hyp
FINAL <- 0
0 <- ("a") ("b")

$ cat kl.hyp
FINAL <- 0
0 <- ("k") ("l")

$ cat xyz.hyp
FINAL <- 0
0 <- ("x") ("y") ("z")

$ hyp Union ab.hyp kl.hyp xyz.hyp
```

The result is shown in Table 10.

```
0 <- 1("a") 2("b") / 0
3 <- 4("k") 5("l") / 0
6 <- 0 3 / 0
7 <- 8("x") 9("y") 10("z") / 0
FINAL <- 11
11 <- 6 7 / 0
```

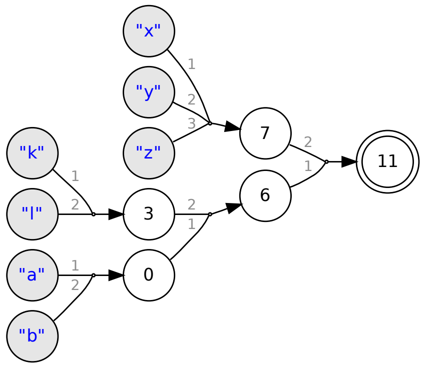

Table 11: Concat result.

4.6 Concat

`hyp Concat` concatenates two or more hypergraphs. If you use `Concat` instead of `Union` in the example above, you obtain the output shown in Table 11.

4.7 Other hyp commands

The `hyp` toolkit also provides the following commands, in addition to the ones discussed above:

- `Complement` creates the complement of an unweighted finite-state hypergraph
- `ConvertStrings` converts string input into string hypergraphs (i.e., hypergraphs like the one shown in Table 7)
- `Determinize` determinizes a finite-state hypergraph (currently only unweighted acceptors allowed)
- `Draw` draws a hypergraph in `dot` format (see the figures in this tutorial)
- `Empty` decides if a hypergraph is empty, i.e., has no valid paths
- `GetString` prints the string contained in a one-path hypergraph
- `Prune` prunes away all unreachable arcs of a hypergraph

- `Reverse` reverses a hypergraph
- `Reweight` modifies arc weights of a hypergraph
- `SamplePath` generates sample paths from a hypergraph

5 Using the C++ API

We now describe basic usage of the C++ API. See also the complete example code in `HypDemo/src/HypDemo.cpp`.

The full API documentation can be generated using `doxygen doxygen/doxy.conf`.

5.1 Creating a forest

The following code creates a toy forest hypergraph:

```cpp
using namespace sdl;
using namespace sdl::Hypergraph;
typedef ViterbiWeight Weight;
typedef ArcTpl<Weight> Arc;
MutableHypergraph<Arc> hyp;
IVocabularyPtr voc(createDefaultVocab());
Sym john = voc->add("John", kTerminal);
Sym loves = voc->add("loves", kTerminal);
Sym likes = voc->add("likes", kTerminal);
Sym mary = voc->add("Mary", kTerminal);
Sym vp = voc->add("VP", kNonTerminal);

StateId s0 = hyp.addState(vp);
StateId s1 = hyp.addState();

hyp.addArc(new Arc(Head(s0),
                  Tails(hyp.addState(likes), hyp.addState(mary)),
                  Weight(2.0f)));
hyp.addArc(new Arc(Head(s0),
                  Tails(hyp.addState(loves), hyp.addState(mary)),
                  Weight(4.0f)));
hyp.addArc(new Arc(Head(s1), Tails(hyp.addState(john), s0),
                  Weight(8.0f)));

hyp.setVocabulary(voc);
hyp.setFinal(s1);
```

The code first creates a vocabulary and adds a few terminal symbols and a nonterminal symbol. It then creates the hypergraph and sets its vocabulary. It then adds states and arcs where each arc has a head, tails, and a weight. It then sets the final state.

5.2 Hypergraph properties

Each hypergraph object has properties, expressed as bits in a 64-bit number, see `sdl/Hypergraph/Properties.hpp`. These can describe properties of the hypergraph at any point in time, e.g., `kFsm`, which is on by default until non-finite-state arcs are added, or `kCfg`. Other properties act as policies that control behavior such as an arc storage discipline. We now describe two of these policies.

5.2.1 Arc storage

By default, each hypergraph object keeps track of incoming and outgoing arcs per state. For large hypergraphs, one might want to save memory by storing only incoming or only outgoing arcs. Typically, if the hypergraph represents a regular language (e.g., string, lattice) just storing outgoing arcs are sufficient. For context-free languages (e.g., tree, forest) incoming arcs are sufficient for running most algorithms on the hypergraph. These property bits can be set in the constructor:

```
  MutableHypergraph<Arc> hyp1(kStoreInArcs);
  MutableHypergraph<Arc> hyp2(kStoreOutArcs);
  MutableHypergraph<Arc> hyp3(kStoreInArcs | kStoreOutArcs); // default
```

5.2.2 Canonical lexical states

Another useful property is `kCanonicalLex`. It has the effect that any terminal label will be associated with one particular ("canonical") state in the hypergraph, so that multiple calls to `addState` with the same terminal symbol return the same state ID.

Note that setting properties in the constructor overrides the default properties, so an arc storage property must be chosen together with `kCanonicalLex`:

```
  MutableHypergraph<Arc> hyp1(kStoreInArcs | kCanonicalLex);
```

In the C++ code above (Section 5.1), if `kCanonicalLex` were set on `hyp` the first call to `hyp.addState(mary)` (line 18) would create a new state with label `mary` and the second call (line 21) would return that same state ID instead of creating a new state.

5.3 Operating on a hypergraph

After constructing a hypergraph we can operate on it. We can, for example, compute the inside costs:

```
boost::ptr_vector<Weight> costs;
sdl::Hypergraph::insideAlgorithm(hyp, &costs);

for (unsigned i = 0, n = costs.size(); i < n; ++i)
  std::cout << i << " " << costs[i] << '\n';
```

6 Optimizing hypergraph feature weights

6.1 Model

The `hyp` toolkit provides functionality to optimize hypergraph feature weights from training data, see the `Optimize` directory.

The `Optimize` code trains a 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 N observed input-output pairs (x_i, y_i), where $i = 1, \ldots, \mathrm{N}$. Each x_i and y_i is a hypergraph, which may, as a special case, mean a string, lattice, tree, or other hypergraph structures.[3]

Each (x_i, y_i) pair may have multiple unobserved derivations $\mathcal{D}(x_i, y_i)$. For example, x_i may be an input string and y_i an observed output string, where multiple different alignments between these are allowed (Dreyer et al., 2008); similarly, unobserved trees may be allowed. Or x_i may be an input string and y_i an observed parse tree, but with underspecified nonterminals, where specific nonterminal annotations are a latent variable (Petrov and Klein, 2008).

K features are defined that describe properties of (x, y) pairs and their derivations, such that $f_k(x, y, d)$ is the value of the kth feature on (x, y) in derivation d. The features must be local to the arcs of the hypergraphs. A feature weight vector $\vec{\theta}$ assigns a weight, or importance, to each feature, where each weight $\theta_k \in [-\infty, +\infty]$. In training, we optimize the feature weights, i.e., we find a $\vec{\theta}$ that maximizes the objective function.

The objective function is defined as the conditional regularized log-likelihood of the training data,

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{\mathrm{N}} \log p_{\boldsymbol{\theta}}(y_i \mid x_i) - \mathrm{C} \cdot \mathrm{R}(\boldsymbol{\theta}) \qquad (1)$$

[3]These hypergraphs should not contain loops, such as unbounded insertions, because these can cause the objective function (Equation 1) to diverge during optimization. This problem can be solved by adding a special regularizer, see Dreyer (2011).

where

$$p_{\boldsymbol{\theta}}(y_i \mid x_i) = \frac{u_{\boldsymbol{\theta}}(x_i, y_i)}{\sum_{y'} u_{\boldsymbol{\theta}}(x_i, y')} \tag{2}$$

and $u_{\boldsymbol{\theta}}(x_i, y_i)$ is a score that measures how well x_i and y_i go together, considering all latent derivations, given the model parameters θ:

$$u_{\boldsymbol{\theta}}(x_i, y_i) = \sum_{d \in \mathscr{D}(x_i, y_i)} \exp \sum_k \theta_k \cdot f_k(x_i, y_i, d) \tag{3}$$

R(**θ**) in Equation 1 is a regularizer that can prevent overfitting by adding penalties for feature weights that deviate too far from zero. The strength of the regularizer is controlled by C, which must be tuned on held-out data. `hyp` implements L_1 and L_2 regularization.

We use gradient-based optimization. The gradient of the *k*th feature is given by the following expression (cf. Sutton and McCallum (2012)):

$$\begin{aligned} \frac{\partial}{\partial f_k} = \textstyle\sum_i (\quad & \textstyle\sum_d \; p_{\boldsymbol{\theta}}(d \mid x_i, y_i) f_k(x_i, y_i, d) \\ & - \textstyle\sum_{y', d'} p_{\boldsymbol{\theta}}(y', d' \mid x_i) f_k(x_i, y', d')) - \mathrm{C} \cdot \frac{\partial \mathrm{R}(\boldsymbol{\theta})}{\partial f_k} \end{aligned} \tag{4}$$

where d' ranges over all derivations of a given x_i, y' pair.

Therefore, the gradient of the *k*th feature is the difference of its expected value given the input-output pairs of the training data minus its expected value given just the input sides of the training data.

6.2 Defining the search space

In `hyp`, the user defines the hypergraphs necessary to compute the numerator and the denominator of Equation 2. That is, for each training example, two hypergraphs are provided by the user: a hypergraph “clamped” to the input and output, and an “unclamped” hypergraph that is not clamped to any output. The “unclamped” hypergraph represents the search space for a given example. The arc weights in these hypergraphs contain the features to optimize.

In the current version, there is no feature template mechanism that would create such hypergraphs for the user given some textual training data. The exact clamped and unclamped hypergraphs for each training example must be provided by user client code. However, see `CrfDemo` for example client code that creates the hypergraphs from a CoNLL input file. That demo implements a chunker with some predefined features that are not fully configurable nor complete.

6.3 Example

As a simplistic example for clamped and unclamped hypergraphs, suppose we train a linear-chain named entity tagger with only two possible tags per word `I` (i.e., the word is inside a named entity) and `O` (i.e., the word is outside a named entity). Each trainig example (x, y) is an input x consisting of a sequence of POS-tagged words and an output y consisting of the observed named entity tag sequence. For a sequence x=("John/NE, loves/V, Mary/NE") and observed tags y=(I, O, I), we have the following clamped hypergraph:

```
START <- 0
1 <- 0 ("John/NE" "I") / 0[0=0,1=0,2=0,3=0]
2 <- 1 ("loves/V" "O") / 0[4=0,5=0,6=0,7=0]
3 <- 2 ("Mary/NE" "I") / 0[8=0,9=0,2=0,10=0]
FINAL <- 3
```

This assumes a simple feature set: Each of the features looks at the current named entity tag and one of the following: {current word, previous POS tag, current POS tag, next POS tag}. See table Table 12 for a mapping from feature IDs to feature names. Feature 2 (t_0=I $\wedge$ p_0=NE) fires both on the first and the third arc.
The unclamped hypergraph would look as follows; it contains the clamped hypergraph and adds any tagging alternatives as additional arcs:

```
START <- 0
1 <- 0 ("John/NE" "I") / 0[0=0,1=0,2=0,3=0]
1 <- 0 ("John/NE" "O") / 0[11=0,12=0,13=0,14=0]
2 <- 1 ("loves/V" "O") / 0[4=0,5=0,6=0,7=0]
2 <- 1 ("loves/V" "I") / 0[15=0,16=0,17=0,18=0]
3 <- 2 ("Mary/NE" "I") / 0[8=0,9=0,2=0,10=0]
3 <- 2 ("Mary/NE" "O") / 0[19=0,20=0,13=0,21=0]
FINAL <- 3
```

6.4 Running the demo

To run the `CrfDemo`, type

```
$ CrfDemo/demo.sh
```

This will run the following commands:

ID	Name
0	t_0=I ∧ w_0=John
1	t_0=I ∧ p_{-1}=<s>
2	t_0=I ∧ p_0=NE
3	t_0=I ∧ p_{+1}=V
4	t_0=O ∧ w_0=loves
5	t_0=O ∧ p_{-1}=NE
6	t_0=O ∧ p_0=V
7	t_0=O ∧ p_{+1}=NE
8	t_0=I ∧ w_0=Jane
9	t_0=I ∧ p_{-1}=V
10	t_0=I ∧ p_{+1}=</s>
11	t_0=O ∧ w_0=John
12	t_0=O ∧ p_{-1}=<s>
13	t_0=O ∧ p_0=NE
14	t_0=O ∧ p_{+1}=V
…	…

Table 12: Simple feature set

```
Release/Optimization/Optimize --model CrfDemo \
  --config CrfDemo/example/config.yml \
  --search-space search-space-train \
  --optimize optimize-lbfgs
Release/Optimization/Optimize --model CrfDemo \
  --config CrfDemo/example/config.yml \
  --search-space search-space-test \
  --test-mode
```

The option `--config` specifies the command YAML config file; `--search-space` specifies the section in that YAML file that contains options to construct the search space, i.e., the clamped and unclamped hypergraphs; `--optimize` specifies the YAML section that configures the optimization process. The optimization options are described in the help:

```
Release/Optimization/Optimize --help
```

The options to construct the search space generally depend on the client code that constructs the clamped and unclamped hypergraphs, which is dynamically loaded at runtime, according to the `--model` option (`CrfDemo` above). See `CrfDemo/CreateSearchSpace.hpp` for its options.

If you wish to create new code that constructs the clamped and unclamped hypergraphs for your specific CRF model you can copy the `CrfDemo` directory to `MyCrf` and insert your own code. It can be built with the following command:

```
make MyCrf-shared
```

which creates a shared library, `MyCrf/libMyCrf-shared.so`. This can then be used with `Optimize --model MyCrf`.

References

Markus Dreyer. *A Non-Parametric Model for the Discovery of Inflectional Paradigms from Plain Text Using Graphical Models over Strings*. PhD thesis, Johns Hopkins University, 2011.

Markus Dreyer, Jason Smith, and Jason Eisner. Latent-variable modeling of string transductions with finite-state methods. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1080–1089, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D08-1113`.

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

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

Jason Eisner. 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 2002. URL `http://cs.jhu.edu/~jason/papers/#eisner-2002-acl-fst`.

Jason Eisner and Nathaniel W. Filardo. Dyna: Extending datalog for modern AI. In *Datalog Reloaded*, pages 181–220. Springer, 2011. URL `http://link.springer.com/chapter/10.1007/978-3-642-24206-9_11`.

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

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

Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64. Association for Computational Linguistics, 2005. URL `http://dl.acm.org/citation.cfm?id=1654500`.

Dan Klein and Christopher D. Manning. Parsing and hypergraphs. In *New developments in parsing technology*, pages 351–372. Springer, 2005. URL `http://link.springer.com/chapter/10.1007/1-4020-2295-6_18`.

John Lafferty, Andrew McCallum, and Fernando Pereira. 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, 2001.

Slav Petrov and Dan Klein. Discriminative log-linear grammars with latent variables. In J. C. Platt, D. Koller, Y. Singer, and S.Editors Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS)*, pages 1153–1160. MIT Press, 2008.

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

Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995. ISSN 0891-2017. URL `http://dl.acm.org/citation.cfm?id=211190.211197`.

Charles Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4), 2012.