# A Primer on Graph Processing with Bolinas

J. Andreas, D. Bauer, K. M. Hermann, B. Jones, K. Knight, and D. Chiang

August 20, 2013

## 1 Introduction

This is a tutorial introduction to the Bolinas graph processing package. The software can be downloaded from `http://www.isi.edu/licensed-sw/bolinas`.

Bolinas lets you answer questions like:

- how can I represent a (possibly infinite) set of graphs G?
- is a specific graph H in the set of graphs G?
- how can I transform an input graph H1 into an output graph H2?
- if G is a set of weighted graphs, what is the weight of H in G?
- how can I estimate the weights of a synchronous graph grammar by training on pairs of input/output graphs?

Bolinas implements generic graph operations, so it can be applied in many fields. The authors happen to work in the field of natural language processing, where we often use data structures like strings, trees, and graphs to represent different aspects of language. For example, a string of letters may make up an English word, a string of English words may make up a sentence, and so on. Trees are useful for representing the nested syntactic parts of a sentence (noun phrases, prepositional phrases, etc), and graphs are useful for capturing the meaning of a sentence.

**Strings.** Much of language processing is concerned with strings. Formal automata can simplify string processing. A finite-state acceptor can capture an infinite set of strings (e.g., legal English sentences), and a finite-state transducer can transform strings of one type (e.g., English words) into strings of another type (e.g., part-of-speech tags). Efficient, generic algorithms have been developed for string automata, and these have been captured in several software toolkits, for example:

Toolkit: Carmel
http://www.isi.edu/licensed-sw/carmel
Tutorial: A Primer on Finite-State Software for Natural Language Processing
(Kevin Knight and Yaser Al-Onaizan, 1999)
http://www.isi.edu/licensed-sw/carmel/carmel-tutorial2.pdf

**Trees.** String transducers have trouble with large-scale re-ordering, of the kind we see in applications like machine translation. Re-ordering and other language phenomena are nicely captured with trees. Tree transducers, for example, can re-order whole sub-trees in a single step. The Tiburon toolkit implements efficient, generic algorithms on trees:

Toolkit: Tiburon
http://www.isi.edu/licensed-sw/tiburon
Tutorial: A Primer on Tree Automata Software for Natural Language Processing
(Jonathan May and Kevin Knight, 2008)
http://www.isi.edu/licensed-sw/tiburon/tiburon-tutorial.pdf

**Graphs.** The document you are reading takes us into the realm of graphs. How do we capture infinite sets of graphs with concise grammars? How do we transduce one graph into another graph, or tree, or string (or vice versa)?

Toolkit: Bolinas
http://www.isi.edu/licensed-sw/bolinas
Tutorial: this document
http://www.isi.edu/licensed-sw/tiburon/tiburon-tutorial.pdf

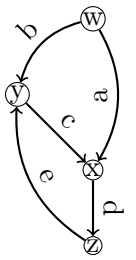
Although not strictly required, we recommend the reader to get familiar with Carmel and Tiburon as a pre-requisite to Bolinas.

# 2 Formalism

In computer science and mathematics, there are many types of graphs: directed, undirected, labeled, unlabeled, cyclic, acyclic, and so on. There are also many graph-generating formalisms. Bolinas implements hyperedge-replacement grammar (HRG), a framework that is well-studied in the theoretical literature. We first describe the kinds of graphs Bolinas accepts, and then we turn to HRG.

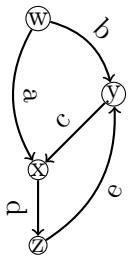
## 2.1 Graphs

Graphs consists of nodes connected by edges. Here is a sample graph:



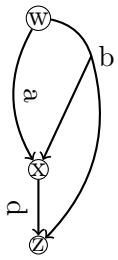

Bolinas graphs are:

- Rooted.
- Directed. Each edge has a distinguished head (source nodes) and tail (destination nodes).
- Edge-labeled and optionally node-labeled. In practice, we may not need node labels, or we may only need node labels on leaves (as in standard feature structures). When Bolinas is used for transduction it is unsafe to use node labels in the target graphs except for leaves.
- Unordered. Edges leaving a node are unordered. For example, this graph is identical with the one above, as far as Bolinas is concerned:



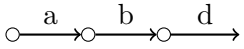

- Cyclic. In practice, most graphs we work with are acyclic. However, Bolinas supports operations on cyclic graphs.

Actually, Bolinas supports hypergraphs in addition to regular graphs. Hypergraphs have hyperedges. A hyperedge can have multiple tails (destination nodes), or no tail at all. Here is an example hypergraph with a hyperedge that has two tails:

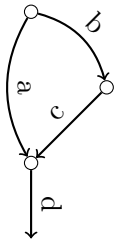


The tails of a hyperedge are ordered. In drawings (and in the syntax used to type graphs into Bolinas), the ordering of the tails is from left to right. Hyperedges are often necessary to define a (possibly infinite) set of graphs, even if the resulting language does not contain hypergraphs. The remainder of this tutorial will generally use the terms hypergraph and graph interchangeably.

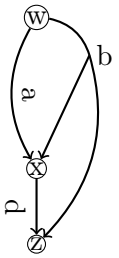
How do we type graphs into Bolinas? Strings and trees are very easy to type: *the boy is here.* or *(S (NP (DT the) (NN boy)) (VP(VBZ is) (NP here)))*. Graphs are more of a pain to type, because a node can have multiple parents (this is called re-entrancy). Here are a few graphs and their Bolinas representations:




```
(.  :a (.  :b (.  :d)))
```




```
(.  :a n1.  :b (.  :c (n1.  :d)))
```



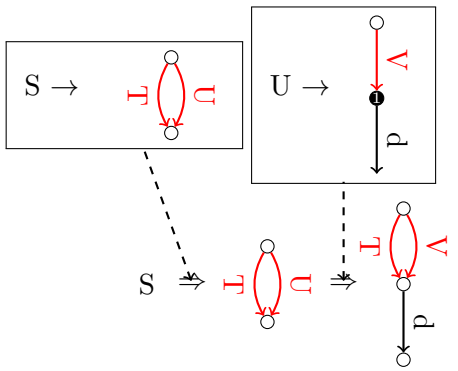

```
(.w :a (n1.x :d n2.z) :b n1.  n2.)
```

In the Bolinas graph format every '.' represents a node. The optional string following the '.' is a label for this node. The string preceding the '.' is an identifier refering to this node. Identifiers are used to refer back to a node later in a graph description. This is necessary if multiple edges connect to a node. For instance the third example contains a node with identifer 'n1' and the label 'x'. Node 'n1' is the tail of the edge labeled 'a'. We use the identifier to specify that the hyperedge 'b' also connects to 'n1'.

## 2.2 Graph Grammars

First, we want to represent a possibly-infinite set of graphs, just like a finite-state acceptor (FSA) represents a possible-infinite set of strings. For example, we might want to capture the set of graphs with an even number of nodes, or the set of all English semantic graphs that "make sense". Once we are able to concisely represent a set of graphs, we can perform set operations such as "test if graph H is in set G".

Bolinas implements hyperedge-replacement grammar (HRG). An HRG builds up (derives) a graph through a sequence of rule applications that rewrite intermediate graphs. We start with a single initial edge, then pick a rule that replaces that edge with some graph fragment. From that graph fragment, we may select another edge and similarly replace it. That way, the graph gets bigger and bigger:



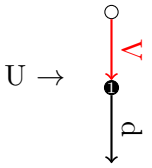

Every rule fragment contains a number of special nodes called external nodes (we usually fill these nodes in black in drawings). When the graph fragment is inserted into the graph its external nodes fuse (are identified) with the nodes that were connected by the hyperedge that is being replaced [1]. Because hyperedge tails are ordered, we sometimes need to make explicit how to connect the external nodes. We assign a number to each external node. The external node with the lowest number is fused with the first node in the tail etc.

Eventually, we stop doing replacements and wind up with a generated graph. When do we stop replacing edges? To answer this, we first say that there are two types of edges:

- Non-terminal edges (red in the example above). These are edges that must get replaced. Any graph with a non-terminal edge represents an incomplete derivation.
- Terminal edges (black in the example above). These are edges that never get replaced. Once they are put into a graph, they are there for good.

A derivation is complete when there are no more non-terminal edges.

Here is a simple HRG rule in Bolinas format:




```
U -> (.  :V$ (.*1 :d ));
```

[1] If you have ever heard of tree-adjoining grammars (TAGs), you can note that they use a similar replacement strategy.

The left-hand side (LHS) matches a non-terminal edge, and the right-hand side (RHS) gives the replacement graph fragment. External nodes are suffexd with a '*' and optionally the number of the external node. By default, external nodes are numbered left to right.

An HRG is a collection of such rules, e.g.:

```
% cat your-hrg
S -> (. :T$ x. :U$ x.);   # Rule 1, defines the start symbol
S -> (. :U$ .);           # Rule 2
U -> (. :V$ (.*1 :d ));   # Rule 3
U -> (. :b (. :W$ .*1));  # Rule 4
V -> (. :b (. :c .*1));   # Rule 5
W -> (. :c (.*1 :d ));    # Rule 6
W -> (. :c .*1);          # Rule 7
T -> (. :a (.*1 :d ));    # Rule 8
T -> (. :a .*1);          # Rule 9
```

In Bolinas the LHS of the first rule in the HRG file is the start symbol. If graph H can be built up through rule applications from grammar G, then we say "G generates H" or "G accepts H" or "H is in the set of graphs represented by G".

We also want to represent sets of weighted graphs. For example, instead of representing English semantic graphs that make sense (black or white), our grammar may assign a soft numerical score to each English semantic graph. We accomplish this by giving weights to individual HRG rules. Here is a weighted HRG rule:

```
U -> (. :V$ (.* :d ));  0.6 # Rule 3
```

Unless specified otherwise, the weights of each rule defaults to 1.0.

The weight of a derivation is a combination of its rule weights (we multiply them), and the weight of a graph is a combination of its derivation weights (we add them).

As you become accustomed to the HRG format, the initial HRGs you type in will likely be ill-formed. Before you do anything with your HRG, you should ask Bolinas whether it is a syntactically well formed HRG:

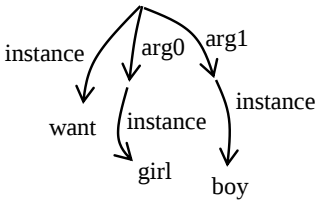
```
% ./bolinas your-hrg
Loaded hypergraph grammar with 9 rules.
```

If your HRG is well-formed, as in this case, Bolinas will report how many rules it has. If it is ill-formed, Bolinas will tell you that instead. While Bolinas may point you to a line and rule number

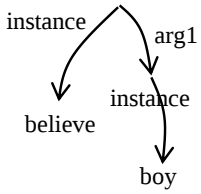
in which it suspects an error it may not give a detailed diagnosis, so you will have to look for problems yourself. Binary search can be helpful – get rid of half your grammar and see if Bolinas still complains, then repeat.

## 2.3 Sample Graph Grammar

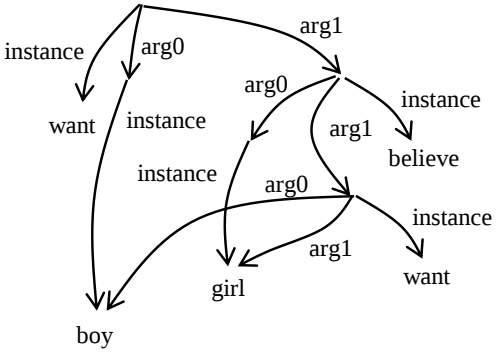
In this section, we'll build an HRG that accepts an infinite set of English semantic graphs in the boy-girl domain. This domain has two predicate types ("believe" and "want"), three relations ("instance", "arg0", and "arg1"), and two entity types ("boy" and "girl"). Here are some members of this graph language:




The girl wants the boy.




The boy is believed.



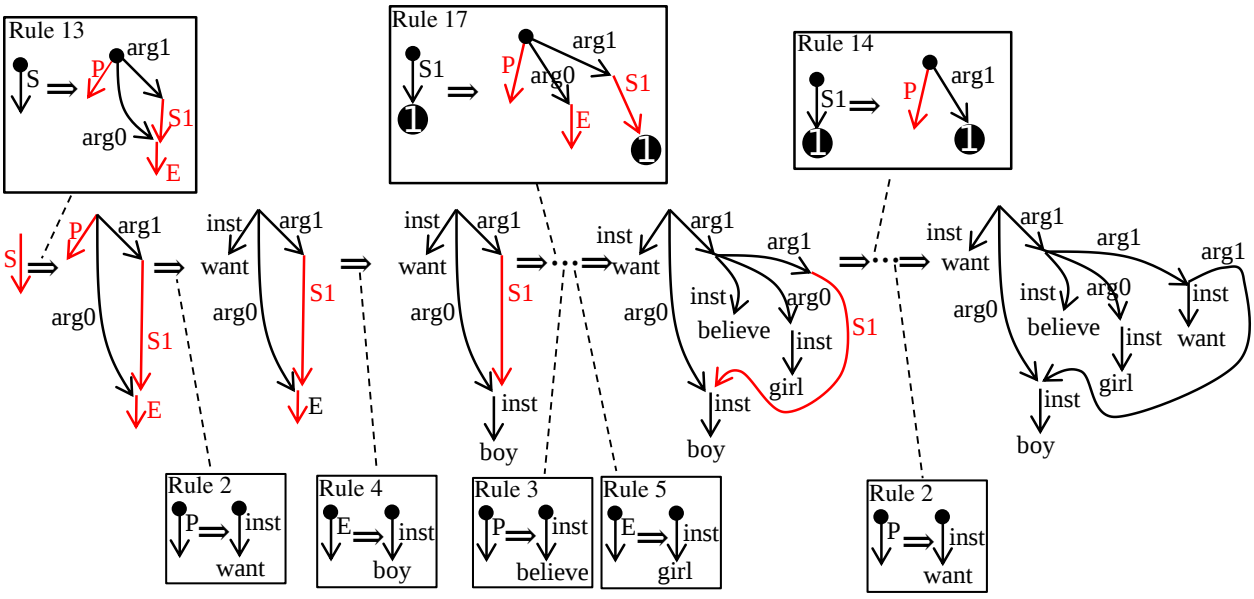

The boy wants the girl to believe he wants her.

We restrict "arg0" to have an animate filler ("boy" or "girl"). It turns out that in this graph language, there are 36 distinct graphs with 5 edges, 112 distinct graphs with 7 edges, 320 distinct graphs with 9 edges, and so on. Here is an HRG that captures graphs in this domain:

```
% cat boy.hrg
Start -> (. :S$);
P     -> (. :instance want);
P     -> (. :instance believe);
E     -> (. :instance boy);
E     -> (. :instance girl);
S     -> (. :P$);
```

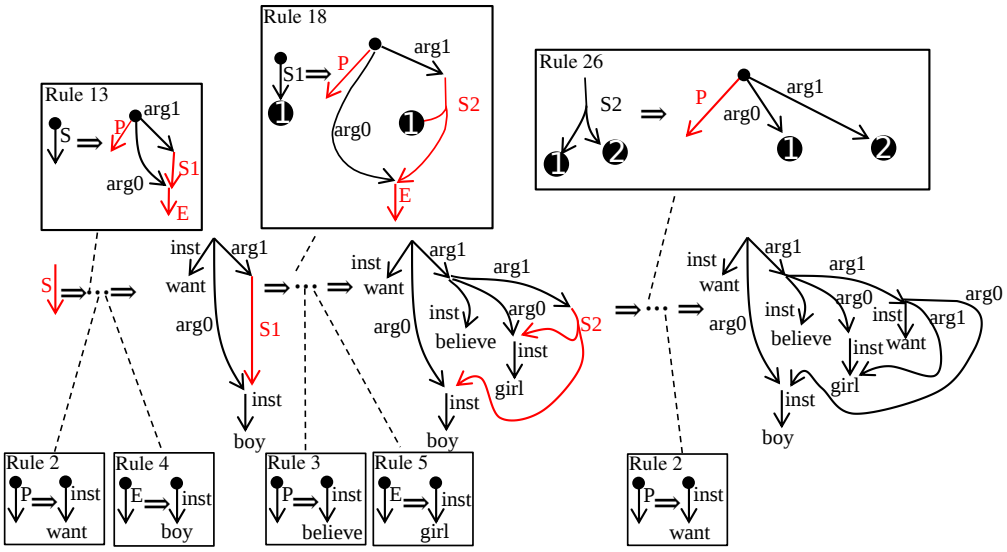
```
S     -> (. :P$ :arg1 (. :E$));
S     -> (. :P$ :arg1 (. :S$));
S     -> (. :P$ :arg0 (. :E$));
S     -> (. :P$ :arg0 (. :E$) :arg1 (. :E$);
S     -> (. :P$ :arg0 (. :E$) :arg1 (. :S$);
S     -> (. :P$ :arg0 (n1. :E$) :arg1 n1.);
S     -> (. :P$ :arg0 (n1. :E$) :arg1 (. :S1$ n1.);
S1    -> (. :P$ :arg1 .*1);
S1    -> (. :P$ :arg1 (. :S1$ .*1));
S1    -> (. :P$ :arg0 (. :E$) :arg1 .*1);
S1    -> (. :P$ :arg0 (. :E$) :arg1 (. :S1$ .*1));
S1    -> (. :P$ :arg0 (n1. :E$) :arg1 (. :S2$ .*1 n1.));
S1    -> (. :P$ :arg0 .*1);
S1    -> (. :P$ :arg0 .*1 :arg1 (. :E$));
S1    -> (. :P$ :arg0 .*1 :arg1 (. :S$));
S1    -> (. :P$ :arg0 n1.*1 :arg1 (. :S1$ n1.));
S1    -> (. :P$ :arg0 n1.*1 :arg1 n1.);
S2    -> (. :P$ :arg0 (. :E$) :arg1 (. :S2$ .*1 .*2));
S2    -> (. :P$ :arg1 (. :S2$ .*1 .*2));
S2    -> (. :P$ :arg0 .*1 :arg1 .*2);
S2    -> (. :P$ :arg0 .*2 :arg1 .*1);
S1    -> (. :P$ :arg0 n1.*1 :arg1 (. :S2$ n1. .*2);
S1    -> (. :P$ :arg0 .*1 :arg1 (. :S1$ .*2);
S1    -> (. :P$ :arg0 n1.*2 :arg1 (. :S1$ .*1 n1.);
S1    -> (. :P$ :arg0 .*2 :arg1 (. :S1$ .*1);

% ./bolinas boy.hrg
Loaded hypergraph grammar with 31 rules.
```

And here is one HRG derivation licensed by the grammar:




The first rule (rule 13) demonstrates how we achieve re-entrancies. We can paraphrase the resulting graph as "an entity E does something involving itself". After applying rule 2 we we get 'an Entity E wants something involving itself". The something is represented by the non-terminal edge labeled "S1". The fourth rule in the derivation (rule 14) replaces the "S1" edge and introduces three new nonterminals ("P", "E", and a new "S1"). If we replace the "P" edge with Rule 3 ("believe") and the "E" edge with Rule 5, the resulting graph can be paraphrased as "the boy wants the girl to believe something about himself". The final rule replaces the "S1" edge with a graph fragment that contains no further non-terminal edges, so the derivation ends with a graph that says "the boy wants the girl to believe that he is wanted".

Note that this derivation only uses edges, not hyperedges. To generate more complex graphs, hyperedges are required. For example:




Here, we start out similarly, but now the second rule (rule 18) introduces a non-terminal hyperedge (labeled "S2") with two tails. After lexicalizing $P$ and $E$ as before (using rules 3 and 5), we can paraphrase the resulting graph as "the boy wants the girl to believe something involving both the girl and himself". Rule 26 then fleshes out the "something".

## 3 Operations on Sets of Graphs

Now that we can type in graphs and HRGs, we can use Bolinas to make inferences for us. Here are several generic operations supported by Bolinas.

**Well-formedness check.** We have already seen this. Just calling Bolinas with a grammar file will verify that the grammar is well-formed and report the number of rules.

```
% bolinas your-hrg
Loaded hypergraph grammar with 9 rules.
```

**Membership check.** Here we check whether a specific graph is accepted by an HRG. If the graph is accepted, Bolinas returns a derivation tree for the graph and the weight of that derivation. Otherwise it prints a blank line.

```
% cat boy.graph
(. :inst want :arg0 (n1. :inst boy) :arg1 (. :inst believe :arg0\
(. :inst girl) :arg1 (. :inst want :arg1 n1.)))
```

```
% bolinas boy.hrg boy.graph
Loaded hypergraph grammar with 31 rules.
1(S$_0(13(P$_2(2) E$_0(4) S1$_1(17(P$_2(3) E$_0(5) S1$_1(14(P$_0(2)))))))) #1.000000
```

The input file can contain many graphs, one per line. Bolinas can also read graphs from the standard input.

```
% bolinas boy.hrg - < boy.graph
Loaded hypergraph grammar with 31 rules.
1(S$_0(13(P$_2(2) E$_0(4) S1$_1(17(P$_2(3) E$_0(5) S1$_1(14(P$_0(2)))))))) #1.000000
```

The output is a derivation tree (in the format understood by Tiburon) that describes which rules are used to replace which nonterminals in the partial derivation. In this example, the derivation starts with rule 1 and replaces nonterminal S with rule 13. It then replaces nonterminal P in rule 13 with rule 2, etc. The indices following the '$' symbol are used to distinguish nonterminals with the same symbol. If no indices are provided in the grammar, Bolinas numbers all nonterminals on the RHS of a rule left to right. The indices are important for synchronous grammars (see below).

**K-best analyses for a graph.** Graphs may have more than one derivation according to the grammar. This command lists the k-best (heighest weighted) derivations for each graph.

```
% cat your-hrg
S -> (. :T$ x. :U$ x.);   0.5
S -> (. :U$ .);           0.5
U -> (. :V$ (.* :d ));    0.6
U -> (. :b (. :W$ .*));   0.4
V -> (. :b (. :c .*));    1.0
W -> (. :c (.* :d ));     0.3
W -> (. :c .*);           0.7
T -> (. :a (.* :d ));     0.2
T -> (. :a .*);           0.8

% cat graph
(.  :a n1.  :b (.  :c (n1.  :d)))

% ./bolinas -k 5 your-hrg graph
Loaded hypergraph grammar with 9 rules.
Found only 3 derivations.
1(T$_1(9) U$_0(3(V$_0(5)))) #0.240000
1(T$_1(9) U$_0(4(W$_0(6)))) #0.048000
1(T$_1(8) U$_0(4(W$_0(7)))) #0.028000
```

**K-best graphs from grammar**. This command lists the k-best (heighest weighted) graphs from the HRG. Bolinas actually searches for the k-best derivations, so the same graph pair may appear multiple times in the k-best list.

Bolinas will normalize weights internally so that all weights for a nonterminal sum up to 1.

```
% ./bolinas -k 10 boy.hrg
Loaded hypergraph grammar with 31 rules.
(. :inst believe )  #0.062500
(. :inst want )     #0.062500
(. :arg1 (. :inst boy ) :inst want )    #0.031250
(. :arg1 (. :inst boy ) :inst believe ) #0.031250
(. :arg1 (. :inst girl ) :inst want )   #0.031250
(. :arg1 (. :inst girl ) :inst believe )        #0.031250
(. :arg0 (x21. :inst boy ) :arg1 x21. :inst want )      #0.031250
(. :arg0 (x21. :inst boy ) :arg1 x21. :inst believe )   #0.031250
(. :arg0 (x21. :inst girl ) :arg1 x21. :inst want )     #0.031250
(. :arg0 (x21. :inst girl ) :arg1 x21. :inst believe )  #0.031250
```

**Derivation forest for a graph.** Bolinas can also output the full derivation forest for each input graph by setting the output type using the "-ot forest" flag.

```
% ./bolinas -ot forest -o out your_hrg.hrg graph
```

Bolinas does not produce any output on the terminal but instead writes each forest into a separate file. "-o out" means that the forest for each graph is written into a file prefixed with "out". The forest for the first input graph will be in "out_1.rtg" etc. While the content of these files may look cryptic, they actually contains a regular tree grammars describing the forest in the format understood by Tiburon. So we can use Tiburon to retrieve the same k-best derivation trees as above:

```
$ tiburon -k 3 out_1.rtg
This is Tiburon, version 0.5.4
1(T$_1(9) U$_0(3(V$_0(5)))) # 0.240000
1(T$_1(9) U$_0(4(W$_0(6)))) # 0.048000
1(T$_1(8) U$_0(4(W$_0(7)))) # 0.028000
```

**Random derivations.** We can also stochastically generated graphs from an HRG. Again, Bolinas will normalize weights internally so that all weights for a nonterminal sum up to 1.

```
% ./bolinas -g 5 boy.hrg
```

```
Loaded hypergraph grammar with 31 rules.
(. :arg1 (. :inst girl ) :inst believe )        #0.031250
(. :arg0 (x21. :inst boy ) :arg1 x21. :inst want )      #0.031250
(. :arg0 (. :inst boy ) :inst want )    #0.031250
(. :arg1 (. :arg0 (. :inst boy ) :arg1 (. :inst boy ) :inst believe )\
 :inst want )      #0.000977
(. :arg0 (. :inst girl ) :arg1 (. :inst girl ) :inst want ) #0.015625
```

**EM training.** If we do not yet have weights on our HRG, we can estimate them from a file of training graphs. EM training will set the weights to maximize the probability of the training graphs:

```
% ./bolinas -t 3 your-hrg training-graphs > new_grammar
Loaded hypergraph grammar with 9 rules.
Iteration 0, LL=-2.173664
Iteration 1, LL=-2.006292
Iteration 2, LL=-2.003363

% cat new_grammar
S -> (_0. :U$_0 x. :T$_1 x.) ; 0.5000000000
S -> (_0. :U$_0 _1.) ; 0.5000000000
U -> (_0. :V$_0 (_1.*0 :d )) ; 0.6608120079
U -> (_0. :b (_1. :W$_0 _2.*0 )) ; 0.3391879921
V -> (_0. :b (_1. :c _2.*0 )) ; 1.0000000000
W -> (_0. :c (_1.*0 :d )) ; 0.5474921894
W -> (_0. :c _1.*0 ) ; 0.4525078106
T -> (_0. :a (_1.*0 :d )) ; 0.3066644770
T -> (_0. :a _1.*0 ) ; 0.6933355230
```

The file of training graphs should contain one graph per line. The number after the "-t" switch tells Bolinas how many iterations of EM to run.

**Intersection of two HRGs.** Sorry! The HRG formalism is not closed under intersection. There are some pairs of HRGs whose intersection is not representable by any HRG, so Bolinas would not know what to return in such cases.

## 4 Graph Transduction

It is certainly fun to capture sets of graphs and do operations on those sets. It is even more fun to transform one graph (input) into another graph (output). We would also like to execute transformations from graphs to trees, trees to graphs, graphs to strings, and strings to graphs.

Such transformations could form the basis of natural language understanding and generation tools, for example.

First, we need a device that concisely captures a possibly-infinite set of graph pairs with weights (g1, g2, 0.3), (g2, g51, 0.5), (g2, g7, 0.7),. Then we can use that device in many ways. For example, we may ask for all output graphs associated with input graph g2. Or the reverse: we can ask for all input graphs associated with output graph g7. So our device is bidirectional.

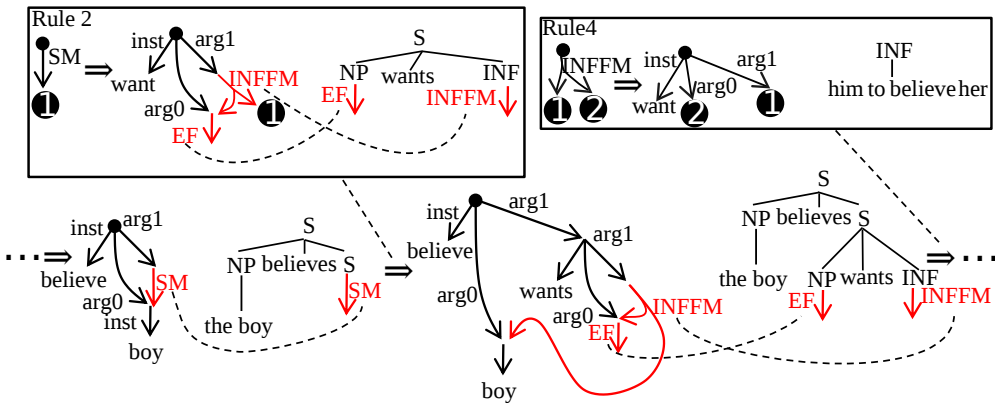
The device we use is a synchronous HRG (SHRG). A SHRG is like a HRG, except that each rule has two RHSs, which are co-ordinated with each other. Bolinas supports tree and string fragments as RHSs [2], as well as graph fragments. This allows us to capture graph-to-tree and graph-to-string transformations.

Here is a SHRG fragment for a graph-to-tree transformation. Since the device is bidirectional, it can also be used for tree-to-graph transformation. When writing bidirectional grammars it is important to make sure that each nonterminal label only appears on hyperedges of the same degree. Otherwise not every tree derivation will have a valid graph derivation. Bolinas does not currently verify that grammars are in this form.

```
% cat graph_to_tree.shrg
Start ->   (. :inst believe :arg0 (n1. :EM$1) :arg1 (. :SM$2 n1.))
         | (S (NP :EM$1) believes (S :SM$2));
SM    ->   (. :inst want :arg0 (n1. :EF$1) :arg1 (. :INFFM$2 n1. .*))
         | (S (NP :EF$1) wants (INF :INFFM$2));
INFFM ->   (. :inst believe :arg0 .*1 :arg1 .*2)
         | (INF to believe him);
INFFM ->   (. :inst believe :arg0 .*2 :arg1 .*1)
         | (INF him to believe her);
INFFM ->   (. :inst believe :arg0 .*2 :arg1 .*1)
         | (INF to be believed by him);
EM    ->   (. :inst boy)
         | (NP the boy);
EF    ->   (. :inst girl)
         | (NP the girl);
```

Here is a partial derivation, using this SHRG:

[2] Trees are interpreted as hypergraphs, i.e. the Bolinas graph format is fully compatible with trees. Sibling child nodes in the tree become the tail of a hyperedge whose root is the parent. Recall that hyperedge tails are ordered, but multiple edges emiting from the same node are not.




## 5 Operations on Sets of Graph Pairs

Now we list generic SHRG operations supported by Bolinas.

**Well-formedness check.** As above. Bolinas will report the format of the grammar. Note that the tree RHS is reported as a hypergraph.

```
% ./bolinas graph_to_tree.shrg
Loaded hypergraph-to-hypergraph grammar with 7 rules.
```

**Transduction.** Here we supply an input graph to a SHRG. Bolinas returns a possible output graph or a blank line if no transduction is possible.

```
% cat graph_tree.graph
(. :inst believe :arg0 (n1. :inst boy) :arg1 (. :inst want :arg0 \
(n2. :inst girl) :arg1 (. :inst believe :arg0 n2. :arg1 n1.)))

% ./bolinas graph_tree.shrg  graph_tree.graph
Loaded hypergraph-to-hypergraph grammar with 7 rules.
(S (NP the boy ) believes (S (NP the girl ) wants (INF to believe him )))       #1.000000
```

Other command line options for HRGs also work for SHRGs. For instance, we can produce k-best output graphs for an input graph with "-k". We can also apply the synchronous grammar in reverse, supply an output graph and get back input graphs (backward application):

```
% cat graph_tree.tree
```

```
(S (NP the boy) believes (S (NP the girl) wants (INF to believe him)))

% bolinas -r graph_tree.shrg graph_tree.tree
Loaded hypergraph-to-hypergraph grammar with 7 rules.
(. :arg0 (x50. :inst boy ) :arg1 (. :arg0 (x20. :inst girl ) \
:arg1 (. :arg0 x20. :arg1 x50. :inst believe ) :inst want ) \
:inst believe )    #1.00000
```

In Tiburon the result of tree tranduction can be represented as a weighted RTG that compactly encodes the set of "answer trees". This supports cascading, i.e. the application of a second tree transducer to the answer RTG. Unfortunately, it is impossible to apply a SHRG to a HRG and to represent the result as a HRG. This would provide a way to compute the intersection of two HRGs, which can't be done in general (see above). Bolinas can only read individual graphs as input and produce k-best graphs or derivation forests as output.

In future versions of Bolinas if may be possible to produce an HRG that compactly encode all answer graphs for an individual input graph.

**Membership check.** Here we check whether a specific pair of graphs is accepted by a SHRG using the "-b" flag. That is, we check whether the SHRG has a derivation that generates the input graph and the output graph simultaneously, in a single derivation. The input file in this case has pairs of graphs in alternating lines. This is sometimes called bitext parsing or synchronous parsing. Note that we also specify "-ot derivation" to print out derivation trees. By default synchronous parsing returns a derived object using the second RHS (a tree in this case).

```
% cat graph_tree.bi
(. :inst believe :arg0 (n1. :inst boy) :arg1 (. :inst want :arg0 \
(n2. :inst girl) :arg1 (. :inst believe :arg0 n2. :arg1 n1.)))
(S (NP the boy) believes (S (NP the girl) wants (INF to believe him)))

% ./bolinas -b -ot derivation graph_tree.shrg  graph_tree.bi
Loaded hypergraph-to-hypergraph grammar with 7 rules.
1(SM$2(2(EF$1(7) INFFM$2(3))) EM$1(6)) #1.000000
```

**EM training.** If we do not yet have weights on our SHRG, we can estimate them from a file of input/output training graphs in the same way as for HRG. EM training will set the weights to maximize the conditional probability of the output graph corpus given the input graph corpus. If the "-b" flag is specified EM will instead maximize the joint probability of inputs and outputs. The file of training graph pairs should contain alternating lines of input and output graphs (or trees, or strings).

**Composition of two SHRGs.** Sorry! The SHRG formalism is not closed under composition. There are some pairs of SHRGs whose composition is not representable by any SHRG. However,

you can still pipeline two SHRGs by passing individual output graphs of the first SHRG along as input to the second SHRG.

# 6 Conclusion

Thanks for taking this tutorial, and please enjoy Bolinas. Feel free to send comments on the software (including bug reports) to `bauer@cs.columbia.edu` and send comments on this tutorial to `knight@isi.edu` and `bauer@cs.columbia.edu`.