LI 569 — Intro to Computational Linguistics
Assignments 3–4: Paths and Parsing

Jason Eisner and Darcey Riley — Summer 2013

(Due date: Thursday, July 25)

General homework policies for this class: Same as on assignments 1–2. In particular, you may work in groups of 2 or 3. When handing in a solution, please clearly indicate all of its authors. Please upload your solution to CTools as a PDF file.

Objectives of this assignment: Let you engage with the material from the 7/5 and 7/11 lectures (slides here) by writing and modifying Dyna code. You will get a better feel for dynamic programming constructions by seeing how they behave on small graphs and grammars.

Please feel free to post questions on Piazza, especially if something isn't working as expected.

Unix hints:

- Remember to `ssh` to one of the machines `a10`, `a11`, `a12`, `a13`, or `a14` to do your work.
- Use the arrow keys to recall and edit previous lines. This is much faster than retyping.
- Remember that the `>` prompt says you are talking to Dyna and the . . . `$` prompt says you are talking to Unix.
- Press Ctrl-C (perhaps repeatedly) to interrupt a program that is taking too long. If that doesn't work, you can use Ctrl-Z to suspend the program.
- It may be handy to open multiple terminal windows—e.g., one to edit your program and another to run Dyna.
- Remember some useful Dyna commands from the tutorial:
  - `help` to find out about the commands,
  - `rules` to list the current rules,
  - `retract_rule` to get rid of an incorrect rule,
  - `sol` to see all items in the current solution,
  - `query` (or `vquery` or `trace`) to see only items that match a given pattern.
- You can download all the files you need for this assignment by typing at the Unix prompt:

```
wget http://cs.jhu.edu/~jason/licl/hw3/hw3.zip  # download zip file
unzip hw3.zip                                   # unpack it to get an hw3 directory
rm hw3.zip                                      # remove the zip file
cd hw3                                          # go into your new hw3 directory
```

1. Pairs of related English words, one pair per line, are listed in the four `.tsv` files:

   `subst5.tsv` Pairs of words that differ by substituting one new letter somewhere

   `swapadj5.tsv` Pairs of words that differ by swapping two adjacent letters

   `swap5.tsv` Pairs of words that differ by swapping any two letters

   `anagram5.tsv` Pairs of words that differ by reordering the letters in any way

   We have limited this to 5-letter lowercase words. A word is never paired with itself, and each pair appears in both orders. The lists are complete with respect to the `/usr/share/dict/words` file, which contains about 99,000 English words of varying lengths.

   You can look at the files, for example using `nano`. If you're curious, Unix's `wc` (word count) utility will tell you how many lines are in each file:

   ```
   wc -l *.tsv
   ```

   Start up Dyna like this, to slurp those 4 files into Dyna and define a graph from them:

   ```
   dyna -i wordgraph.dyna --load 'anagram = tsv("anagram5.tsv")' \
      'subst = tsv("subst5.tsv")' 'swap = tsv("swap5.tsv")' 'swapadj = tsv("swapadj5.tsv")'
   ```

   Look at the file `wordgraph.dyna` to see exactly how the graph is defined. As usual, each edge has a cost. What's new is that this graph has *labeled* edges—so the items look like `edge(&subst,"black","blank")`, with the new first argument telling you that this is a `subst` edge from `subst5.tsv`.

   Here are some basic questions that you should be able to answer by typing appropriate things at the Dyna prompt. Tell us what the answers are, and what you typed to find out.

   (a) What are all the words that are connected to `"black"` by a single edge?

   (b) What do you learn from the following query?

   ```
   query edge(_,"black",X) + edge(_,X,Y)
   ```

   Remark: You can change the way Dyna prints the results. For example, you may prefer `vquery` to `query`. What happens if you change both copies of `X` to `_X`? How could you get Dyna to report the labels on the edges?

   (c) How many edges are in the whole graph? (*Hint:* Use `+=` to count.)

   (d) How many different words are in the graph? Make sure to only count each word once, even if there are many edges attached to it! (*Hint:* Use `:=`, or optionally `|=` or `:-` if you know what those do.)

   (e) Find some "squares" of the form

   $$A \xrightarrow{\text{subst}} B \xrightarrow{\text{swap}} C \xrightarrow{\text{subst}} D \xrightarrow{\text{swap}} A$$

   If you're curious, you can ask Dyna to count how many squares exist.

(f) Define `count_anagrams(U)` (for each `U`) to be the number of anagrams that `U` has, not counting itself.

Now, what is the maximum number of anagrams that any word has? (*Hint:* Use `max=`.) List all words with exactly that many anagrams.

(g) An old puzzle asks: What is the shortest sequence of words that gradually turns `black` into `white`, one small change at a time? Solve this puzzle. The traditional question only allows `subst` moves, but we'll allow you to use any edges in our graph.

As you know, here is a standard program for finding the minimum-cost path from `start` to `V` in a graph.

```
pathto(start) min= 0                       with_key [start].
pathto(V)     min= pathto(U) + edge(U,V) with_key [V | bestpath(U)].
bestpath(V) = $key(pathto(V)).
```

`pathto(V)` is the minimum cost achievable, and `bestpath(V)` is a (reversed) path that actually achieves that cost—ties are broken arbitrarily.

However, you will have to modify this to work with our labeled graph. If possible, make `bestpath(V)` include the words and the edge labels alternately, roughly as shown above in the "square" question.

(h) Can `black` be turned into *every* word in the graph in this way? How do you know?

Note: You may also enjoy turning `trees` into `house`, `bills` into `coins`, `beast` into `child`, `truth` into `error`, `never` into `maybe`, or whatever strikes your fancy. Alas, for complicated reasons, the current version of Dyna is slow to recompute the answer when you change or retract `start`. It's usually quicker to `exit` Dyna and start it again. To start up more quickly, you can put your rules into a file `mincost.dyna` and include it on the command line alongside `wordgraph.dyna`.

(i) Of the words that are reachable from `black`, which one is *farthest* from it, in the sense that the minimum-cost path to it is as long as possible? (You may want to use `with_key`.)

What other words are far away from `"black"`? You can try this query, after defining `maxdistance` to be the distance of the farthest word:

```
vquery [pathto(V), bestpath(V)] for pathto(V) > (maxdistance - 25).
```

What do you notice about these long paths?

Can you change which word is farthest from `"black"` by manipulating the edge costs? (The original costs are defined in `wordgraph.dyna`.)

Now you're warmed up! We'll return to this setting later in the assignment.

2. For this problem, you'll be parsing sentences using the Dyna programs you saw in class, as well as a small toy grammar that can generate some simple English sentences. Have a look at the grammar file, `pirates.gr`. Each rule in the grammar takes one of the following forms, where X, Y, and Z are nonterminals, and W is a word:

$$X \to Y\ Z$$
$$X \to Y$$
$$X \to W$$

In the file `pirates.gr`, these rules are formatted as follows, where P is the rule's probability (to be precise, the probability that a given token of the nonterminal X would choose this as its rewrite rule):

```
P X Y Z
P X Y
P X W
```

Note that all the "interesting" rules in `pirates.gr` are listed in the first 21 lines; the rest of the file is just vocabulary. Also note that the start symbol of this grammar is called `START`.

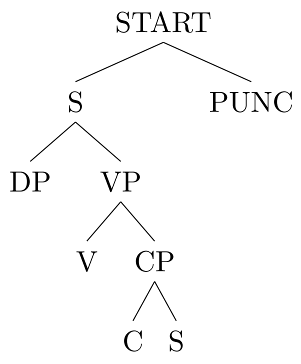
(a) Recall how the grammar can be used to generate sentences (e.g., via the `randsent` program). You begin with the `START` nonterminal, and you roll the `START` die to determine how to expand it. Then, for each child nonterminal, you roll the corresponding die, continuing to expand nonterminals until there are none remanining at the leaves of the tree (because they have all expanded as words from the vocabulary).

According to this process, the probability of a syntax tree (given its root nonterminal) is the product of the probabilities of the rules that were used when generating it:

$$p(T) = \prod_{X \to \alpha \,\in\, T} p(\alpha \mid X)$$

(Here, T is the tree, and the three rule types from above have been abstracted into a single rule type $X \to \alpha$.)

According to the PCFG in `pirates.gr`, what is the probability that the top of the tree will look like this?



(**Note:** You don't have to give your answer as a number; you can just write out the formula to show which quantities need to be multiplied. The dice to expand the DP, the V, etc., have not been rolled yet, so don't multiply their probabilities in.)

(b) Now let's try parsing some sentences using this grammar. Here's some code for the probabilistic CKY algorithm, very similar to what you saw in class:

```
phrase(X,I,J)   max= rewrite(X,W) for word(W,I,J).
phrase(X,I,J)   max= rewrite(X,Y) * phrase(Y,I,J).
phrase(X,I,J)   max= rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).
goal            max= phrase(start,0,sentence_length).
start             := "START".
```

The second rule is new. Why do we need it?

We've put this in `parser.dyna` for you to get started, so you can run `dyna -i parser.dyna` at the Unix prompt. You will add to this file during the exercise. Please include a copy of the finished file with your writeup.

(c) Let's use the pirate grammar to try parsing the following sentence:

```
the raccoon swallowed a cold ice cream sandwich .
```

To do this, you'll need to load in the grammar using the `tsv` loader once you've started Dyna,

```
load grammar_rule = tsv("pirates.gr")
```

or else by starting Dyna like this in the first place:

```
dyna -i parser.dyna --load 'grammar_rule = tsv("pirates.gr")'
```

This will give you items that look like this:

```
grammar_rule(RowNum,P,X,Y,Z) = true
```

But as you can see from the code, the parser needs items that look like this:

```
rewrite(X,Y,Z) = P
```

Edit `parser.dyna` so that the latter items will be automatically defined from the former ones. That is, add Dyna rules that define `rewrite` in terms of `grammar_rule`.

You'll also need to add a `word(W,I,J)` rule for each word in the sentence, and define `sentence_length` to be the end position (`J` position) of the last word.

Once Dyna has the `word` and `rewrite` items it needs, the rules that define `phrase` and `goal` should make the rest of the parse spring into being.

(**Hint:** `tsv` reads in `P` as a string of characters, but you want Dyna to interpret it as a number. The built-in functor `float` can help you with this. If `Str` is the string representation of a number, then the value of `float(Str)` is the number itself.)

(**Remark:** Whoever wrote `pirates.gr` already ensured that for each nonterminal X, the probabilities of all $X \to \cdots$ rules already add up to 1. So you don't have to worry about normalizing them by dividing through by their sum. But you could easily write Dyna rules to do that if you wanted to make life easier for the poor grammar-writer.)

(d) What is the probability of the best parse of the sentence? Also, explore the parse chart a bit using `query`. What are some of the entries that appear in the parse chart, and what are their probabilities? What entries don't appear in the parse chart (meaning that that constituent cannot appear at that position)?

(**Hint:** You won't be able to view the parse itself until you implement backpointers in problem 5, but the following command might be helpful in the meantime: `trace goal`.)

(e) Try writing a sentence that gets a higher-probability parse than the raccoon sentence from problem 2c. What is the highest-probability sentence you can find? What principles did you use when constructing it?

(Your sentence should be the same length as our sentence. However, you may also enjoy playing with the number of words in the sentence to see how that changes the probability of the parse.)

(**Hint:** If you get tired of typing in the `word(W,I,J)` rules, then you can read the sentences in from a file. Suppose each line of your file contains one sentence. Then you can use the `matrix` loader, as you did with the Brown corpus in the tutorial:

```
load token = matrix("sentences.txt",astype=str)
```

This will give you items of the form `token(S,I)`, where `S` is the sentence number, and `I` is the position of the word in the sentence. Then you can use these to define the `word(W,I,J)` items. One design is that if you set `sentnum := 5`, then the `word` items should define the words from the fifth sentence, and changing `sentnum` will give you a different sentence. Alternatively, you can change the program so that every word and phrase and goal item has an extra argument, e.g., `word(W,I,J,Sentence)` for a word from position `I` to `J` in sentence `Sentence`.

3. The code you've been working with just finds the probability of the *best* parse for a sentence. If you want to sum up the probability of *all* parses for a sentence, you can change `max=` to `+=`. This is called the "inside algorithm."

   (a) Try running this algorithm on the raccoon sentence from problem 2c. Why is the probability different from the one you got before? Be specific about which chart cells (`phrase` items) have different values, and why.

   (Again, you might find `trace goal` to be useful. To compare the `max=` version and one with the `+=` version, you may want to run two copies of Dyna in side-by-side terminal windows.)

   (b) In problem 2e, you looked for a sentence that gave you the highest value for `goal`. Try again now that you're using the inside algorithm. How has your sentence-constructing strategy changed?

4. It's easy to modify the inside algorithm so that it *counts* the number of parses, by giving each rewrite rule in the grammar a weight of 1.

   (a) Implement this. What did you have to change about the program? How many parses are there of the sentence you wrote for 2e?

   (b) Why does this program correctly compute the number of parses? What are the values of the various `phrase` items?

   (c) Consider the following grammar, which generates ambiguous noun-noun compounds like the ones you saw in "One, Two, Tree" (but for a language whose only noun is spelled "`noun`").

$$\text{Noun} \rightarrow \text{Noun Noun}$$
$$\text{Noun} \rightarrow \text{noun}$$

   Using this grammar and your modified parser, parse inputs of various lengths (e.g. "`noun noun noun noun noun`"). This should compute the correct number of bracketings (called the *Catalan number*), as given in the solution" to "One, Two, Tree". What is $f(8)$? $f(9)$?

Note: If you prefer to construct your input of length `n` automatically, you can write this:

```
word("noun",I,I+1) = true for I in range(0,n).
```

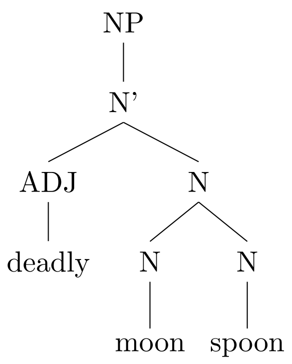
(d) Running a parser seems like overkill compared to the relatively simple formula in the "One, Two, Tree" solution. What would that formula look like in Dyna?

Can you regard the Catalan number program as a simplified version of the parsing program—one that is specialized to the noun-noun compound grammar? (How do the `f` items in the former correspond to certain items in the latter?)

(e) If you like, you can also use the modified parser to count the parses of adjective-noun sequences from later in the "One, Two, Tree" problem. If you are having fun and feeling even more ambitious, you could even implement a parser for "Twodee" (perhaps using the `matrix` loader to read the two-dimensional sentences).

5. The CKY algorithm from (1b) tells you the *probability* of the best parse, but it doesn't tell you what the parse actually *is*. In this problem, you'll augment the algorithm to keep track of the best parse for each phrase.

(a) Before you can implement backpointers, you'll need a way to represent parse trees in Dyna. Consider the noun phrase "deadly moon spoon". According to our grammar, its parse should look like this:



We'll encode this structure in Dyna using the following notation:

```
t("NP",t("N'",t("ADJ","deadly"),t("N",t("N","moon"),t("N","spoon"))))
```

As you can see, each constituent `X` has a parse of the form `t(X,TY,TZ)`, `t(X,TY)`, or `t(X,W)`, where `TY` and `TZ` are the children of this constituent (also written in this tree notation), and `W` is a word.

But what is `t(X,TY,TZ)`, exactly? It looks similar to `phrase(X,I,J)`, but it doesn't have a value, and even if it did, we wouldn't care. The answer is that both are *terms*, but only `phrase(X,I,J)` is the name of an *item* (something with a defined value). We are not using the term `t(X,TY,TZ)` as an item name, but rather for its own sake—as a **data structure** to represent the parse tree.

In most places in a Dyna program, writing
`t("NP",t("N'",t("ADJ","deadly"),t("N",t("N","moon"),t("N","spoon"))))`
will cause Dyna to try to look up the values of various `t(...)` items, which don't exist:

```
> mytree = t("NP",t("N'",t("ADJ","deadly"),t("N",t("N","moon"),t("N","spoon")))).
> query mytree
No results.
```

So instead, you'll need to explicitly tell Dyna not to try to *evaluate* these terms (look up their value), but just use the literal terms. Recall from class that you can can do this with the `&` operator. Here's a simple example of how this works. Suppose you define two items, `a` and `b`, like this:

```
a = 1.
b = 1.
```

Then you can define the following four items, each with a different value:

```
e(a,b)   = 2.
e(&a,b)  = 3.
e(a,&b)  = 4.
e(&a,&b) = 5.
```

Since Dyna evaluates `a` and `b` when they aren't preceded by `&`, `e(a,b)` is equivalent to `e(1,1)`, which is equivalent to `e(b,a)`. But `e(&a,&b)` is not equivalent to `e(&b,&a)`.

Using the `&`, correct the rule for `mytree` from above.

(b) Modify `parser.dyna` to find the actual highest-probability parse, using `max=` and `with_key` as in problem 1g.

(c) Use this code to find the parse tree for the high-probability sentence you wrote for problem 2e.

(d) In order to find that sentence from problem 2e, you probably had to stare at the rules of the grammar, or try out a bunch of different sentences until you found one with a high-probability parse.

All of that turns out to be completely unnecessary, because you can use the parsing algorithm to *automatically* find the highest-probability sentence of the language. Instead of specifying a *particular* input with items like

```
word("the",0,1).
```

you can specify an *ambiguous* input using items like

```
word(W,0,1) :- true for in_vocab(W).
```

This is saying that *any* first word is possible, not just `the`. This way, when the parser is looking for the highest-probability tree, it is free to *nondeterministically choose* whichever words maximize the probability!

What is the best sentence of length 9, and what is its parse tree? What about length 8? Length 10? As in problem 4c, you may want to use the `range` construct to get a sentence of length `n`.

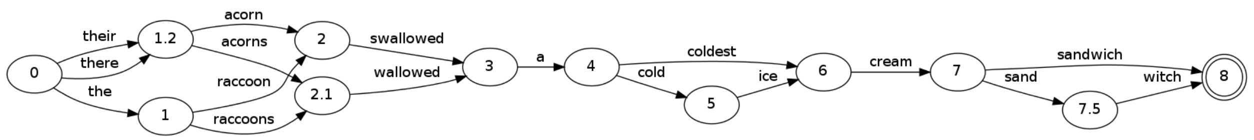
Note: You should define `in_vocab` to consist of all symbols that appear on the right-hand side of a (unary) rewrite rule but never on the left-hand side

```
in_vocab(W) := true for _ is rewrite(_,W).
in_vocab(W) := false for _ is rewrite(W,_).    % overrides previous line
in_vocab(W) := false for _ is rewrite(W,_,_).
```

(e) Usually you don't want to pick the highest-probability sentence in the language, but rather the highest-probability sentence that matches some ambiguous input such as a speech signal. Did the sentence end with `word("sandwich",7,8)` or with `word("sand",7,7.5)` followed by `word("witch",7.5,8)`?[1] If the speech recognizer wasn't sure, it can just make all of those `word` items true, and let the parser sort out which ones are more plausible, in context, given the grammar.

(**Remark:** Note that `ice cream sand witch` has more parses than `ice cream sandwich`, thanks to the Catalan numbers, so it might do well under the inside algorithm with `+=`. However, `max=` can't see this. It is only looking for the single best parse, not the single best sentence.)

In general, the input to the parser can be a *speech lattice* that was produced by a speech recognizer. A speech lattice is really just an FSA, where the states might be labeled with time positions relative to the start of the acoustic utterance:



To assert all the `word` edges in this lattice to be simply `true` is to ignore useful information that comes back from the speech recognizer. The grammar tells you whether `N` is likely to rewrite as `acorns`, but the speech recognizer tells you whether `acorns` is likely to rewrite as the stretch of acoustic signal from time 1.2 to time 2.1. In other words, each edge in the lattice has a number attached to it. How could you modify your parser to take these numbers into account?

6. Now let's go back to problem 1 and do a whimsical exploration of unsupervised learning. Following the "Orwellspeak" problem, imagine that you are a press officer for a totalitarian government. Mysterious changes often happen in the printing office. For example, once you wrote `"black"` but the government printed `"white"`.

   Your best guess about what happened in the printing office is the minimum-cost path of edits that you reconstructed in problem 1g. However, you are not sure. There are other paths from `black` to `white` and there's no telling for sure which one is right!

   You are trying to figure out what goes on in the printing office, so that you can guess how your words will be corrupted in future. Fortunately you have several examples to learn from. Once `"black"` turned to `"white"`. On other occasions, `"right"` turned to `"wrong"` and `"fight"` turned to `"peace"`.

   On more typical occasions, however, `"parse"` turned to `"pares"`, `"truth"` into `"tripe"`, and `"enemy"` into `"enema"`. It was examples like those—plus your informants inside the printing office—that gave you the idea to model the printing office as making *orthographic* rather than semantic edits. That is why you constructed `wordgraph.dyna`. You are blithely confident

[1]The Dyna program doesn't actually care what you call the positions in the sentence, and 7.5 is a perfectly good name for this moment in time in the acoustic signal.

that the graph edges are correct; you're just not so sure about their costs and would like to improve those.

(Having a good model of the printing office could be useful. For example, you could use it to work backwards—what should you write if you want the government to print `"truth"`? That's just a `max=` problem once you have the right model.)

(a) This setting is whimsical, but it is related to real unsupervised learning problems in CL and NLP. Give an example of a "real" case where one linguistic form turns into another by a sequence of edits, or is hypothesized to do so.

Is your example one of production, comprehension, or something else? What would "working backwards" correspond to?

(b) Let's build a model where the low-cost paths have high probability. We can use the usual interpretation of cost as negative log-probability.

For example, consider the path

$$\texttt{print} \xrightarrow{\text{subst}} \texttt{paint} \xrightarrow{\text{anagram}} \texttt{inapt}$$

with total cost $10 + 6 = 16$. We would like its probability to be proportional to $e^{-16} = e^{-10+-6} = e^{-10} \cdot e^{-6}$. We can arrange that by defining a *potential* for each edge:

```
edge_potential(L,U,V) = exp(-edge(L,U,V)).
```

Then the probability of a path is proportional to the product of its edge potentials—in this case, e^{-10} and e^{-6}. (In machine learning, "potentials" are the things you multiply together to get something proportional to the probability.)

Since e^{-1} is more than 8000 bigger than e^{-10} (which is very small), a `swapadj` arc is over 8000 times as likely as a `subst` arc. That might be a little too strong. By dividing the costs by 3, we can reduce this discrepancy:

```
edge_potential(L,U,V) = exp(-edge(L,U,V) / temperature).
temperature := 3.0.   % should be positive
```

Now `swapadj` is only about 20 times as likely as `subst`.[2]

Add the above two lines to `wordgraph.dyna`. Look at the new graph that is defined by the `edge_potential` items (which are identical to the `edge` items except that they have different values). What are the potentials on the edges from problem 1a?

(c) Now, you will need to write the forward-backward algorithm. Properly speaking, the name "forward-backward" is used for HMMs. On this more general graph, we call it the *sum-product algorithm* (because it sums up products of potentials).

You can start with the Dyna code from class to compute the α and β scores. Note that this is similar to the code in problem 1g, except that

- You're multiplying edge potentials instead of adding edge costs.
- You're using `+=` instead of `min=`.
- You're using the backward algorithm to compute beta values, instead of following backpointers (so you don't need `with_key`).

[2] 20 is the cube root of 8000; this is because we divided by 3.0. Dividing by 2.0 would have given us the square root. In general, a temperature > 1 will make different paths more similar in probability, while a temperature < 1 will exaggerate the differences betwen paths. As the temperature approaches 0, the maximum-probability path (if unique) becomes far more probable than any other, so `+=` becomes indistinguishable from `max=`.

And to repeat from class, here is how you get the vertex and edge counts, conditioned on the fact that you know that there was *some* path from `start` to `end`. According to your current model, the relative probabilities of those paths are proportional to their potentials.[3]

```
z = alpha(end).
count_posterior(V)     = alpha(V) * beta(V) / z.
count_posterior(L,U,V) = alpha(U) * edge_potential(L,U,V) * beta(V) / z.
```

although the current version of Dyna will be considerably faster (for complicated reasons) if you help it out by breaking the final line into two steps:

```
alpha(L,U,V) = alpha(U) * edge_potential(L,U,V).
count_posterior(L,U,V) = alpha(L,U,V) * beta(V) / z.
```

Unfortunately, because the graph has cycles, the current Dyna has trouble converging to a solution for the `alpha` and `beta` values when you define them with `+=`. I suggest using `max=` instead. This gives a popular approximation to the *sum-product* algorithm, known as the *max-product* algorithm. It is a good approximation because typically, each of these sums involves one larger probability and a bunch of much smaller probabilities (especially at low `temperature`), so the sum is only a few percent larger than the maximum.[4]

Thus, put your computation in a file called `maxproduct.dyna`.

Now set `start := "black".` and `end := "white".`. Having observed that the printing office turned `"black"` into `"white"`, we can use the posterior counts of the edges to estimate what may have happened inside the printing office.

So, what are the estimated total counts of the different edge types (subst, swapadj, swap, or anagram) that the printers used to change `"black"` into `"white"`, according to the current model?[5] Use Dyna to add them up.

How about the counts of edge types used to change `"parse"` into `"pares"`, or `"enemy"` to `"enema"`? Why are these different?

[3]By the potential of a path, we mean the product of its edges' potentials. We know we took exactly one path that was consistent with the evidence. For example, we took either path A with potential 0.0004 or path B with potential 0.0001. Dividing these by the total $\texttt{z} = 0.0004 + 0.0001 = 0.0005$ gives posterior probabilities of 4/5 and 1/5 for paths A and B respectively.

Thus, an edge that appears only on path B will have posterior count of 1/5, while an edge that appears on both paths will have posterior count of $4/5 + 1/5 = 1$. When counting a vertex or edge, we count a path multiple times if it goes through the given vertex or edge multiple times. For example, an edge that appears once on path A and twice on path B will have a posterior count of $4/5 + 1/5 + 1/5 = 1.2$, which is the "expected" number of times that the unknown true path passed through the edge. (This is a kind of average: the true number is 1 or 2, with 1 being more likely.)

The Dyna formula obtains this posterior count as $(0.0004 + 0.0001 + 0.0001)/0.0005 = 1.2$. The numerator—computed by the sum-product algorithm—is the total *potential* of all paths from `start` to `end` that pass through the edge. Dividing by `z` at the end gives the total *probability* of those paths, $0.0004/0.0005 + 0.0001/0.0005 + 0.0001/0.0005$.

[4]Under the max-product approximation, the numerator of the posterior count is actually the *maximum* potential of any *single* path to use that edge (or vertex). Hopefully that is close to the *total* potential of *all* such paths, as explained above. The denominator `z` is also an approximation: the single best path from `start` to `end` instead of the total.

[5]And if you read footnote 2, what would you expect to happen to these counts if you use a `temperature` close to 0, such as 0.1? Try it: are you right?

(d) If you accumulated these fractional counts over a lot of pairs that you observed from the printing office, you might discover that they differ from the counts that the model would have predicted when unconstrained by the evidence of such pairs. So, you could adjust the model parameters to bring the expected counts in line with the observed counts.

This adjustment is an example of a step in the Expectation-Maximization (EM) learning algorithm, and our model is actually a conditional log-linear model that defines conditional probability distribution of the form $p(\text{outcome} \mid \text{context})$.

In this question, you'll tie practically the whole course together by figuring out how the concepts you've learned relate to specific things in this setting. Think carefully!

i. How many parameters does the model have?

ii. Which part of the learning strategy is the expectation step?

iii. Which part is the maximization step?

iv. What are the *contexts* in our log-linear model?

v. What are the *possible outcomes* in each context? Remember, you're defining a probability distribution over those outcomes.

vi. What are the features of a (context, outcome) pair? (*Hint*: The model has 4 features.)

vii. How did you initially define the weights of the features?

viii. To estimate the parameters of a log-linear model, you can do gradient ascent on the regularized log-likelihood. The gradient involves a difference of *observed* and *expected* counts. Which ones did you already compute using Dyna? How would you compute the others using Dyna?

(e) **Extra credit**: Our probability model above was a model of a choice *among completely formed paths*. However, perhaps what really happens in the printing office is that the edges are chosen one at a time.

So perhaps we'd do better by modeling the sequence of choices *among edges*—one choice per step. Under this model, the first printing officer to receive your text `"black"` chooses a random edge from `black`—that is, one of the edges that you listed in problem 1a. Perhaps she chooses to follow the edge `black` $\xrightarrow{\text{subst}}$ `slack`. Thus the second officer receives `slack` and chooses a random edge from `slack`: perhaps `slack` $\xrightarrow{\text{anagram}}$ `calks`. The third officer receives `calks` and decides to stop and print `"calks"`.

In effect, we're using a single noisy-channel model that mutates the word by a single change, but the output of the noisy channel keeps getting fed back through the same noisy channel again. (Phonologists call this a "cyclic rule.")

Notice that if an officer receives word W and there are k outgoing edges from W to other words, then the officer has $k+1$ possible choices—namely, the k edges as well as the STOP action. The probabilities of these $k+1$ choices should sum to 1, as if there is a special $(k+1)$-sided die sitting at W and the officer rolls it to make a choice. You can define these $k+1$ probabilities by a conditional log-linear model of $p(\texttt{choice} \mid W)$.

Your job is to redefine the edge potentials in Dyna so that they implement this model. You can use the same features as before. Questions:

i. What are your new Dyna rules?

ii. How does this change the estimated counts on the `"black"` $\rightarrow$ `"white"`, `"parse"` $\rightarrow$ `"pares"`, and `"enemy"` $\rightarrow$ `"enema"` paths (compared to question 6c)?

iii. How does this change your procedure for reestimating the feature weights from the estimated counts (compared to question 6d)?

iv. Can you adjust the stopping probability by changing one or more feature weights? What effect does that have on the estimated counts?

v. Is the probability of `"enemy"` $\rightarrow$ `"enema"` the same as the probability of `"enema"` $\rightarrow$ `"enemy"`? Were these probabilities equal under the old model? Discuss.

vi. In previous questions, you were working with a "globally normalized" model in which entire paths compete for probability. Each path gets a fraction of the overall probability mass, proportionate to its overall score. But in this extra credit question, you are working with a "locally normalized" model in which the edges at each W compete for probability.

    Can you construct an example where word W is more likely to be printed as word A under the globally normalized model, but more likely to be printed as word B under the locally normalized model?

vii. Which approach (global or local) is more like the CFGs and HMMs we studied in class?

viii. Could CFGs and HMMs be converted to use the *other* approach (global vs. local)? How would that change parsing and part-of-speech tagging?

Remarks about the global/local difference: Under the new locally normalized model, the edge `enemy` $\xrightarrow{\text{subst}}$ `enema` is very likely despite being a high-cost substitution edge. That's because it has no local competition: it's the *only* edge from `enemy` (that is, $k = 1$)! The officer who received your word `"enemy"` had no other choices, except to stop immediately and print `"enemy"`.

By the way, the relevant part of the graph is actually rather small:

$$\texttt{enemy} \overset{\text{subst}}{\leftrightarrows} \texttt{enema} \overset{\text{subst}}{\leftrightarrows} \texttt{edema}$$

According to the locally normalized model, the printing office will randomly wander left and right along this graph until it chooses to stop and print what it has.