1.

(a)

109594 words.

(b)

27 distinct characters (the English alphabet + apostrophe).

(c)

935216 character occurrences.

2.

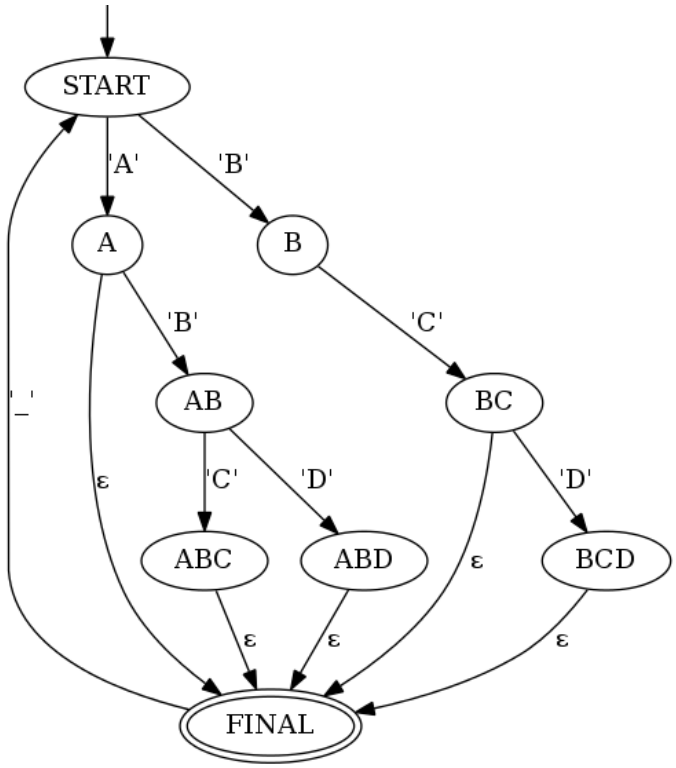
(a)

250322 states, 359915 transitions.

(b)

The structure of the FSA is like a prefix tree, except that all the leaves further transit to a final state with ϵ. The final state transit back to the start state (i.e. the root of the `tree') with `_'.

For example, if the vocabulary consists of {A, ABC, ABD, BC, BCD}, the FSA would be:



(c)

In order for the output to be clearer, carmel option is set to `-slibIWQ' instead of `-slib'. Standard output for the `strings`(first 5 lines, with spaces omitted for clarity):

```
LIST*e*_THE*e*_FLIGHTS*e*_FROM*e*_BALTIMORE*e*_TO*e*_SEATTLE*e*_THAT*e*_STOP*e*_IN*e*_MINNEAPOLIS*e*
DOES*e*_THIS*e*_FLIGHT*e*_SERVE*e*_DINNER*e*
I*e*_NEED*e*_A*e*_FLIGHT*e*_TO*e*_SEATTLE*e*_LEAVING*e*_FROM*e*_BALTIMORE*e*_MAKING*e*_A*e*_STOP*e*_IN*e*_MINNEAPOLIS*e*
I*e*_NEED*e*_TO*e*_HAVE*e*_DINNER*e*_SERVED*e*
I*e*_HAVE*e*_TWO*e*_FRIENDS*e*_THAT*e*_WOULD*e*_LIKE*e*_TO*e*_VISIT*e*_ME*e*_ON*e*_WEDNESDAY*e*_HERE*e*_IN*e*_WASHINGTON*e*_D*e*_C*e*
```

`strings.bad` doesn't write anything to standard output. Here's what it writes to standard error (first 6 lines, with spaces and quotes omitted for clarity):

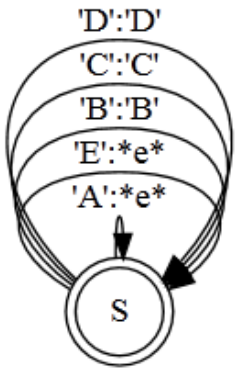
```
Input line 1: I_WUNT_TO_LEEVE_MONDAY_MORNING
    (0 states / 0 arcs)
Empty or invalid result of composition with transducer "english.fsa".
Input line 2: NOW_I_NEAD_A_FLIGHT_ON_TOOSDAY_FROM_PHEENIX_TO_DETROIT
    (0 states / 0 arcs)
Empty or invalid result of composition with transducer "english.fsa".
```

3.

(a)

The FST has only 1 state, as both its start state and final state. The FST accepts any character as its input, output the input character when it's not a vowel, or ϵ otherwise.

For example, if the alphabet only consists of {A, B, C}, then the FST would be:



(b)

First 5 lines of `strings.novowels` (with spaces and quotes omitted for clarity):

```
LST_TH_FLGHTS_FRM_BLTMR_T_STTL_THT_STP_N_MNNPLS
DS_THS_FLGHT_SRV_DNNR
```

```
_ND__FLGHT_T_STTL_LVNG_FRM_BLTMR_MKNG__STP_N_MNNPLS
_ND_T_HV_DNNR_SRVD
_HV_TW_FRNDS_THT_WLD_LK_T_VST_M_N_WDNSDY_HR_N_WSHNGTN_D_C
```

(c)

The command only returns the top 10, but there are infinite number of such strings. The output for each vowel input is a ϵ, therefore we can put arbitrary number of arbitrary places when we try to restore the word.

4.

(a)

The first 5 lines of `strings.restored` (with spaces and quotes omitted for clarity):

```
LST_TH_FLGHTS_FRM_BLTMR_T_STTL_THT_STP_N_MNNPLS
DS_THS_FLGHT_SRV_DNNR
_ND__FLGHT_T_STTL_LVNG_FRM_BLTMR_MKNG__STP_N_MNNPLS
_ND_T_HV_DNNR_SRVD
_HV_TW_FRNDS_THT_WLD_LK_T_VST_M_N_WDNSDY_HR_N_WSHNGTN_D_C
```

(b)

Accuracy: 0.012865497076

(c)

The FST has no idea when it should restore a vowel and which vowel to pick; actually it doesn't have any motivation to restore any vowel, nor does it attempt to.

5.

(a)

Prepend the `english` FSA to the `remove-vowels` FST. This composition results in a FST that accepts inputs that conform to the vocab file, and gives output with vowels removed:

```
cat strings.novowels | carmel -sribWEIk 1 english.fsa remove-vowels.fst
```

(b)

Accuracy: 0.394152046784

(c)

This is a big improvement, but still not satisfactory. Possible reasons:

i. The model doesn't know the frequency of the words in English, consequently produces very obscure words that are very likely to be wrong.
ii. The model doesn't understand the context of the words, consequently producing words that doesn't make sense when put together.

6.

I built WFSAs that assign weights to restored strings according to unigram and bigram statistics. I achieved an accuracy of 0.883 using unigram, 0.978 using bigram.

(1) Preparation

Before explaining how these WFSAs are built, two problems need to be addressed.

Firstly, we need to decide how the statistics should be acquired. I could have simply used the `strings` file, and get a high accuracy by overfitting it, but this would not make much sense and might be deemed as cheating. The sentences in `strings` are airline ticket queries, which are very domain-specific, so it wouldn't be good enough to acquire the statics from some random English texts, either.

I Googled a few sentences in `strings`, and found some similar corpus[1] that can be used as "training data" after a bit of modification. I also added the names of some major U.S. cities and all U.S. states[2] to the corpus. To avoid "cheating", all sentences that appears in `strings` are removed from the training data.

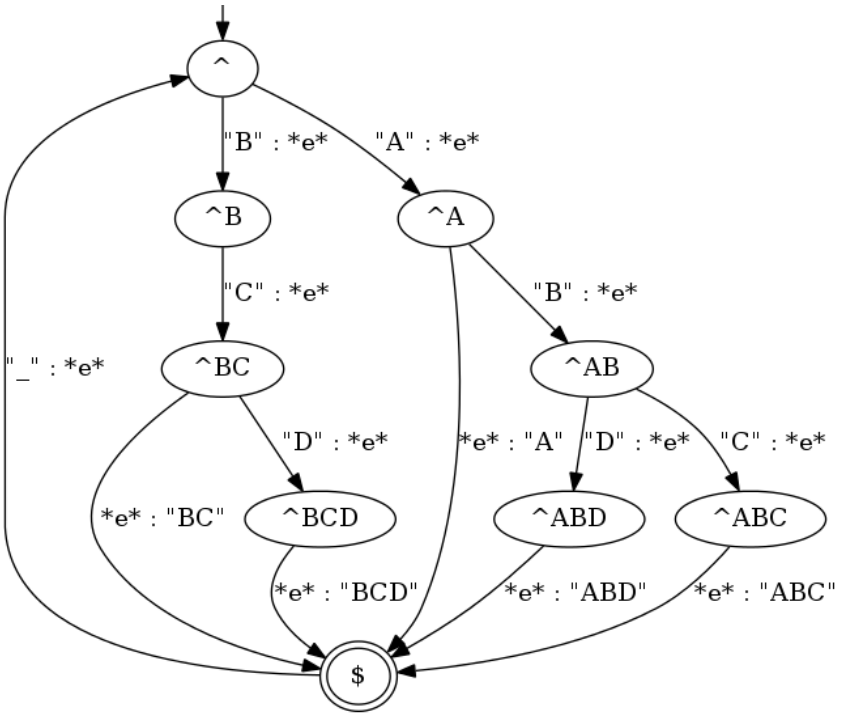
The second problem is the difficulty of encoding word unigram / bigram information in a FSA / FST with letter transitions. To bypass this issue, I built two FSTs to enable conversion between word transitions and letter transitions (e.g. `"APPLE" <-> "A" "P" "P" "L" "E"`).

The FST that translates letter sequences to words in the vocabulary is based on the FSA I built for 2(a). The original transitions are inputs; each letter input corresponds to an ϵ output, while each ϵ transition to final state corresponds to an output of the completed word.

For example, if the vocabulary consists of {A, ABC, ABD, BC, BCD}, the FST would be:

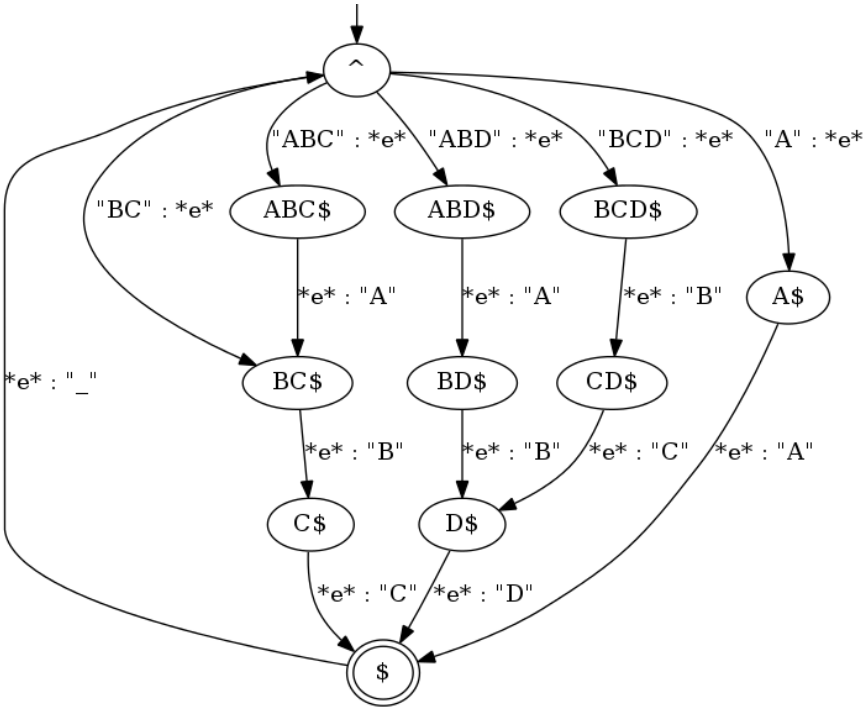
[1] http://nlp.stanford.edu/courses/cs224n/2000/jpaul2/AppendixB.txt
http://courses.washington.edu/ling571/ling571_WIN2011/hw/sents.test
http://archive.lib.msu.edu/RECYCLER/S-1-5-21-1454471165-57989841-1417001333-1387/Df3/atis/atis3

[2] http://en.wikipedia.org/wiki/List_of_United_States_cities_by_population
http://en.wikipedia.org/wiki/U.S._state



Theoretically speaking, this FST can also be used symmetrically to convert words back to letter sequences. However, this would cost a huge amount of time and memory during actual computation, so I built another FST to do the word-to-letters conversion.

This FST has a similar Trie-like structure, but is built for each word from right to left. In other words, it is like a prefix tree for suffixes. For example, for the same vocabulary, the FST looks like:



I call the first FST `letters2word.fst`, the second `word2letters.fst`. For convenience, I also created an inverted version for each of them using `carmel -v`, named `letters2word-invert.fst` and `word2letters-invert.fst` respectively.

Now, if we had implemented an WFSA named `wfsa` that assigns weights to sentences, we could chain the WFSA and FSTs together to restore strings:

```
cat strings.novowels | carmel -sribWEIk 1 word2letters-invert.fst wfsa letters2word-invert.fst remove-vowels.fst
```

Next I will explain how I built WFSAs that captures unigram and bigram information.

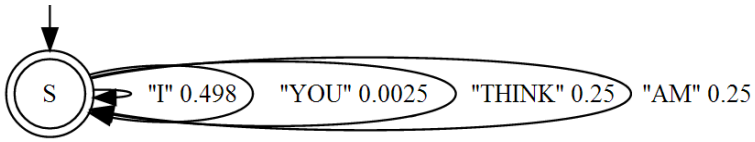
(2) Unigram

The WFSA for unigram has only one state, being both its start and final state. Each word w_0 in `vocab` corresponds to a transition from this state to itself, with weight:

$$P(w) = \frac{\#w_0 + \lambda_1}{\sum_{w \in V}(\#w + \lambda_1)}$$

where $\#w$ is the frequency of word w in training corpus, V is the set of word in `vocab`, λ_1 is a smoothing term, which we set to 0.01. Note that words that appeared in training corpus but not in `vocab` are ignored.

For example, if the vocabulary consists of {I, YOU, THINK, AM}, the training corpus consists of one sentence "I THINK THEREFORE I AM", the generated unigram would be:



(3) Bigram

The problem with bigram is that with a vocabulary consists of more than 100,000 words, it becomes quite infeasible to put in a WFSA all the transition probabilities of each possible word pair. Instead, we only worry about the consecutive word pairs that has appeared in the training corpus, while using the unigram weight as a fallback.

The WFSA for bigram has a state "$" as both its start state and final state. Each word w in the vocabulary also corresponds to a state, each of which has an transition w from "$" with weight $P(w)$ as defined earlier, and an transition ϵ back to "$" with weight

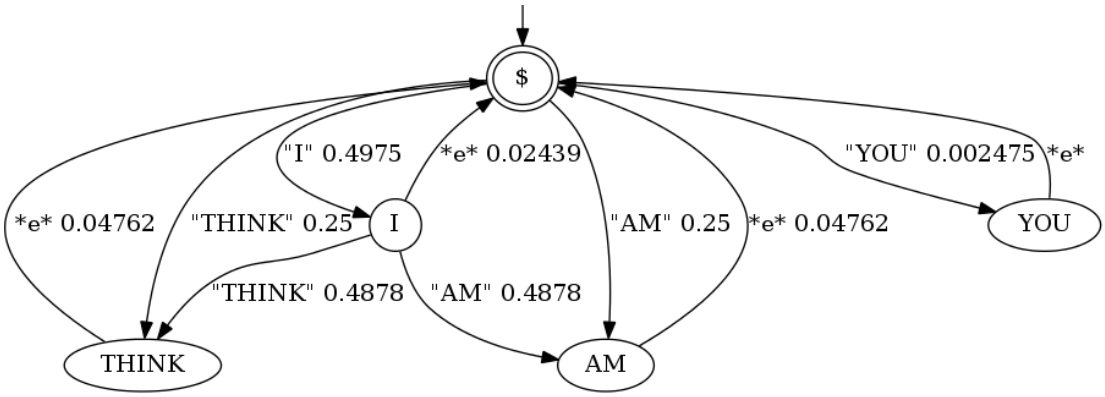
$$P(\$|w) = \frac{\lambda_2}{\#w + \lambda_2}$$

where λ_2 is another smoothing term, which we set to 0.05. For each consecutive word pair (w_l, w_r), there's an transition w_r from w_l to w_r with weight

$$P(w_r|w_l) = \frac{\#(w_l, w_r)}{\#w_l + \lambda_2}$$

where $\#(w_l, w_r)$ is the frequency of such consecutive pair in training corpus. Note that $P(*|w)$ doesn't always add up to 1, but this doesn't seem to be a problem for carmel.

For example, for the same vocabulary and training corpus, the WFSA would look like:



Written in carmel format:

```
$
($ (I "I" 4.975e-01))
($ (YOU "YOU" 2.475e-03))
($ (THINK "THINK" 2.500e-01))
($ (AM "AM" 2.500e-01))
(I (AM "AM" 4.878e-01))
(I (THINK "THINK" 4.878e-01))
(I ($ *e* 2.439e-02))
(YOU ($ *e* 1.000e+00))
(THINK ($ *e* 4.762e-02))
(AM ($ *e* 4.762e-02))
```