Report of Automatic Speech Recognition Assignment

s2226904, s2029927

Abstract

Based on the weighted finite state transducers (WFST), this report introduces a speech recognition decoder model by using the Viterbi algorithm. Based on some Hypothesis, this report arranges several experiments on different strategies of improving the (WER). By summarizing and analyzing the results, this report finally explains how the strateries effects and gives the optimal optimization in this assignment data.

1. Introduction

1.1. Dataset

We use the dataset provided by the Uoe ASR course. It contains 354 pieces of records which are composed of 10 distinct words: "a, of, peck, pepper, peter, picked, pickled, piper, the, where's". There are 16477 words in totally in the reference transcriptions. We also find that among all of the lexicons, there are two special words that owns two pronunciations, 'a' and 'the'. If we use the same methods to process these two words as others, there will be only one way to generate them in the WFSTs. So, we mark one of the pronunciation with '*', like 'a*', 'the*'. In this way, the WFSTs will be complete and another pronunciation of these two words will be correctly recognised by the ASR system.

1.2. Task Defination

There are 4 tasks in the project, which could be classified into three classes. The first one is to inital the ASR system, which could be used as a baseline model; the second one is to improve the accuracy of the automatic speech recognition system and the last one is to improve the computational efficiency of the system.

In the part of improving the accuracy, two methods are used, tuning hyperparameters and using different grammars, like unigram and bigram. The hyperparameters we need to tuning contain the self-loop probability, the transition probability and whether adding silence states to the wfst. In the part of improving the efficiency, we try several methods, pruning, beam-search and tree structure lexical model.

The metric we use to evaluate the accuracy of an ASR system is word error rate (WER)(Jurafsky & Martin, 2009)

$$WER = \frac{S + D + I}{N}$$

where S, D and I indicate subsitutions, deletions and insertions between the transcriptions and the output of the ASR system, and N represents the number of words in the transcriptions.

The metric we use to measure the efficiency of the ASR system is the memory space it uses and the time it takes to recognise a piece of record. To measure the memory it uses, we compute the number of HMM states and arcs in the WFSTs. At the same time, we record the number of forward computations (FC), which is how many times that forward function is called during the Viterbi decoding procession. We choose to compute the time that Viterbi decoder uses to measure the time the system takes.

2. Initial systems

To train a baseline model, we use uniform probability distribution, which means that one HMM state will transition into the next state with a probability equalling to the probability that it loops to itself (the self-loop probability: 0.5; the transition probability: 0.5). At the same time, the start state will go into each word HMM model with the same probability.

Methodology: The baseline experiment is designed in two methods:

1. The WFST only output phone (states) labels, which shows in Figure 6. Therefore, the recognition sequence will be the phone sequence. Then we design linear phone sequence WFST and lexicon transducer WFST: L. Then use the recognition phone sequence to generate its linear phone sequence WFST. Do compose with the L to transfer phone sequence into word sequence. Due to the linear WFST will add epsilon as output but final word for the match phone sequence, the composed WFST need to remove all the epsilon. Also the remove epsilon will both remove the output word but input epsilon. So it need to do copy all output in input before the remove action. Finally, by cycling all the arcs to get recognition word sequence.

2. The WFST output epsilon label for phone (states) input, but only output corresponding word label on the final arc to end state, shows in Figure 7. This will directly output the recognition as word sequence by implementing the Viterbi algorithm.

All the arc weights are computed by the following formula.

$$Arc\_weight = -log(P)$$

where P is the arc probability.

Experiment	Self-loop	Transition	WER	FC per wav
Word1-9	0.1	0.9	1.479	75153.098
Word2-8	0.2	0.8	1.028	75615.143
Word3-7	0.3	0.7	0.874	75935.343
Word4-6	0.4	0.6	0.813	76191.670
Word-Baseline	0.5	0.5	0.769	76418.861
Word6-4	0.6	0.4	0.732	76640.061
Word7-3	0.7	0.3	0.706	76881.067
Word8-2	0.8	0.2	0.681	77174.090
Word9-1*	**0.9**	**0.1**	**0.659**	**77611.414**
Phone1-9	0.1	0.9	1.479	75153.098
Phone2-8	0.2	0.8	1.028	75615.143
Phone3-7	0.3	0.7	0.874	75935.343
Phone4-6	0.4	0.6	0.813	76191.670
Phone-Baseline	0.5	0.5	0.769	76418.861
Phone6-4	0.6	0.4	0.732	76640.061
Phone7-3	0.7	0.3	0.706	76881.067
Phone8-2	0.8	0.2	0.681	77174.090
Phone9-1*	**0.9**	**0.1**	**0.659**	**77611.414**

Table 1. Result of WFSTs output phone and word sequence

All the language model probabilities will also be converted into negative log type as previous declaration.

For both two methods, the are weight from start state to each first phone of words WFST state are set as the same weight (which is negative log(1)).

Result: We set 9 groups of comparison for the 2 methods by varying the arcs weight (tlaked in Section 3.1): The probabilities pairs vary from (0.9, 0.1) to (0.1, 0.9) (probability interval: 0.1). The result shows in Table 1. After sample words sequence recognition, the results of two methods are completely the same, as well as the average of WER, decoding time, backtrace time and forward counts.

Due to the time consuming and energy consuming of method 2 because there's no compose and generate linear phone sequence WFST. ALl the following experiments are set by using method 2 to do the recognition.

Our baseline model as shown in Table 2.

3. Improving the recognition accuracy

3.1. Varying the self-loop probabilities and transition probabilities

Hypothesis: Depending on the method of decoding, introduced in Section 2, the optimal recognition sequence is decided by the cost of the HMM path. The cost of different paths are computed by 3 variables: emission probability, transition probability and the path cost of reaching the last state. Also, the transition probability is presented by self-loop probability and transition probability. Thus, varying the self-loop probabilities and transition probabilities may affect the recognition. By setting self-loop probability from 0.1 to 0.9 (interval 0.1), and accordingly setting transition probability from 0.9 to 0.1, this series of experiments can show us the trend that how self-loop probabilities affect the recognition accuracy.

Methodology: According to the method of using WFSTs to represent HMM to recognize speech, the sum of all of the arc probabilities that coming out from a state should be 1. Depending on the WFST structure and that rule, the experiment will varying the self-loop weight and transition weight synchronously for one state's all arcs. Other parameters of the experiment keep unchanged with baseline.

Result: We can see from Table 3. With the improvement of self-loop probability, the performance of the model increases. When the self-loop probability equals to 0.9, the model performs best with a WER equalling to 0.657. This may be because most of the phones are not only three states when they are recognised. If the self-loop probability is low, the phones will be more probable to go into next state, and there will be more phones recognised. When WER is computed, there will be more deletions so that the WER is high.

Conclusion: Increasing the probability of self-loop will increase the accuracy of recognition.

3.2. Varying the final probabilities and the propagate probabilities

Hypothesis: The final probability is the end states weight in WFST. The propagate probability describes the arc weight from an end state to the start state. According to the law introduced in Section 3.1, the end state's end weight and the weight for back to start state to propagate should be sum to probability 1. Intuitively, the end weight decides how possible the recognition progress ends in this state. Accordingly, the back arc weight decides how possible the recognition progress continue to recognize. Thus, varying the final probabilities and the propagate probabilities may affect which word state has more probability to end the recognition or continue, which will affect on recognition accuracy.

Methodology:

1. Setting the end weight and back arc weight of each end state as 2 groups: 0.1 (end weight) and 0.9 (back arc weight); 0.5 and 0.5. Keep the same weight group for all end state and keep other parameter unchanged with baseline.

2. Setting the end probability through probability 1 minus the propagate probabilities for each end state. set the the propagate probabilities through the probability of how possible the represented word of that state appears in a recognition, which will be computed with the all transcription by unigram rule. Keep other parameter unchanged with baseline.

Result: For methodology 1, the results of two groups are the same and also are the same with baseline result. The reason of this is that the end weight hasn't be computed or compared in the Decoding algorithm. The path cost (weight) only equals to sum of emission probability and transition probability (self-loop weight and transition weight) and linked last state cost at last time. Also, due

Experiment	Self-loop	Transition	Total Error	WER	FC per wav	Avg Decode Time	Avg Backtrace Time
Baseline	0.5	0.5	2182	0.769	76418.861	3.462 s	4e-4 s

Table 2. Variable self-loop probability and transition probability

to all the back arcs weight are the same, this will donate to the path cost equally for each word path through the 'traverse_epsilon_arcs' function in the decode algorithm. Therefore, it just arbitrarily add same cost to different word paths.

For methodology 2, the results of it is the same to the result using the unigram model. This will be discussed in detail in Section 3.3

Conlusion: Whether change the final probability (end state weight) doesn't change the result because it doesn't be computed or compared in the Viterbi decode algorithm. However, it only is used as condition check for selecting the paths cost ending in end state out because only the end state output word symbols.

3.3. Adding unigram word probabilities

Hypothesis: The unigram word probability describes the likelihood of the word appearing in the document (the document means 354 transcription files in this experiment). If it be added in the WFST as transition weight of start state extending arcs to each word WFST. That means using the how possible a word appears in the speech transcription to decide how much it increases the score of this word recognized (how much the weight be added into that word WFST from the start state). Rather than arbitrarily add equal weight to each word WFST from start state as baseline setting. Therefore, adding unigram word probability from language model may improve the recognition accuracy.

Methodology: The rule of unigram word probability computation:

$$P(w_i) = \frac{C(w_i)}{N}$$

where w_i represents a specific word, $C(w_i)$ indicates the number of w_i appearing in the transcriptions and N is the total word number.

Result:

We can see from Table 3 that the unigram model helps the baseline model increase the accuracy by around 0.002 except U1-9 with an improvement of 0.033 and U2-9 with an improvement of 0.01. We can know that unigram model is not very useful for increasing the performance of the asr system on this dataset. One possible reason is that the lexicon vocabulary is small and another one is that many of the transcriptions are not meaningful with simple combinations of several words.

Conclusion: Unigram is not suitable for this system because of the limitation of the dataset. It may be useful for real speech recognition tasks.

3.4. Adding silence state in WFST only & Combining with the best result of Section 3.1

Hypothesis: According to the result of baseline, the insertions of baseline wer dominates highest component. One possible explanation of that is silence frames of speech have been recognized as word due to the fact that there must be silence intervals in a speech. Thus adding silence recognition may improve the recognition accuaracy, especially in reduction of insertions.

Methodology: According to the specification, the observation can compute the observation probability for silence using the label "sil_N" for N=(1,2,...5). In order to adapt to all length of silence duration, the silence path WFST should be generated to go through the silence path from one arc or infinity arcs. Therefore, the loop path should be added into the silence path WFST. The silence arcs probabilities are set into 1/N (N is count of arcs from a silence state).

In order to compare with baseline, set the pair (self-loop probability, transition probability) as (0.5, 0.5) the same with baseline.

As the result of Variable self-loop probability and transition probability in Section 3.1, the best recognition of probabilities pair is (0.9, 0.1). Thus set another experiment with adding silence on probabilities pair (0.9, 0.1) to verify if the accuracy will be improve further.

The wfst is shown in Figure 1

Result: The result shows in Table 4 Baseline comparing: the WER improves 0.293 from 0.769 to 0.476 Comparing with adding unigram in the best probabilities pair of section 3.1: the WER improve 0.207 from 0.659 (unigram) to 0.452 (silence)

Conclusion: So far, the improvement of adding silence is the biggest, rather than adding unigram and verying the probabilities pairs (because adding silence has the lowest WER). This also confirms what the hypothesis supposed that there must be silence frames due the fact that people speak words with intervals. If there's no state to recognize the silence (intervals), these speech frame will be recognized as words in the WFST which must be errors.

3.5. Improving the grammar & Combining with the best result of Section 3.1

Hypothesis: The N-gram model is a contiguous sequence of n items from a given sample of text or speech. bigram model (N=2) and trigram model (N=3) can describe how

Experiment	Self-loop	Transition	Uniform WER	Unigram WER	Bigram WER
U1-9	0.1	0.9	1.479	1.446	1.224
U2-8	0.2	0.8	1.028	1.018	-
U3-7	0.3	0.7	0.874	0.873	-
U4-6	0.4	0.6	0.813	0.810	-
U-Baseline	0.5	0.5	0.769	0.766	0.875
U6-4	0.6	0.4	0.732	0.731	-
U7-3	0.7	0.3	0.706	0.704	-
U8-2	0.8	0.2	0.681	0.679	-
U9-1*	**0.9**	**0.1**	**0.659**	**0.657**	**0.779**

Table 3. Variable self-loop probability and transition probability for Uniform, Unigram and Bigram model

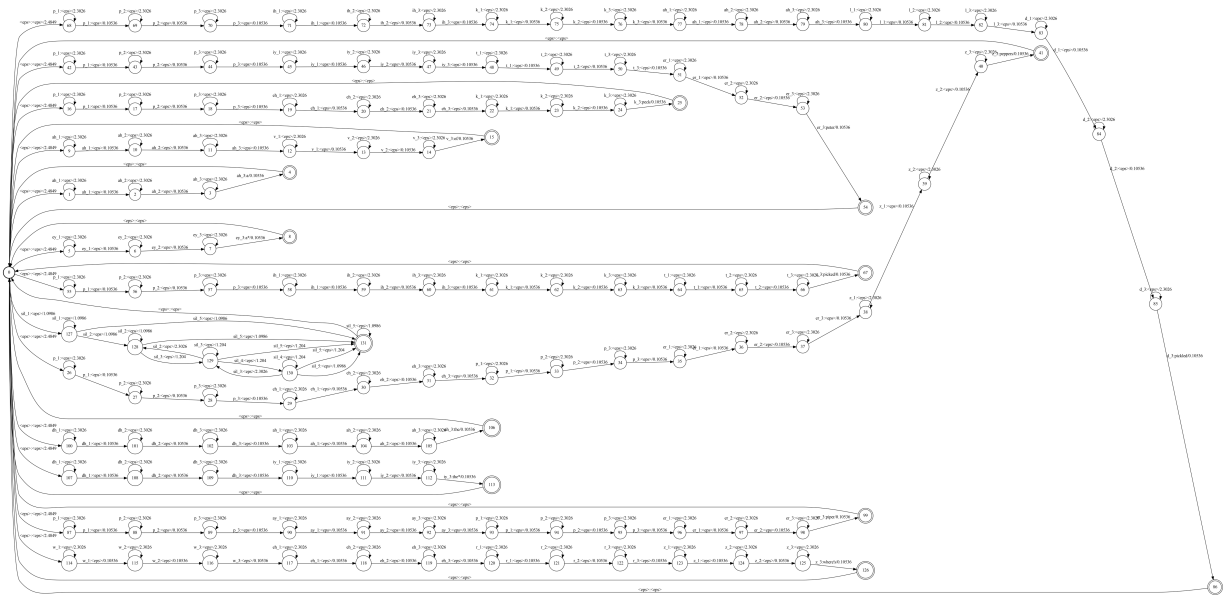

Figure 1. WFST adding silence

Experiment	Self-loop	Transition	WER	S-D-I
U-Baseline	0.5	0.5	0.769	740-109-1341
U9-1	0.9	0.1	0.659	738-136-1092
S-Baseline	0.5	0.5	0.476	704-205-448
S9-1	0.9	0.1	0.452	699-293-296

Table 4. Adding silence to baseline model and U9-1

the words are modeling languages which may indicate how the grammar construct languages. Therefore, according to the rule of bigram and trigram, generate the bigram probabilities and trigram probabilities from dataset transcriptions and add them into the word sequence cost in decoding process. That will intuitively add more weight of constrains of word sequence to balance each words sequence paths costs which may bring more confidence on the minimum cost path with this condition. Therefore, adding bigram words probabilities and trigram words probabilities might improve recognition accuracy.

Methodology: Bigram words probabilities rule:

$$P(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

where $C(w_{i-1}, w_i)$ means the number of the bigram appearing in the transcription and $C(w_{i-1})$ indicates the count of unigram w_{i-1}. According to the previous formula of bigram words probabilities, it's easy to compute them from the dataset transcriptions (354 transcriptions files) and save them into a dictionary (a variable type of python) to extract further. However, to add them into speech recognition process is difficult. Following are two methods for that:

1. Add the bigram probability into Viterbi decoding process. Briefly, add the weight (bigram probability converted) into the arc from start state to the first phone label state of the second word. The weight value is chosen by the first word which should be corresponding to the last state before start state at last time and the second time which should be corresponding to the output word of the path of located arc's path. The located path means where the arc bigram weight added sets.

However, we haven't finished that, although we have to

tried revise the Viterbi decode algorithm to realise that thought! Here're some insight reasons why it couldn't be implemented in the Viterbi decode algorithm from lab: First, that thought needs output word (the first word) of last end state at last time. The output word can be picked by the end state number or index. The end state (in all states) can be conditional check by the end state weight (initially 0.0). Secondly, if the located arc and the corresponding phone label state are known, how to get the second word. It is possible to generate a table to match the first arc from start state and the represented word on that path, where the first arc locates in, when just finishing generating the WFST for whole system. This because all the arcs are indexed in Openfst model as soon as the WFST settled. Finally, if there are first word and second word, it's possible to get the bigram weight. However, another problem is we don't know which last end state point to the start state at last time when it point to the located arc's phone lable state in the Viterbi algorithm from lab.

2. Generate a WFST for modeling the bigram rule which should set word as both input and output of an arc. The input word represented the first word of bigram. Accordingly, the output word represented the second word of bigram. The arc weight corresponds to the bigram weight, converted by corresponded bigram probability. If there's no probabilities of some sequence of the words, so don't add that arc from the first word state of that sequence to the second word state of that sequence. If the same word shows in sequence, that will be add a self-loop arc on that word state. The WFST is shown in Figure 2

As the word recognition WFST sets each word WFST paths, the inputs labels of arcs on that path are phones label per constant number (the number describe how many state to be model for each phone). The output labels of arcs on that path are epsilon except the arc from the final phone's final state to the end state. That output label is the represented word. Therefore, this word recognition WFST can be composed with bigram WFST. Due to the bigram WFST output are words, so the composed WFST outputs also words and extend each original word path output from the one word to all the word in the lexicon. For different word output, the weight of the arc from final phone's final state to the all words end state of the composed WFST are added the corresponding bigram weight. All the weight are converted in negative log type.

In order to compare with baseline, set the pair (self-loop probability, transition probability) as (0.5, 0.5) the same with baseline. Besides, set the probabilities pairs as (0.1, 0.9) the worst result in 3.1 and (0.9, 0.1) the best result in 3.1 to explore how the bigram affect on different arc weight settings, and what's the trend of improvement when combining the language model and acoustic model.

Trigram words probabilities rule:

$$P(w_i \mid w_{i-1}, w_{i-2}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-1}, w_{i-2})}$$

where $C(w_{i-2}, w_{i-1}, w_i)$ represents the count of trigram (w_{i-2}, w_{i-1}, w_i) and $C(w_{i-1}, w_{i-2})$ means the count of the bigram.

According to the previous formula of trigram words probabilities, it's easy to compute them from the dataset transcriptions (354 transcriptions files) and save them into a dictionary (a variable type of python) to extract further. Based on generating bigram WFST, the trigram WFST only take should be generated with 2 state for each the tri-words sequence. However, it starts disorders when add reach and back arcs between two states.

$$P(theofpicked) = 0.1P(pickedofthe) = 0.2$$

$$P(pickedtheof) = 0.3P(ofthepicked) = 0.4$$

$$P(ofpickedthe) = 0.5P(thepickedof) = 0.6$$

The upper trigram probabilities correspond the Figure 3, Figure 4, Figure 5. To keep the sequence correct, the first word of the sequence set in input from the first word state and the output arc as '<eps>' because it don't want to add any more word in that sequence as the second arc has both input and output words as the words sequence. Therefore, when combine these 3 WFS, there is trouble as the same direction of arc and the same start and end states have different weight, depending on the corresponding trigram probabilities. So we haven't found a solution to solve that problem.

Result We can see from Table 3 that when the self-loop probability is low, the bigram will help to increase the accuracy of recognition. However, when the self-loop probability is high, bigram will decrease the accuracy. This may be because that bigram can help the system find a better path to generate a right pair of words when self-loop probability is low.

Conclusion: The accuracy of add bigram doesn't improve. The one possible reason is that the transcriptions of dataset do not follow the English grammar well. Eg:"peppers peppers peppers peppers peter" in '0008.txt' transcription file. Thus, this increases the complexity of the bigram (increase combinations of lexicon words). However, there are only 354 files which means small amount of transcriptions. Therefore, the bigram probabilities computed from these data are less accurate. If adding them into the recognition system, it may lowered the accuracy more.

4. Improving the recognition efficiency

4.1. Pruning

Hypothesis: In this experiment, we use dropping out possible paths with low probabilities as our first pruning strategy. By cutting off these paths, the decoder does not need to compute each path and the time that the decoder takes will be reduced. We will set a specific threshold, which will be compared with the negative log likelihood of a specific

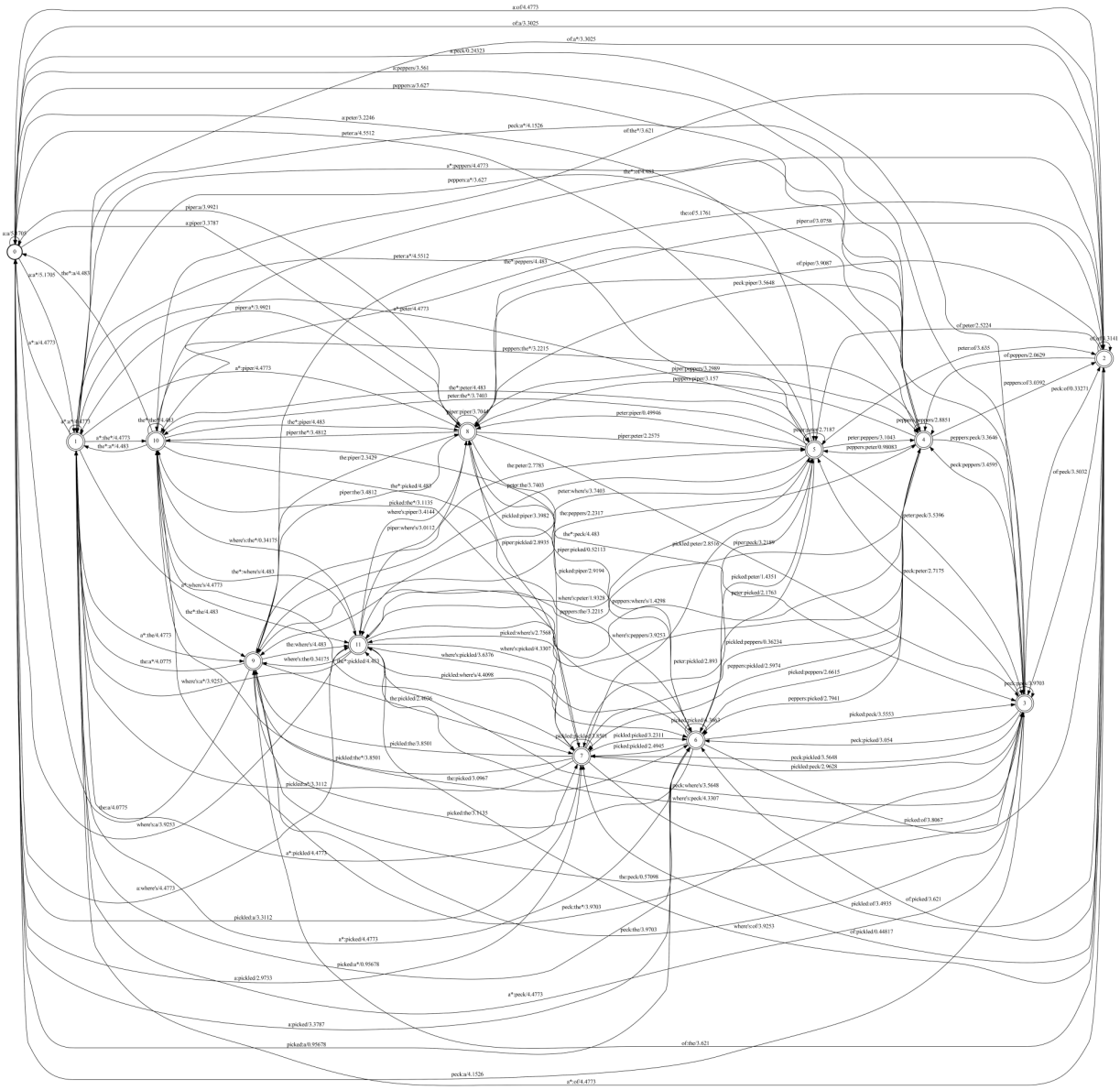

Figure 2. WFST adding bigram

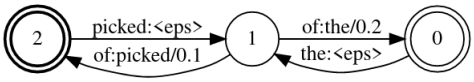

Figure 3. Trigram Example 1

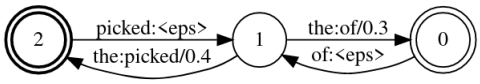

Figure 4. Trigram Example 2

path. If the likelihood is smaller than the threshold, it will be dropped out.

Methodology: The pseudo code is shown in Algorithm 1. We design the experiments with pruning_threshold equalling to 3000, 6000, 10000, 20000, 30000, because during out attempts to minimize the threshold, the model

Algorithm 1 Pruning

```
Initialization: pruning_threshold ← t
for state in wfst_states do
    path_cost ← negative_log_likelihood(state)
    if path_cost ≠ ∞ then
        if path_cost ≤ pruning_threshold then
            Viterbi Decoder forward()
        end if
    end if
end for
```

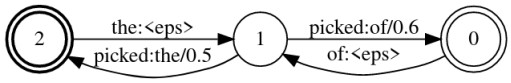

Figure 5. Trigram Example 3

does not work well because it cannot compute the effective ways to compute WER. Other parameters are the same with baseline.

Result: We can see from Table 5 that compared with baseline model, when the pruning_threshold is smaller than 20000, the pruning can help to reduce the number of forward being called and the decoding time, especially when the threshold is 3000, the number of forward being called per wave is reduced from 76418.861 to 55284.037. When the threshold is bigger than 20000, the models perform the same as baseline, that is because the threshold is so big that it cannot cut off any path while decoding. Actually, when it's 10000, it is already close to baseline. However, pruning will decrease the accuracy of the recognition, though it can help increase the efficiency. When the threshold is 3000, the WER increases from 0.769 (baseline) to 0.788. This is because there may be some correct paths being dropped out due to the high path cost.

4.2. Beam search

Hypothesis: We use another pruning strategy called beam search. We first define a variable *beam_width*. In the process of decoder forward, the negative log likelihoods of all of the states in the WFSTs are sorted from the smallest one to the biggest one. We will get the biggest *beam_width* states and continue the forward function. In this way, the paths with bigger negative log likelihood, which could be regarded as the cost of the path, will be cut off. Thus, the decoder forward function will be more efficient.

Methodology: The pseudo code is shown in Algorithm 2.

Algorithm 2 Beam Search

```
Initialization: beam_width ← b
for state in wfst_states do
    path_cost ← negative_log_likelihood(state)
    path_cost_list.append(path_cost)
end for
Sort(path_cost_list)
beams ← path_cost_list[: beam_width]
for state in beams do
    if likelihood ≠ ∞ then
        Viterbi Decoder forward()
    end if
end for
```

We conduct the experiments with beam_width equalling to 50, 100, 150, because we find that the maximum beam_width is around 120, so a beam_width far from 120, close to 120 and bigger than 120 can clearly help us to find the affect of beam_width on the model. Other parameters are the same as baseline.

Result: We can see from Table 4 that beam search with a width of 50 and 100 can help to reduce the forward computation counts and the decoding time, especially 50. It reduces the average decoding time from 3.580 s to 2.669 s, and reduces the count that forward is called from 76418.861 to 37098.109, almost by 50 percent. When the beam_width is 150, the model will perform the same as the baseline model. This is because 50 and 100 is smaller than the maximum of beam_width (about 120), so some of the paths will be cut off, and the model will be trained with less memory.

4.3. Tree-structured lexicon

Hypothesis: The tree-structure of lexicon will reduce reuse of arcs for the same phone between words. Intuitively, this will reduce the number of states and arcs of the WFST. Thus, the decode loop counts will also be reduced because the loops are cycled by the total states of the WFST and the arcs from that states. Therefore, adding the structure may reduce the time of decode and backtrace time, as well as the forward computations counts.

Methodology: In fact, the wfst created by the tree-structure lexicon is the same as the determinize principle of an ordinary wfst of the lexicon because the determinize make WFST co-use the same input labels (phones) and add the plus weight later when the arc is independent (cannot be splited to more than one word), which will still keep the total weight of the word path as the same as before determinize. The tree-structured WFST shows in Figure 9

Result: According to Table 5, comparing to baseline: the average decoding time reduce 0.711 s from 3.580 s to 2.869. the average backtrace time increases 1e-4 s from 4e-4 to 5e-4. The average forward count reduce 25047.073 from 76418.861 to 51371.788. And the state/arc nums reduce , from 127/252 to 87/193. This means not only the decoding time that tree structured lexicon takes is shorter, but also the memory it uses is smaller than the baseline model. At the same time, the accuracy of the tree structured model is the same as the baseline model, which indicates the accuracy is not affected when increasing the efficiency.

Conclusion: The Tree-structure lexicon method works perfect not only on implementing it easily, but also on a good improvement with no risk of decrease the accuracy of recognition.

5. Conclusions

In this project, we implement some techniques to increase the accuracy and efficiency of an ASR system. For the experiment data, the best model should be self-loop probability 0.9, transition probability 0.1, adding silence, adding unigram and using tree structured lexicon. This model will not only gain a low WER, but it can be trained quickly.

Experiment	Avg Decoding Time	Avg Backtrace Time	FC per wav	States/Arcs	WER
U-baseline	3.580 s	4e-4 s	76418.861	127/252	0.769
Beam width-50-baseline	2.669 s	7e-4 s	37098.109	127/252	0.848
Beam width-100-baseline	3.229 s	5e-4 s	61676.892	127/252	0.775
Beam width-150-baseline	3.880 s	5e-4 s	72222.887	127/252	0.769
Pruning-3000-baseline	4.143 s	7e-4 s	55284.037	127/252	0.788
Pruning-6000-baseline	3.713 s	5e-4 s	72034.045	127/252	0.763
Pruning-10000-baseline	3.916 s	5e-4 s	75617.206	127/252	0.769
Pruning-20000-baseline	3.899 s	6e-4 s	76418.862	127/252	0.769
Pruning-30000-baseline	4.432 s	6e-4 s	76418.862	127/252	0.769
Tree Structure-baseline	2.869 s	5e-4 s	51371.788	87/193	0.769

Table 5. Different Methods of improving efficiency with self-loop probability 0.5 and transition probability 0.5

References

Jurafsky, Daniel and Martin, James H. Speech and language processing, 2009.

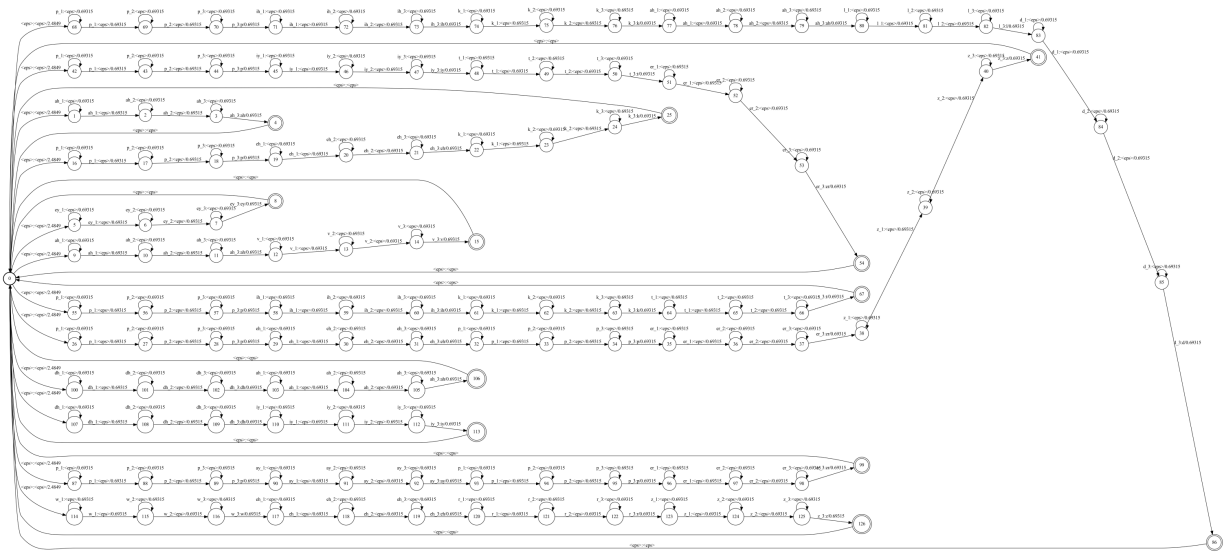

Figure 6. $baseline_{o}utput_{p}hone_{l}abel$

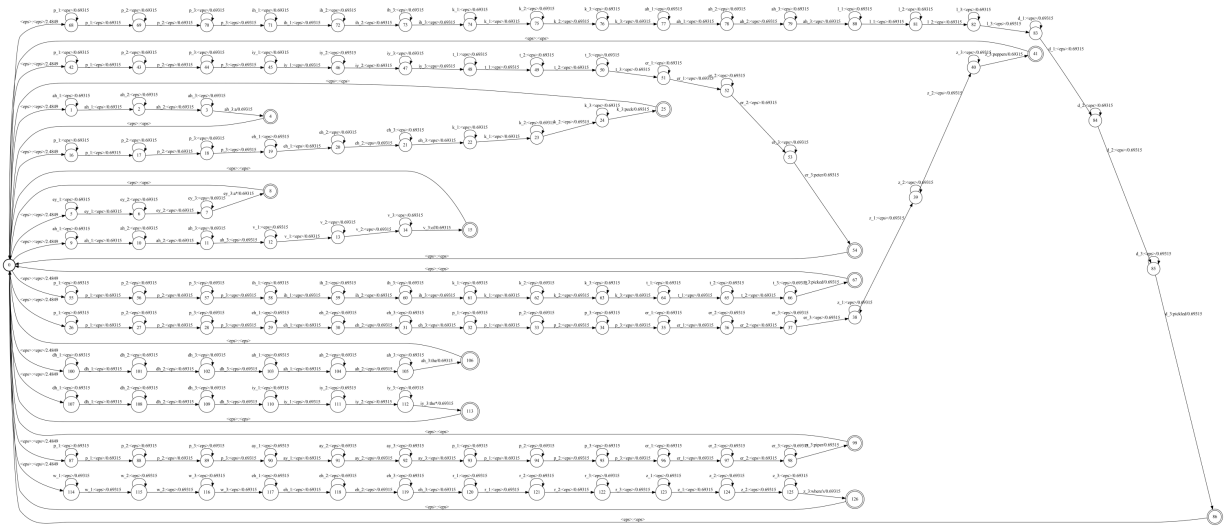

Figure 7. baseline_word

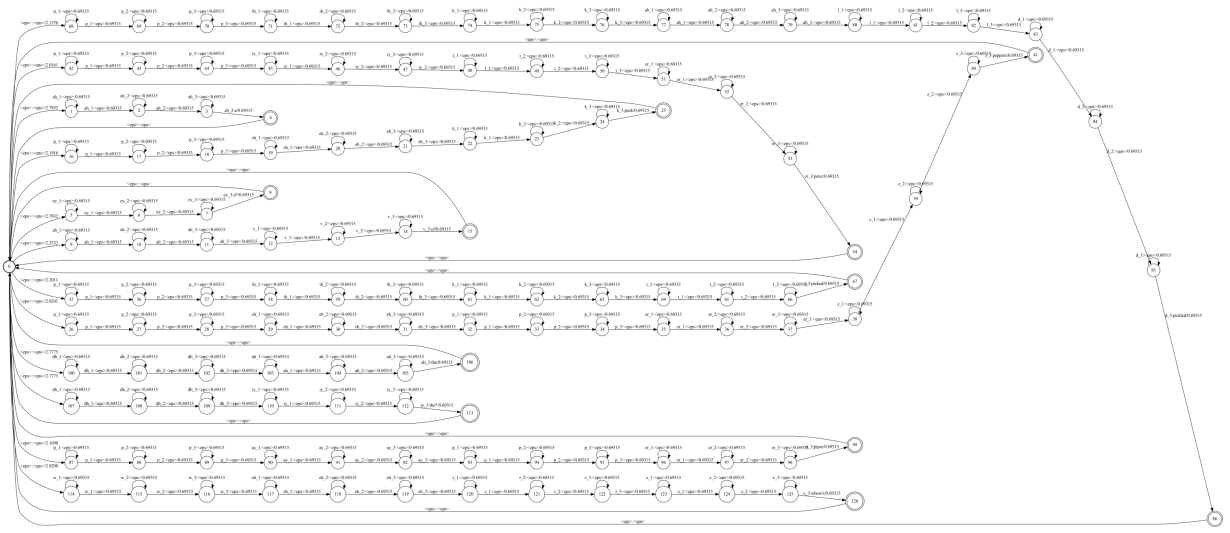

Figure 8. baseline_unigram_word

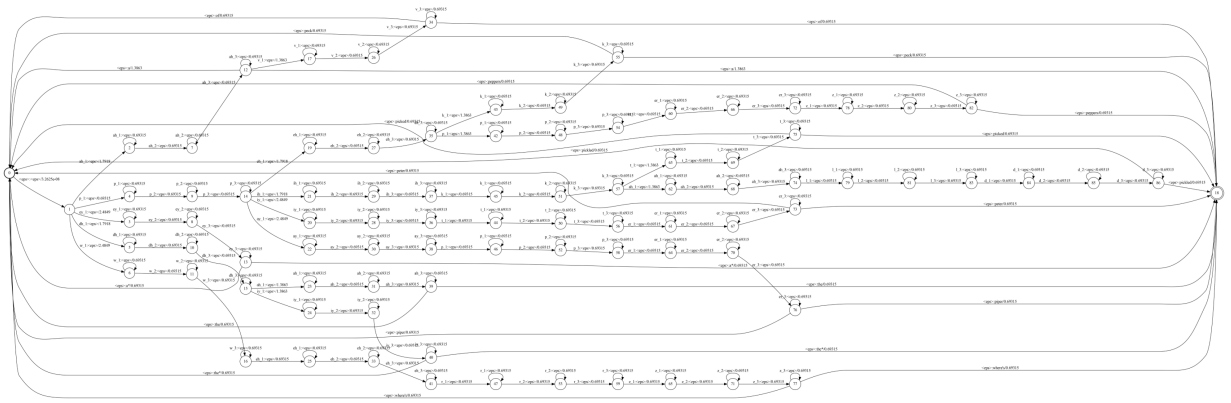

Figure 9. *baseline_tree_word*

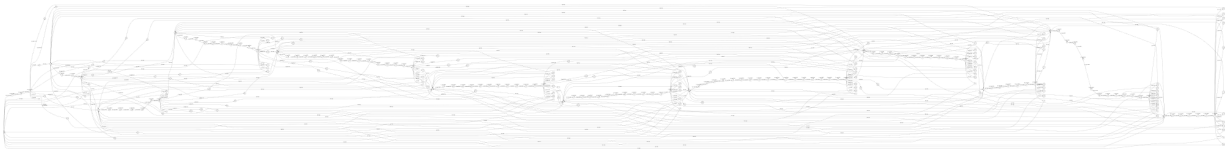

Figure 10. *com_bigram_word*