INTERSPEECH 2009 BRIGHTON

Algorithms for Speech Indexing in Microsoft Recite

Kunal Mukerjee, Shankar Regunathan and Jeffrey Cole

Microsoft Corporation

{kunalm, shanre, jecole}@microsoft.com

Abstract

Microsoft Recite is a mobile application to store and retrieve spoken notes. Recite stores and matches n-grams of pattern class identifiers that are designed to be language neutral and handle a large number of out of vocabulary phrases. The query algorithm expects noise and fragmented matches and compensates for them with a heuristic ranking scheme. This contribution describes a class of indexing algorithms for Recite that allows for high retrieval accuracy while meeting the constraints of low computational complexity and memory footprint of embedded platforms. The results demonstrate that a particular indexing scheme within this class can be selected to optimize the trade-off between retrieval accuracy and insertion/query complexity.

Index Terms: recite, speech index, speech retrieval

1. Introduction

This paper describes indexing for storage and retrieval of spoken notes on a mobile platform. An example of such an application is Microsoft Recite (http://recite.microsoft.com). Notes are inserted as well as queried using speech. In this context, the primary challenge that emerged was the design of an indexing scheme that would serve as a consistent and noise robust one-way hashing function from a single user's speech into a symbol stream.

This application and device context imposes certain constraints which have guided our algorithm design choices in order to improve the overall user experience. First there is the issue of out of vocabulary (OOV) phrases. User studies revealed early on in the project, that our target device would have to deal with a large percentage (e.g. 20%+) of OOVs. A typical example is: "Grocery list: milk, eggs, hummus, pita bread, Camembert". Second, the quality of retrieval must be exceptionally good. If the correct match is not in the top 5 results then the user will probably consider the query as having failed because a mobile interface (Figure 1) makes it very arduous to go through a long list of suggested matches. Third, the system needs to be exceptionally noise robust, because the device may be used under many different environmental conditions, as well as need to cope with intra-user speech variations. Also, the index and retrieval system must support out of order query terms, because the user is very likely to not remember the order of things they inserted, e.g. the query for the above reminder is very likely to be: "Grocery list: pita, milk". Everything about such a system, including recognition, compression, tagging, indexing and query/retrieval must be real-time and fit on a mobile or embedded platform with frugal memory footprint and CPU cycle budgets. All of these design constraints make ours a challenging research problem. Next we will outline the rationale underlying some of our design choices.

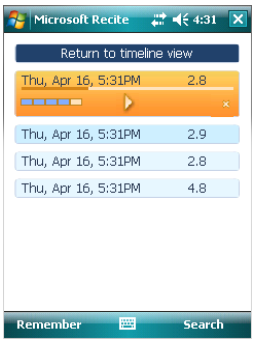

Figure 1: *Recite query interface (on mobile phone)*

The choice of "what to index" is a critical one for speech based indexing. The existing literature shows a rich diversity of approach in this area. Large Vocabulary Continuous Speech Recognition (LVCSR) word and phoneme based lattices are compared in [2]. Sub-word units or "particles" are reported in [3]. We investigated indexing with traditional language model (LM) based approaches, but our results corroborate those of [5], i.e. this approach fared very poorly given our high OOV rate. Furthermore, the large memory and CPU cost of using LMs could not be justified on our resource-constrained platform. We also investigated indexing with more primitive representations such as quantized smooth group delay spectrum features [4], but found that the data rate and entropy (i.e. quantization range) incoming into the database were too large to support real time retrieval.

In our application context, it is a high priority to deal with OOV, as well as be language neutral to a large degree – these made indexing on symbols derived from phonemes an attractive proposition. At the same time, Recite is primarily meant to be used by a single user, as it is a mobile phone application. That allows us to utilize certain speaker-centric features. Therefore, we combined supervised classes derived from confusability clusters of phonemes, as well as unsupervised classes which capture information about speech transients and other non-phonemic information, into an extended symbol alphabet. The index stores and matches n-grams of this set of symbols.

The next task involved defining a new metric to report and track success, given the needs of our particular application. In this respect, Recite differs from many speech indexing applications reported in the literature ([2][3]) which target huge databases and have a "computer screen" like visual form factor interface for presenting the query results. In our case the response must always and only be real time, and the CPU/memory footprints must always be small enough to fit on a mobile or embedded platform. But more importantly, due to our restricted interface, the correct query results need to be at the top, or it just won't work for the end user. Therefore, after doing user group studies we came up with the following simple metric to track and report success of our end-to-end system:

10.21437/Interspeech.2009-451

Successful result = Exact Match In Top 5 (1)

Our query algorithm is based on approximate string matching with an inverted n-gram index that expects errors in both inserted and query strings. It has similarities to [6], in that we also define a noise robust index for fragmented/overlapping matching sub-sequences, and the scoring is similar, in that in a final step, all high ranking hypotheses are further examined by aligning and scoring the areas of corresponding similarity regions, and paired regions are extended in both directions. However, our queue based insert and query algorithms are 1-pass and *O(N)* in the strings being inserted and queried, whereas the querying algorithm used in [6] is a 2-pass algorithm, and we use an exponential scoring scheme that is more resilient to noise.

We will proceed to describe and analyze the indexing algorithms in greater detail in the following sections.

2. Indexing Algorithms

There are two versions of n-gram indexes in the Recite family: a hierarchical and a multi-level version. Both versions are basically an inverted n-gram symbol index. The hierarchical version trades off space for query speed (i.e. bigger disk footprint), whereas the multi-level version takes less disk space and allows trading off query speed and search accuracy. These combinations allow us to tune these algorithms for different run time platforms, all the way from small mobile devices to server based systems, where we can turn up the accuracy.

2.1. Hierarchical N-gram index

In the hierarchical version, at insertion time we index the incoming sequence of symbols by creating 2-gram, 3-gram, …, *MaxNGram* nodes of all n-gram subsequences that occur in the lattice of symbols. At query time, we check to see how the n-grams from the query string match with existing nodes in the database and score the ones that have long contiguous matching runs. Our scoring algorithm described in section 3 accounts for fragmentation of long runs, i.e. have some number of insertions, deletions and substitutions within a run of matching symbols. Additionally, the scores of the matching regions are boosted based on the relative similarity of time durations of matching symbols.

2.1.1. Hierarchical Insert and Query

Both insert and query use queue-based algorithms. As each symbol is processed, we adjust either the index at insertion time or the matching runs at query time, and the symbol queue moves to the next symbol position. Items get de-queued either when their length exceeds *MaxNGram* at insert time, or when they stop matching strings in the database at query time. Both insert and query are 1-pass and only ever visit each symbol position once, i.e. they are *O(N)* in the length of the inserted and queried strings.

Figure 2 and Figure 3 show pseudo-code for insert and query algorithms respectively. Hashed tree traversals are constant time, such as in LZW [1]. In these figures, R is a set of (S, p) pairs that share a common n-gram, where S is an inserted string id and p is a position within S. R_n denotes the set R that is associated with n-gram node n. S_i is the i^{th} symbol in S and Q_i is the i^{th} symbol in Q.

```
Insert(InsertSequence S)
{
  SQ <- Empty queue of n-grams (Source)
  DQ <- Empty queue of n-grams (Destination)
  for each position p in S do
    Swap(SQ, DQ)
    Enqueue the 1-gram Si on DQ
    for each n-gram ngi in SQ do
      if HashedTreeTraversal(ngi, Si) exists
      then Add(S, p) to Rn
      else if Length(ngi.Si) < MaxNGrams
      then CreateNewNode(ngi.Si), Add(S, p)
      end if
      if Length(ngi.Si) < MaxNGrams
      then Enqueue the n+1 gram ngi.Si on DQ
    end for
  end for
}
```

Figure 2: *Hierarchical insert algorithm*

At insertion time, we insert all 2-grams, 3-grams, … *MaxNGram*-grams at each symbol position. In practice, *MaxNGram* = 4 is a good choice where we can trade off node congestion with size of the database (see also Figure 5 in the experimental results section). For a string of length L and n-grams of length N, we know that the number of n-grams is $L-N+1$. Additionally, if there are A alternates at each symbol position in the lattice, we get A^N possibilities. In most practical phoneme based systems like ours and [7], $A \sim 2$ is sufficient. Therefore the index grows as: $(L-3) \times 2^4 + (L-2) \times 2^3 + (L-1) \times 2 = 26L - 66 = O(L)$, i.e. linearly in L.

The query algorithm proceeds by finding all matches with proper subsets of matching symbols. It is trivial to prove that all sequences that match on an n-gram must also match on the n-1 gram, i.e. matching prefixes. This allows us to "harvest" all the nodes that stop matching at a given symbol position and score these partial matches. Besides being easy to prove correct, this design has the added benefit of being cache friendly, because we start by caching in the biggest set of matches (e.g. for 2-grams), and refine that by sub-setting, as we proceed along the query sequence.

```
Query(QuerySequence Q)
{
  B <- Empty set of (S, p)
  SQ <- Empty queue of n-grams (Source)
  DQ <- Empty queue of n-grams (Destination)
  for each position p in Q do
    Swap(SQ, DQ)
    Enqueue the 1-gram Qi on DQ
    for each n-gram ngi in SQ do
      if HashedTreeTraversal(ngi, Qi) exists
      then Add set difference: Ri-Rn to B
      else Add the set Ri to B
      end if
    end for
  end for
  Rank strings in B using SCORE
  return ranked B
}
```

Figure 3: *Hierarchical query algorithm*

2.2. Multi-level N-gram index

In the multi-level index, at insertion time we index the incoming symbol stream by creating n-grams where "n" is fixed at a single configurable setting. This allows us to choose any length of n-grams to work with, including the degenerate

unigram or 1-gram case. Reducing n to unigrams improves matching accuracy because it allows fragmented matches down to the symbol level. However, using low n such as 1 or 2 slows down the scoring part of the query because the scoring algorithm now has to piece together 1-grams or 2-grams to find all the match regions. Thus, the multi-level index allows us to fine tune for speed vs. expected noise/desirable accuracy on the target platform, e.g. we only use 1-grams on PC and server based systems.

Since n is fixed, the space requirement of the multi-level index is much smaller than the hierarchical one, which redundantly stores all the sub n-grams. Additionally, we don't need to perform hashed tree traversals because the n-gram nodes may be directly looked up inside a contiguous address space numbered 0..*MaxNGrams*-1.

2.2.1. Multi-level Insert and Query

Multi-level insertion is very simple. Since the "n" of each n-gram is fixed at a single setting, we simply traverse the lattice and sequentially store off each n-gram under its node in the database, along with associated offset and duration information for those symbols.

At query time we once again travel along the lattice of symbols, and at each point we look up the n-gram node in the database, gather all the matching regions from stored sequences, and forward them to the scoring algorithm to piece together and rank. If n is low, e.g. 2, then there is a lot of collision to be expected in the database in each n-gram node, and so we have implemented caching and MRU queuing of n-gram nodes as optimizations. Additionally, when n is low, scoring is expected to do a lot more work in piecing everything together, but also be more accurate, because we can then account for very fine grained fragmentation. These observations are corroborated by the results presented in section 4 (Table 1).

3. Noise robust scoring of query results

We now describe the scoring algorithm that we use to rank the similarity between strings in the database, *S*, and a query string *Q*. The scoring algorithm starts with matching n-grams retrieved at query time, and pieces the n-grams together into longer matching regions and computes a similarity score for each string *S*. The same algorithm applies to both hierarchical and multi-level indexes.

3.1. Problem Formulation

We define Σ as the symbol alphabet. Prior to reaching the index for insertion or query, each string first passes through a noisy channel, *NC*: $\Sigma^* \rightarrow \Sigma^*$. Therefore, one way of looking at our problem is that it is a 2-pass (one at insertion, second at query) noisy channel indexing system. This abstraction captures all forms of noise, e.g. wind, electrical, quantization, model/data mismatch, intra-speaker variability, etc., and allows us to formulate our algorithms by dealing with noise under a unified framework. After inserting *N* strings, $\{S\}$, we wish to query on a string *Q*.

We define a function, SCORE, that takes a pair of strings, S_i and *Q*, and returns a number that induces an ordering on $\{S\}$ such that $S_i < S_j$ implies SCORE (Q, S_i) < SCORE (Q, S_j).

SCORE has the following properties:

- Longer contiguous matching sequences will receive higher scores than shorter matching sequences.
- Contiguous matching sequences should allow for some small amount of mismatch between the query and indexed sequences because of expected corruption of each by the noisy channel, *NC*.
- A sub-string of *Q* occurring multiple times in S_i will receive a score for each occurrence. However, a sub-string of S_i occurring multiple times in *Q* will not receive additional scores for each time it occurs.
- A score for matching symbols at any position is boosted based on similarity of the time durations associated with the matching symbols.

The unit of overlap is defined as a matching region. A matching region is a pair of substrings, *{SM, QM}* (for string match and query match), such that:

- The first symbol of *SM* matches the first symbol of *QM*;
- The last symbol of *SM* matches the last symbol of *QM*
- There must be an alignment between *SM* and *QM*, such that there are at most, *MaxSkips* un-aligned symbols between any two aligned symbols. For example, if *MaxSkips* = 1, *SM* = *abce* and *QM* = *abde*, this condition holds, but not if *SM* = *abce* and *QM* = *ae*.

3.2. Allowing match gaps in query result ranking

The similarity score can be computed by traversing through the indexed database and finding the set of exact matched lengths between each string *S* and *Q*. If each matched length is l_i and the duration similarity is d_i, the similarity score is simply the sum of all n matched regions:

$$SCORE(Q,S) = \sum_{i=0}^{n} 2^{(l_i-1)*d_i} \quad (2)$$

We choose exponential scoring because of its noise resilience as it automatically filters out short matches in favor of long ones, and also because it is cheap to compute with bit shifts. We make a small modification to this straightforward scoring system to account for the expected gaps in matching regions caused by the noise channel *NC*. Rather than requiring that a matched region should contain identical symbols for the length of the match, the process of combining smaller match regions (MR) into longer extended match regions allows for some minor errors in alignment.

The SCORE algorithm presented in Figure 4 does the necessary MR extensions and "hole-filling" operations that we need to combat expected noisy matches. It combines all substring matches between query string *Q* and candidate strings *S* in the database. The SCORE algorithm tolerates match deviations that are inside of an edit distance *E*.

4. Experimental results

Theoretical bounds calculated in Section 2.1 show that the absolute size of the index in hierarchical indexing grows linearly as the length of inserted sequences. In Figure 5 we see that the growth rate of n-gram nodes is linear for synthetic randomly generated data, but sub-linear for real data. The sub-linear growth in real data is due to the fact that certain n-grams, e.g. "z" followed by "b" are much less likely than "ih"

followed by "ng". Time to query is linear with respect to database size and also in the length of the query string.

```
SCORE(M: {Candidate sub-strings from Si})
{
  for i = 0 to |M-1| do
    for j = i+1 to |M| do
      if Mi could extend Mj with edit dist E
      then
        Extend Mj to include Mi
        Remove Mi from M
      end if
    end for
  end for
  Sort M based on length (descending order)
  Score = 0
  for i = 0 to |M| do
    if Mi overlaps with scored region Mprev
    then
      MNew = Mi - Mprev
      Add MNew to M
    else Score(Q, S)+= 2^(length(Mi)-1)
    end if
  end for
  return Score
}
```

Figure 4: *SCORE algorithm*

Small *MaxNGram* causes more collisions at each node and increases processing burden at query/scoring time, but the index grows at a slower rate and processing time decreases for insertions. Thus, the value of *MaxNGram* can be selected to trade-off query complexity with indexing complexity.

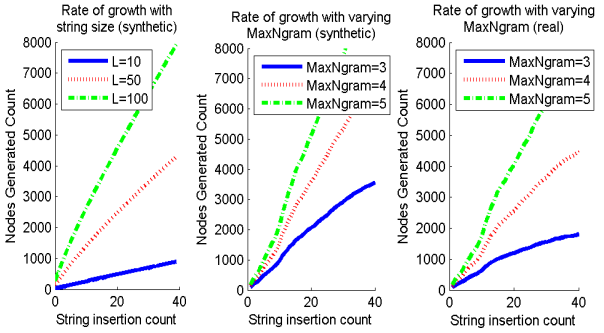

Figure 5: *Growth rates with a) insertion length; b) MaxNGrams parameter; and c) on real data*

Table 1 presents the performance of the multi-level indexing schemes on a "spoken notes" dataset of 15 users. For each user, 100 unique remembrances are inserted. The retrieval algorithm is tested using 150 queries from the same user, where 30 unique queries are repeated 5 times each. To ensure sufficient variability, the user repeats the same query only after speaking several other intervening phrases. The retrieval accuracy for each user is computed for the cases where the exact match result is in the Top 1, Top 5 and Top 10 of the retrieved results. The overall accuracy is computed as an average over all the users.

Table 1 illustrates the trade-off between retrieval accuracy and complexity for various n-gram lengths in multi-level indexing. As the n-gram length decreases, the retrieval accuracy increases and the insertion complexity decreases. However, the query complexity increases, since the scoring algorithm needs to perform a lot of additional MR extensions. Conversely, as the n-gram length increases, the query complexity decreases. However, the retrieval accuracy decreases since the scoring may miss smaller match regions, and the insertion complexity increases due to the additional effort in building larger n-gram indexes. For mobile and embedded platforms, 2-grams seem to provide a reasonable tradeoff between retrieval accuracy and complexity. For platforms with greater computational resources, 1-grams may be preferred. An important topic for future research is to compare the retrieval performance of the low-complexity Recite system against retrieval using LVCSR word and phoneme based lattices on the network.

Table 1. *Accuracy vs. complexity trade-off for different Ngrams in multi-level indexing.*

	Av. Accuracy			Av. Complexity (ms)		Memory bytes
	Top 1%	Top 5%	Top 10%	Insert	Query	
1-gram	72	87.5	92.3	1.18	261.37	261,636
2-gram	67.1	84.6	90.7	8.98	28.86	302,242
3-gram	62.3	80.9	87.9	56.12	16.97	384,174

5. Conclusions

We have presented two classes of noise robust n-gram symbol indexes for "spoken notes" retrieval applications. We also presented a query algorithm based on exponential scoring that is robust to noise. Experimental results using hierarchical indexing show that index size grows linearly with the length of inserted sequences. We demonstrated that a particular indexing scheme within the multi-level indexing class can be selected to optimize the trade-off between retrieval accuracy and insertion/query complexity. These indexing algorithms can be tailored towards low-complexity speech retrieval applications such as Microsoft Recite.

6. Acknowledgements

The authors wish to thank Kazuhito Koishida, Brendan Meeder, Nikhul Patel, Stathis Papaefstathiou and Stewart MacLeod for their help and support.

7. References

[1] Welch, T.A., "A technique for high-performance data compression", Computer, Vol. 17, pp. 8-19.

[2] Burget, L., et. al., "Indexing and search methods for spoken documents", in Proc. Ninth International Conference on Text, Speech and Dialogue (TSD), pp. 351-358, Berlin, 2006.

[3] Logan, B., Goddeau, D., Van Thong, J.M., "Real-world audio indexing systems", Proc. ICASSP, 2005.

[4] Singer, H., Umezaki, T., Itakura, F., "Low Bit Quantization of the smoothed group delay spectrum for speech recognition", pp. 761-764, ICASSP 1990.

[5] Abberley, D., Cook, G., Renals, S., Robinson, T., "Retrieval of broadcast news documents with the THISL system", In Proceedings of the 8th Text Retrieval Conference (TREC-8), 1999.

[6] Floratos, A., et. al., "Sequence homology detection through large scale pattern discovery", Proc. 3rd Annual International Conference on Computational Molecular Biology, pp. 164-173, France, 1999.

[7] Schwarz, P., Matejka, P., Cernocky, J., "Towards Lower Error Rates in Phoneme Recognition", Proc. 7th International Conference Text Speech and Dialogue, 2004.