Rapid Unit Selection from a Large Speech Corpus for Concatenative Speech Synthesis

Mark Beutnagel *Mehryar Mohri* *Michael Riley*

{mcb,mohri,riley}@research.att.com
http://www.research.att.com/info/{mcb,mohri,riley}

AT&T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932-0971, USA

ABSTRACT

Concatenative Text-to-Speech (TTS) systems such as those described by Hunt and Black [6] can select at synthesis time from a very large number of recorded units. The selected units are chosen to minimize a combination of *target* and *join* costs for a given sentence. However, the *join costs*, in particular, can be quite expensive to compute, even when this computation has been optimized. If possible, we would avoid this computation by precomputing and caching all the possible join costs, but their number is prohibitive. Although the search space of possible joins is large, we have found that only a small fraction are selected in practice. By synthesizing a large quantity of text and logging the units actually selected, we were able to gather usage statistics and construct a practical and efficient cache of concatenation costs. Use of this cache dramatically decreased the runtime of the AT&T Next-Generation TTS system [1] with negligible effect on speech quality. Experiments show that by caching 0.7% of the possible joins, 99% of the join cost computations can be avoided.

1. INTRODUCTION

Early concatenative speech synthesis systems used one stored acoustic unit sequence that matched each phonetic sequence to be synthesized [8]. More recently, synthesis systems have emerged that use an entire speech corpus as the acoustic inventory and that automatically select the units at run-time to match the phonetic and prosodic input [6, 3, 1].

This approach avoids time-consuming manual selection, and, with a large enough database, eliminates or minimizes the signal processing of the units to match prosodic conditions. It results in a more natural voice quality. A potential disadvantage is the prohibitive computational requirements of run-time unit selection. It is this issue, especially for large databases, that we address below.

2. RUN-TIME UNIT SELECTION

The automatic unit selection problem is formulated by introducing two cost functions [6]: (1) a *target cost* defined between each acoustic unit and each phone in a particular phonetic and prosodic context, and (2) a *concatenation* (or *join*) *cost* defined between each pair of acoustic units. The selected acoustic unit sequence is the one that minimizes the sum of the target and concatenation costs for a given phonetic and prosodic input.

Unit selection begins with a phonetic and prosodic specification for a desired utterance. Each phone has a feature vector, including at least pitch, duration, and stress, and (implicitly) also carries the phonetic context from its preceding and following phones.

The target cost is an estimate of the mismatch between a recorded acoustic unit and the predicted specification. Its function is to choose "appropriate" units, i.e., a good fit to the specification that will require little or no signal processing.

Target cost C^t for a phone specification t_i and acoustic unit u_i is the weighted sum of target subcosts C^t_j across features j from 1 to p.

$$C^t(t_i, u_i) = \sum_{j=1}^{p} w^t_j C^t_j(t_i, u_i) \qquad [1]$$

The concatenation cost estimates the acoustic mismatch with the goal of smooth segmental joins, and is independent of the predicted features. Concatenation cost C^c for units u_{i-1} and u_i is the weighted sum of subcosts C^c_j across acoustic features j from 1 to p.

$$C^c(u_{i-1}, u_i) = \sum_{j=1}^{p} w^c_j C^c_j(u_{i-1}, u_i) \qquad [2]$$

The task of unit selection then is to find units u_i from the recorded inventory which minimize the sum of these two costs, accumulated across all phones i in the utterance:

$$C(t^n_i, u^n_i) = \sum_{i=1}^{n} C^t(t_i, u_i) + \sum_{i=2}^{n} C^c(u_{i-1}, u_i) \qquad [3]$$

This approach is well described by Hunt and Black in [6], along with the training which produces some of the specific weights. However, as implemented, the subcost computations themselves are less elegant, and employ heuristics at several points, for example, by providing special treatment for stops, or by predisposing mid-phone concatenation.

3. THE AT&T LABS TTS SYSTEM

The AT&T Labs Next-Generation TTS system [1] is a hybrid of our previous Flextalk system [9], the CHATR system [3] from ATR, and the Festival system [4] from University of Edinburgh. Within the general architecture of Festival, FlexTalk modules provide text analysis and produce the phonetic/prosodic specifications. Unit selection is based on CHATR's implementation of unit selection, but has been modified extensively. Synthesis is typically performed by HNM [10].

At AT&T, the unit selection approach has been extended by using half-phone units [5]. This, together with a a large and accurately labeled speech database, has resulted in very high-quality synthesis [2, 1].

4. CACHING CONCATENATION COSTS

This section outlines the motivation for pre-tabulating concatenation costs, how the "concatenation cost" cache was constructed, and most importantly, experiments to show that surprisingly good coverage is achieved with a very small subset of the possible concatenations.

4.1. Motivation for Using a Cache

The use of half-phones [5] greatly increases the flexibility of unit selection, allowing diphone-style synthesis with mid-phone transitions, and also phone-boundary transitions when warranted. Unfortunately, the use of half-phones doubles the number of "phone" specifications for which the search must find optimal acoustic units. If a sentence contains N phones, we must now deal with $2N$ half-phones. Each half-phone p_i gets some number n_i of candidate acoustic units, each of the correct phone identity and with suitable features. A naive Viterbi search will evaluate the concatenation cost of adjacent half-phone units $n_{i-1} * n_i$ times for each of the half-phone slots. This search is quadratic in the number of candidates, and so is inherently expensive for large databases.

Independent of pruning strategies to reduce the number of candidates considered, the use of half-phones requires twice the work of a comparable system using whole phones. The increase might have been even more extreme if the base units had been diphones, since the number of allowed neighbors at phone boundaries is not restricted to the same phone identity.

Our most successful system uses an 84,000 *half-phone* unit inventory from our speaker corpus. With such a large inventory and large number of possible unit combinations, we found the run-time search as described in [6] and implemented in [3] was the predominant time spent in the overall synthesis: a run-time profile of our Festival-based system showed that approximately 78% of the unit selection time and 51% of the entire synthesis time was spent in just computing the concatenation costs in the Viterbi search.

To make the synthesizer run faster, we initially considered tabulating the possible concatenation costs off-line and doing a simple lookup at runtime, rather than calling relatively expensive distance functions. The size of the search space quickly led us instead to consider caching a subset of the possible transitions.

4.2. The Search Space Of Concatenation Costs

With a database of 84,000 half-phones, there are nearly 1.8 billion possible unit pairs. Each of the 42,000 left-half units can be joined to each of the 42,000 right-half units, in effect yielding 1.76 billion diphones. Of course, some diphones don't occur naturally, but since the synthesis of an arbitrary phonetic sequence can be requested by a user who enters it manually, the system must do something reasonable with any phonetic input.

In addition, each of the 42,000 right-half units can be joined to each left-half unit of the same phone identity, e.g. /aa1/ with /aa2/, where the *1* suffix indicates the first, or left, half, and the *2* suffix the second, or right, half. The phone instances exist in unequal numbers, but on average there are 42,000 units / 48 phone types, or 875 units/type. Thus there are 42,000 $*$ 875, or about 36 million possible joins contributed by mid-phone transitions.

4.3. Building and Using a Cache

It was not practical for us to pre-tabulate all 1.8 billion possible join costs in any form that is usable at run-time. But our experiments showed that not all unit pairs need to be tabulated. To find the most frequently used transitions, we synthesized 10,000 Associated Press (AP) newswire stories . The first 1000 files from each of the first 10 months of 1995 were selected. This corresponds to the first 6 to 7 days of news across most of a year, and so may be somewhat more diverse than taking all the stories from a smaller interval. The full range of news topics was used, including international, business, financial, weather, etc. Duplicate stories, which are common as incremental revisions are submitted, had previously been removed from this corpus, leaving only the most recent revision of each story.

These 10,000 stories produced approximately 250,000 sentences containing 48 million half phones. From each sentence, we logged which units were ultimately selected by the unit selection module, along with the concatenation cost between each pair of units.

Fifty million (non-unique) unit pair instances were produced by unit selection for the full set of sentences, representing 1.2 million distinct pairs. These sentences and the unit pairs they produce we declared our *training corpus*. A separate set of 8000 AP sentences from the same time period served as our *test corpus*. The test corpus produced 1.5 million (non-unique) unit pair instances. Comparing these to the unit pairs in the training set, we found 99% of unit pair instances in the test set are present in the training set.

To determine how this coverage varies as a function of training set size, we constructed training subsets of various sizes by sampling the full training set and measured the coverage of unit pair instances from the test set in these training subsets. Figure 1 shows the results of this analysis. We see, for example, that a training subset of only five million unit pair instances already covers 95% of the test unit pair instances.

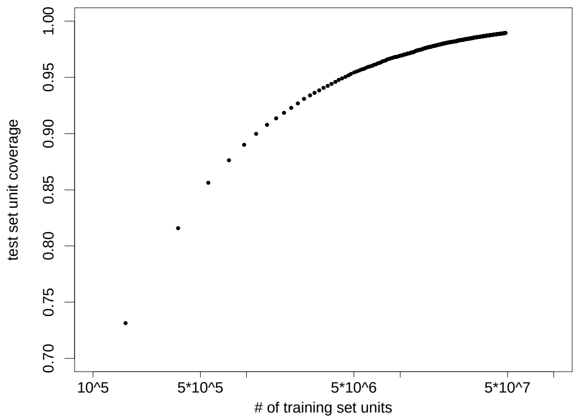

Figure 1: Percentage of (non-unique) unit pair instances from the test set also contained in the training subsets of various sizes.

Based on the good coverage shown from these statistics, we modified the unit selection to look up the join costs in a pre-computed table holding the 1.2 million unit training pairs. The 1.2 million unique unit pairs are much more manageable than 1.8 billion, and represent just 0.7% of the distinct possible concatenations. This tabulation sped up unit selection as a whole by $3\frac{1}{2}$ times, and unit selection also includes two other expensive operations: the target cost computations and the Viterbi search algorithm itself.

If a pair is not present in the table, a default large cost is returned. Alternatively, we could have called the actual cost functions, but absence from the cache already tells us the transition is unlikely to be chosen. The large default cost is sufficient to eliminate the join under any reasonable pruning, but does not disallow the transitions entirely. It is possible that situations will arise in which the Viterbi search considers two sets of candidates for which there are **no** cached transitions. Unit selection continues based on the default costs. The fact that all the concatenation costs are the same is mitigated by the individual unit costs, which do still vary, and provide a means to distinguish better units from worse.

We resynthesized the test set using this modified unit selection and found that 98.2% of units selected were the same as in the original unit selection. Further, even when a different unit was selected, it was on average close in cost to the original. One measure of this was that the average increase in cost per sentence was only 0.15% when switching to the pre-tabulated join costs. Figure 2 shows the per unit cost differences between the original and resynthesized cases for the 1.8% of the unit instances that do differ. By comparison, the average (uncached) test sentence cost is 203.162.

Informal listening tests confirm that using this concatenation cost subset produces synthesis that is nearly indistinguishable from using full join costs. We hope to have formal test results to confirm this later in the year.

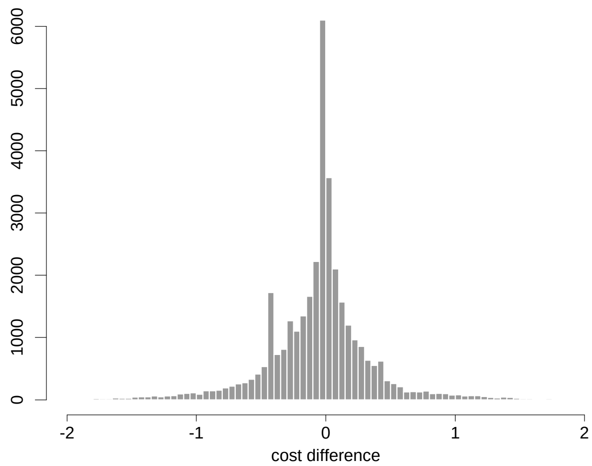

Figure 2: Per unit cost difference between the uncached and cached cases for the 1.8% unit instances that differ in the test set. By comparison, the average (uncached) test set sentence cost is 203.162.

4.4. Cache Implementation

The speed of finding concatenation costs is critical, even if a cache is used. A hash table was the obvious way to implement this cache given that the values used are *very* sparse compared to the total search space.

Hash tables provide a more compact representation of sparse data. The hash *key*, or keys, are converted by a *hash function* into an array index where the desired value should be found. In the case at hand, our hash function maps two unit numbers to a hash table entry containing the cost (plus some additional information).

An initial trial with a very general hashing library provided substantial speed-ups – $3\frac{1}{2}$ times faster for unit selection as a whole – but was still far from optimal. Hashing functions map keys to hash table elements in a fixed size array. In general, there is nothing to prevent two sets of keys mapping to the same table entry. The table entries themselves contain enough information to determine which values belong to which keys. Also note that many keys do not have values in the table – we're using a hash table precisely because the data is sparse.

To avoid the overhead associated with the general hashing routines, we implemented a *perfect hashing* for the cache lookup, which performed more than $2\frac{1}{2}$ times faster than the original hashing scheme. The new hash gives nearly optimal results. Perfect hashing is the use of hashing functions which produce no *collisions*, i.e. each table entry contains at most one hash item, thus avoid the overhead of examining multiple items. With such techniques membership queries can be performed in constant time.

Our perfect hashing algorithm is a refinement and extension of the technique presented by [11]. It consists of compacting a large matrix into a one-dimensional table by taking advantage of its sparseness. The algorithm also uses a waiting threshold to accelerate the construction of the table (see [7] for a full description and analysis of the algorithm).

Even more significant is the implementation of perfect hashing developed specifically for this application. Each of the two unit numbers is used as an array index, giving the hash table entry. One last test determines if the table entry matches the requesting keys (the entry could be empty). Lookup times are so small that it is difficult to imagine a faster implementation. By comparison, say the concatenation cost is based on the weighted sum of only two features. At a minimum, this involves two subtractions to find the feature differences, two multiplications by the weights, and one addition to get the aggregate cost. In practice, the full join computation involves multiple features and the feature costs involve logic that singles out special cases and non-integral features. The new hash incurs only two array references and an integer comparison.

Primarily as a result of this efficient cache, we are now able to run TTS in a small fraction of real-time on standard computer hardware (e.g. a 400 MHz Pentium II).

5. CONCLUSION

Experiments show that a cache of 1.2 million unit pair concatenation costs, from a possible set of 1.8 billion, provides 99% coverage of the unit pair instances in held-out test data. A cache constructed off-line from the 1.2 million unique unit pairs produces unit selection sequences which are 98.2% identical to units selected in the full search space. Moreover, the relative cost difference in those units that do change averages only 0.15%.

The speed of the unit selection module was increased by at least a factor of four, which is particularly significant since unit selection dominates the system runtime.

Informal listening tests confirm that audible output is nearly indistinguishable. We hope to have formal listening test results available later in the year.

We finally note that with this method we are now free to explore much more complex join cost functions, since these can now be computed off-line. Existing cost functions appear to have been chosen at least in part with an eye on fast computation. Without this constraint, we may be able to find join functions that give higher quality synthesis.

6. REFERENCES

1. M. Beutnagel, A. Conkie, J. Schroeter, Y. Styliano, and A. Syrdal. The at&t next-gen tts system. In *Proceedings of the Joint Meeting of ASA, EAA, and DAGA, Berlin, Germany*, March 1999.
2. M. Beutnagel, A. Conkie, and A. Syrdal. Diphone synthesis using unit selection. In *Proceedings of the 3rd International Workshop on Speech Synthesis, Jenolan Caves, Australia*, November 1998.
3. A. Black. Chatr, version 0.8, a generic speech synthesis system. System documentation., ATR Interpreting Telecommunications Laboratories, Kyoto, Japan, March 1996.
4. A. Black and P. Taylor. The festival speech synthesis system: System documentation. Technical Report HCRC/RT-83, Human Communications Research Centre, University of Edinburgh, Scotland, UK, January 1997.
5. A. Conkie. A robust unit selection system for speech synthesis. In *Joint Meeting of ASA/EAA/DAGA in Berlin, Germany*, March 1999.
6. A. Hunt and A. Black. Unit selection in a concatenative speech synthesis system. In *Proceedings of ICASSP-96, Atlanta, GA*, 1996.
7. M. Mohri. *Finite-state transducers*. The MIT Press, 1999, to appear.
8. J. Olive. Rule synthesis by speech from diadic units. In *ICASSP'77*, pages 568–570, 1997.
9. R. Sproat, J. Hirschberg, and D. Yarowsky. A corpus-based synthesizer. In *Proceedings of ICSLP-92, Banff, Canada*, October 1992.
10. Y. Stylianou, T. Dutoit, and J. Schroeter. Diphone concatenation using a harmonic plus noise model of speech. In *Proceedings of Eurospeech'97, Rhodes, Greece*, 1997.
11. R. E. Tarjan and A. C.-C. Yao. Storing a Sparse Table. *Communications of the ACM*, 22: 11:606–611, 1979.