Storing the Web in Memory: Space Efficient Language Models with Constant Time Retrieval

David Guthrie
Computer Science Department
University of Sheffield
D.Guthrie@dcs.shef.ac.uk

Mark Hepple
Computer Science Department
University of Sheffield
M.Hepple@dcs.shef.ac.uk

Abstract

We present three novel methods of compactly storing very large n-gram language models. These methods use substantially less space than all known approaches and allow n-gram probabilities or counts to be retrieved in constant time, at speeds comparable to modern language modeling toolkits. Our basic approach generates an *explicit* minimal perfect hash function, that maps all n-grams in a model to distinct integers to enable storage of associated values. Extensions of this approach exploit distributional characteristics of n-gram data to reduce storage costs, including variable length coding of values and the use of *tiered* structures that partition the data for more efficient storage. We apply our approach to storing the full Google Web1T n-gram set and all 1-to-5 grams of the Gigaword newswire corpus. For the 1.5 billion n-grams of Gigaword, for example, we can store full count information at a cost of 1.66 bytes per n-gram (around 30% of the cost when using the current state-of-the-art approach), or quantized counts for 1.41 bytes per n-gram. For applications that are tolerant of a certain class of relatively innocuous errors (where unseen n-grams may be accepted as rare n-grams), we can reduce the latter cost to below 1 byte per n-gram.

1 Introduction

The availability of very large text collections, such as the Gigaword corpus of newswire (Graff, 2003), and the Google Web1T 1-5gram corpus (Brants and Franz, 2006), have made it possible to build models incorporating counts of billions of n-grams. The storage of these language models, however, presents serious problems, given both their size and the need to provide rapid access. A prevalent approach for language model storage is the use of compact trie structures, but these structures do not scale well and require space proportional to both to the number of n-grams and the vocabulary size. Recent advances (Talbot and Brants, 2008; Talbot and Osborne, 2007b) involve the development of Bloom filter based models, which allow a considerable reduction in the space required to store a model, at the cost of allowing some limited extent of false positives when the model is queried with previously unseen n-grams. The aim is to achieve sufficiently compact representation that even very large language models can be stored totally within memory, avoiding the latencies of disk access. These Bloom filter based models exploit the idea that it is not actually necessary to *store* the n-grams of the model, as long as, when queried with an n-gram, the model returns the correct count or probability for it. These techniques allow the storage of language models that no longer depend on the size of the vocabulary, but only on the number of n-grams.

In this paper we give three different models for the efficient storage of language models. The first structure makes use of an explicit perfect hash function that is *minimal* in that it maps n keys to integers in the range 1 to n. We show that by using a minimal perfect hash function and exploiting the distributional characteristics of the data we produce n-gram models that use less space than all know approaches with no reduction in speed. Our two further models achieve even more compact storage while maintaining constant time access by using variable length coding to compress the n-grams values and by using tiered hash structures to parti-

Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, pages 262–272,
MIT, Massachusetts, USA, 9-11 October 2010.

tion the data into subsets requiring different amounts of storage. This combination of techniques allows us, for example, to represent the *full* count information of the Google Web1T corpus (Brants and Franz, 2006) (where count values range up to 95 billion) at a cost of just 2.47 bytes per n-gram (assuming 8-bit fingerprints, to exclude false positives) and just 1.41 bytes per n-gram if we use 8-bit quantization of counts. These costs are 36% and 57% respectively of the space required by the Bloomier Filter approach of Talbot and Brants (2008). For the Gigaword dataset, we can store full count information at a cost of only 1.66 bytes per n-gram. We report empirical results showing that our approach allows a look-up rate which is comparable to existing modern language modeling toolkits, and much faster than a competitor approach for space-efficient storage. Finally, we propose the use of *variable length fingerprinting* for use in contexts which can tolerate a higher rate of 'less damaging' errors. This move allows, for example, the cost of storing a quantized model to be reduced to 1 byte per n-gram or less.

2 Related Work

A range of *lossy* methods have been proposed, to reduce the storage requirements of LMs by discarding information. Methods include the use of entropy pruning techniques (Stolcke, 1998) or clustering (Jelinek et al., 1990; Goodman and Gao, 2000) to reduce the number of n-grams that must be stored. A key method is *quantization* (Whittaker and Raj, 2001), which reduces the value information stored with n-grams to a limited set of *discrete* alternatives. It works by grouping together the values (probabilities or counts) associated with n-grams into clusters, and replacing the value to be stored for each n-gram with a code identifying its value's cluster. For a scheme with n clusters, codes require $\log_2 n$ bits. A common case is 8-bit quantization, allowing up to 256 distinct 'quantum' values. Different methods of dividing the range of values into clusters have been used, e.g. Whittaker and Raj (2001) used the Lloyd-Max algorithm, whilst Federico and Bertoldi (2006) use the simpler Binning method to quantize probabilities, and show that the LMs so produced out-perform those produced using the Lloyd-Max method on a phrase-based machine translation task. Binning partitions the range of values into regions that are uniformly populated, i.e. producing clusters that contain the same number of unique values. Functionality to perform uniform quantization of this kind is provided as part of various LM toolkits, such as IRSTLM. Some of the empirical storage results reported later in the paper relate to LMs recording n-gram *count* values which have been quantized using this uniform binning approach. In the rest of this section, we turn to look at some of the approaches used for *storing* language models, irrespective of whether lossy methods are first applied to reduce the size of the model.

2.1 Language model storage using Trie structures

A widely used approach for storing language models employs the *trie* data structure (Fredkin, 1960), which compactly represents sequences in the form of a *prefix tree*, where each step down from the root of the tree adds a new element to the sequence represented by the nodes seen so far. Where two sequences share a prefix, that common prefix is jointly represented by a single node within the trie. For language modeling purposes, the steps through the trie correspond to *words* of the vocabulary, although these are in practice usually represented by 24 or 32 bit integers (that have been uniquely assigned to each word). Nodes in the trie corresponding to *complete* n-grams can store other information, e.g. a probability or count value. Most modern language modeling toolkits employ some version of a trie structure for storage, including SRILM (Stolcke, 2002), CMU toolkit (Clarkson and Rosenfeld, 1997), MITLM (Hsu and Glass, 2008), and IRSTLM (Federico and Cettolo, 2007) and implementations exist which are very compact (Germann et al., 2009). An advantage of this structure is that it allows the stored n-grams to be enumerated. However, although this approach achieves a compact of representation of sequences, its memory costs are still such that very large language models require very large storage space, far more than the Bloom filter based methods described shortly, and far more than might be held in memory as a basis for more rapid access. The memory costs of such models have been addressed using compression methods, see Harb et al. (2009), but such extensions of the

approach present further obstacles to rapid access.

2.2 Bloom Filter Based Language Models

Recent randomized language models (Talbot and Osborne, 2007b; Talbot and Osborne, 2007a; Talbot and Brants, 2008; Talbot and Talbot, 2008; Talbot, 2009) make use of Bloom filter like structures to map n-grams to their associated probabilities or counts. These methods store language models in relatively little space by not actually keeping the n-gram key in the structure and by allowing a small probability of returning a false positive, i.e. so that for an n-gram that is not in the model, there is a small risk that the model will return some random probability instead of correctly reporting that the n-gram was not found. These structures do not allow enumeration over the n-grams in the model, but for many applications this is not a requirement and their space advantages make them extremely attractive. Two major approaches have been used for storing language models: Bloom Filters and Bloomier Filters. We give an overview of both in what follows.

2.2.1 Bloom Filters

A Bloom filter (Bloom, 1970) is a compact data structure for membership queries, i.e. queries of the form "Is this key in the Set?". This is a weaker structure than a dictionary or hash table which also associates a value with a key. Bloom filters use well below the information theoretic lower bound of space required to actually store the keys and can answer queries in O(1) time. Bloom filters achieve this feat by allowing a small probability of returning a false positive. A Bloom filter stores a set S of n elements in a bit array B of size m. Initially B is set to contain all zeros. To store an item x from S in B we compute k random independent hash functions on x that each return a value in the range $[0\,..\,m-1]$. These values serve as indices to the bit array B and the bits at those positions are set to 1. We do this for all elements in S, storing to the same bit array. Elements may hash to an index in B that has already been set to 1 and in this case we can think of these elements as "sharing" this bit. To test whether set S contains a key w, we run our k hash functions on w and check if all those locations in B are set to 1. If $w \in S$ then the bloom filter will always declare that w belongs to S, but if $x \notin S$ then the filter can only say with high probability that w is not in S. This error rate depends on the number of k hash functions and the ratio of m/n. For instance with $k = 3$ hash functions and a bit array of size $m = 20n$, we can expect to get a false positive rate of 0.0027.

Talbot and Osborne (2007b) and Talbot and Osborne (2007a) adapt Bloom filters to the requirement of storing values for n-grams by concatenating the key (n-gram) and value to form a single item that is inserted into the filter. Given a quantization scheme allowing values in the range $[1\,..\,V]$, a quantized value v is stored by inserting into the filter all pairings of the n-gram with values from 1 up to v. To retrieve the value for a given key, we serially probe the filter for pairings of the key with each value from 1 upwards, until the filter returns false. The last value found paired with the key in the filter is the value returned. Talbot and Osborne use a simple logarithmic quantization of counts that produce limited quantized value ranges, where most items will have values that are low in the range, so that the serial look-up process will require quite a low number of steps *on average*. For alternative quantization schemes that involve greater value ranges (e.g. the 256 values of a uniform 8-bit scheme) and/or distribute n-grams more evenly across the quantized values, the average number of look-up steps required will be higher and hence the speed of access per n-gram accordingly lower. In that case also, the requirement of inserting n-grams more than once in the filter (i.e. with values from 1 up to the actual value v being stored) could substantially reduce the space efficiency of the method, especially if low false positive rates are required, e.g. the case $k = 3, m = 20n$ produces a false positive rate of 0.0027, as noted above, but in a situation where 3 key-value items were being stored per n-gram on average, this error rate would in fact require a storage cost of 60 bits per original n-gram.

2.2.2 Bloomier Filters

More recently, Talbot and Brants (2008) have proposed an approach to storing large language models which is based on the *Bloomier Filter* technique of Chazelle et al. (2004). Bloomier Filters generalize the Bloom Filter to allow values for keys to be stored in the filter. To test whether a given key is present in a populated Bloomier filter, we apply k hash functions to the key and use the results as

indices for retrieving the data stored at k locations within the filter, similarly to look-up in a Bloom filter. In this case, however, the data retrieved from the filter consists of k *bit vectors*, which are combined with a fingerprint of the key, using *bitwise XOR*, to return the stored value. The risk of false positives is managed by making incorporating a fingerprint of the n-gram, and by making bit vectors longer than the minimum length required to store values. These additional *error bits* have a fairly predictable impact on error rates, i.e. with e error bits, we anticipate the probability of falsely construing an unseen n-gram as being stored in the filter to be 2^{-e}. The algorithm required to correctly populate the Bloomier filter with stored data is complicated, and we shall not consider its details here. Nevertheless, when using v bits to represent values and e bits for error detection, this approach allows a language model to be stored at a cost of is $1.23 \cdot (v + e)$ bits per n-gram.

3 Single Minimal Perfect Hash Ranking Approach

We first describe our basic structure we call Single Minimal Perfect Hash Rank (S-MPHR) that is more compact than that of Talbot and Brants (2008) while still keeping a constant look up time. In the next two sections we describe variations on this model to further reduce the space required while maintaining a constant look up time. The S-MPHR structure can be divided into 3 parts as shown in Figure 1: Stage 1 is a minimal perfect hash function; Stage 2 is a fingerprint and rank array; and Stage 3 is a unique value array. We discuss each stage in turn.

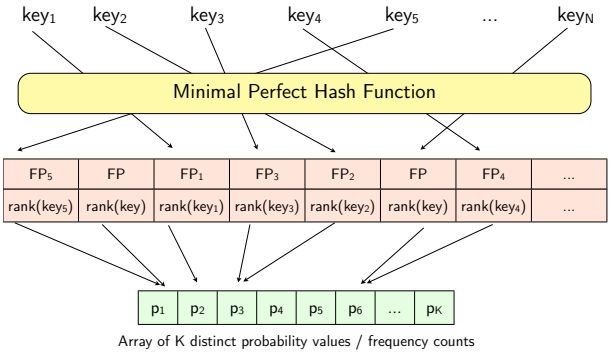

Figure 1: The Single MPHR structure

3.1 Minimal Perfect Hash Function

The first part of the structure is a *minimal* perfect hash function that maps every n-gram in the training data to a distinct integer in the range 0 to $N - 1$, where N is the total number of n-grams to store. We use these integers as indices into the array of Stage 2 of our structure.

We use the *Hash, displace, and compress (CHD)* (Belazzougui et al., 2009) algorithm to generate a minimal perfect hash function that requires 2.07 bits per key and has $O(1)$ access. The algorithm works as follows. Given a set S that contains $N = |S|$ keys (in our case n-grams) that we wish to map to integers in the range 0 to $N - 1$, so that every key maps to a distinct integer (no collisions).

The first step is to use a hash function $g(x)$, to map every key to a bucket B in the range 0 to r. (For this step we use a simple hash function like the one used for generating fingerprints in the pervious section.)

$$B_i = \{x \in S | g(x) = i\} \; 0 \leq i \leq r$$

The function $g(x)$ is not perfect so several keys can map to the same bucket. Here we choose $r \leq N$, so that the number of buckets is less than or equal to the number of keys (to achieve 2.07 bits per key we use $r = \frac{N}{5}$, so that the average bucket size is 5). The buckets are then sorted into descending order according to the number of keys in each bucket $|B_i|$.

For the next step, a bit array, T, of size N is initialized to contain all zeros $T[0 \ldots N - 1]$. This bit array is used during construction to keep track of which integers in the range 0 to $N - 1$ the minimal perfect hash has already mapped keys to. Next we must assume we have access to a family of random and independent hash functions $h_1, h_2, h_3, \ldots$ that can be accessed using an integer index. In practice it sufficient to use functions that behave similarly to fully random independent hash functions and Belazzougui et al. (2009) demonstrate how such functions can be generated easily by combining two simple hash functions.

Next is the "displacement" step. For each bucket, in the sorted order from largest to smallest, they search for a random hash function that maps all elements of the bucket to values in T that are currently set to 0. Once this function has been found those

positions in T are set to 1. So, for each bucket B_i, it is necessary to iteratively try hash functions, h_ℓ for $\ell = 1, 2, 3, \ldots$ to hash every element of B_i to a distinct index j in T that contains a zero.

$$\{h_\ell(x)|x \in B_i\} \cap \{j|T[j] = 1\} = \emptyset$$

where the size of $\{h_\ell(x)|x \in B_i\}$ is equal to the size of B_i. When such a hash function is found we need only to store the index, ℓ, of the successful function in an array σ and set $T[j] = 1$ for all positions j that h_ℓ hashed to. Notice that the reason the largest buckets are handled first is because they have the most elements to displace and this is easier when the array T contains more empty positions (zeros).

The final step in the algorithm is to compress the σ array (which has length equal to the number of buckets $|B|$), retaining $O(1)$ access. This compression is achieved using simple variable length encoding with an index array (Fredriksson and Nikitin, 2007).

3.2 Fingerprint and Rank Array

The hash function used in Stage 1 is perfect, so it is guaranteed to return unique integers for seen n-grams, but our hash function will also return integer values in the range 0 to $N - 1$ for n-grams that have not been seen before (were not used to build the hash function). To reduce the probability of these unseen n-grams giving false positives results from our model we store a fingerprint of each n-gram in Stage 2 of our structure that can be compared against the fingerprints of unseen n-grams when queried. If these fingerprints of the queried n-gram and the stored n-gram do not match then the model will correctly report that the n-gram has not been seen before. The size of this fingerprint determines the rate of false positives. Assuming that the fingerprint is generated by a random hash function, and that the returned integer of an *unseen* key from the MPH function is also random, expected false positive rate for the model is the same as the probability of two keys randomly hashing to the same value, $\frac{1}{2^m}$, where m is the number of bits of the fingerprint. The fingerprint can be generated using any suitably random hashing algorithm. We use Austin Appleby's Murmurhash2[1] implementation to fingerprint each n-gram and then store the m highest order bits. Stage 2 of the MPHR structure also stores a *rank* for every n-gram along with the fingerprint. This rank is an index into the array of Stage 3 of our structure that holds the unique values associated with any n-gram.

[1] http://murmurhash.googlepages.com/

3.3 Unique Value Array

We describe our storage of the values associated with n-grams in our model assuming we are storing frequency "counts" of n-grams, but it applies also to storing quantized probabilities. For every n-gram, we store the 'rank' of the frequency count $r(key)$, ($r(key) \in [0...R-1]$) and use a separate array in Stage 3 to store the frequency count value. This is similar to quantization in that it reduces the number of bits required for storage, but unlike quantization it does not require a loss of any information. This was motivated by the sparsity of n-gram frequency counts in corpora in the sense that if we take the lowest n-gram frequency count and the highest n-gram frequency count then most of the integers in that range do not occur as a frequency count of any n-grams in the corpus. For example in the Google Web1T data, there are 3.8 billion unique n-grams with frequency counts ranging from 40 to 95 Billion yet these n-grams only have 770 thousand distinct frequency counts (see Table 2). Because we only store the frequency rank, to keep the precise frequency information we need only $\lceil \log_2 K \rceil$ bits per n-gram, where K is the number of distinct frequency counts. To keep all information in the Google Web1T data we need only $\lceil \log_2 771058 \rceil = 20$ bits per n-gram. Rather than the bits needed to store the maximum frequency count associated with an n-gram, $\lceil \log_2 \text{maxcount} \rceil$, which for Google Web1T would be $\lceil \log_2 95119665584 \rceil = 37$ bits per n-gram.

	unique n-grams	maximum n-gram frequency count	unique counts
1gm	1,585,620	71,363,822	16,896
2gm	55,809,822	9,319,466	20,237
3gm	250,928,598	829,366	12,425
4gm	493,134,812	231,973	6,838
5gm	646,071,143	86,943	4,201
Total	1,447,529,995	71,363,822	60,487

Table 1: n-gram frequency counts from Gigaword corpus

	unique n-grams	maximum n-gram frequency count	unique counts
1gm	13,588,391	95,119,665,584	238,592
2gm	314,843,401	8,418,225,326	504,087
3gm	977,069,902	6,793,090,938	408,528
4gm	1,313,818,354	5,988,622,797	273,345
5gm	1,176,470,663	5,434,417,282	200,079
Total	3,795,790,711	95,119,665,584	771,058

Table 2: n-gram frequency counts from Google Web1T corpus

3.4 Storage Requirements

We now consider the storage requirements of our S-MPHR approach, and how it compares against the Bloomier filter method of Talbot and Brants (2008). To start with, we put aside the gains that can come from using the ranking method, and instead consider just the costs of using the CHD approach for storing any language model. We saw that the storage requirements of the Talbot and Brants (2008) Bloomier filter method are a function of the number of n-grams n, the bits of data d to be stored per n-gram (with $d = v + e$: v bits for value storage, and e bits for error detection), and a multiplying factor of 1.23, giving an overall cost of $1.23d$ bits per n-gram. The cost for our basic approach is also easily computed. The explicit minimal PHF computed using the CHD algorithm brings a cost of 2.07 bits per n-gram for the PHF itself, and so the comparable overall cost to store a S-MPHR model is $2.07 + d$ bits per n-gram. For small values of d, the Bloomier filter approach has the smaller cost, but the 'break-even' point occurs when $d = 9$. When d is greater than 9 bits (as it usually will be), our approach wins out, being up to 18% more efficient.

The benefits that come from using the ranking method (Stage 3), for compactly storing count values, can only be evaluated in relation to the distributional characteristics specific corpora, for which we show results in Section 6.

4 Compressed MPHR Approach

Our second approach, called Compressed MPHR, further reduces the size of the model whilst maintaining O(1) time to query the model. Most compression techniques work by exploiting the redundancy in data. Our fingerprints are unfortunately random sequences of bits, so trying to compress these is fruitless, but the ranks associated with n-grams contain much redundancy and so are likely to compress well. We therefore modify our original architecture to put the ranks and fingerprints into separate arrays, of which the ranks array will be compressed, as shown in Figure 2.

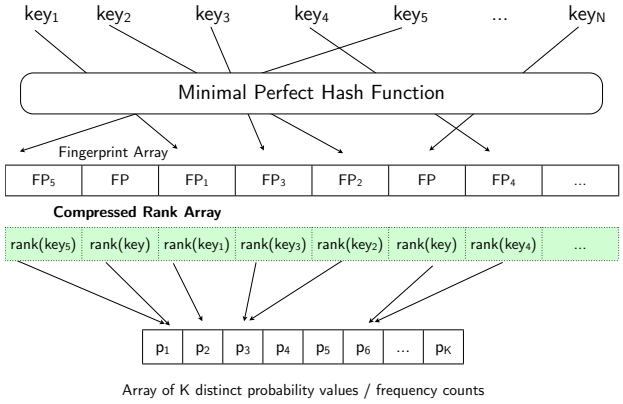

Figure 2: Compressed MPHR structure

Much like the final stage of the CHD minimal perfect hash algorithm we employ a random access compression algorithm of Fredriksson and Nikitin (2007) to reduce the size required by the array of ranks. This method allows compression while retaining O(1) access to query the model.

The first step in the compression is to encode the ranks array using a dense variable length coding. This coding works by assigning binary codes with different lengths to each number in the rank array, based on how frequent that number occurs. Let $s_1, s_2, s_3, \ldots, s_K$ be the ranks that occur in the rank array sorted by there frequency. Starting with most frequent number in the rank array (clearly 1 is the most common frequency count in the data unless it has been pruned) s_1 we assign it the bit code 0 and then assign s_2 the bit code 1, we then proceed by assigning bit codes of two bits, so s_3 is assigned 00, $s4$ is assigned 01, etc. until all two bit codes are used up. We then proceed to assign 3 bit codes and so on. All of the values from the rank array are coded in this form and concatenated to form a large bit vector retaining their original ordering. The length in bits for the ith number is thus $\lfloor \log_2 (i+2) \rfloor$ and so the number of bits required for the whole variable length coded rank array is: $b = \sum_{i=0}^{K} f(s_i) \lfloor \log_2 (i+2) \rfloor$. Where $f()$ gives the frequency that the rank occurs

and K is the total number of distinct ranks. The code for the ith number is the binary representation with length $\lfloor \log_2 (i+2) \rfloor$ of the number obtained using the formula:

$$\text{code} = i + 2 - 2^{\lfloor \log_2 (i+2) \rfloor}$$

This variable length coded array is not useful by itself because we do not know where each number begins and ends, so we also store an index array hold this information. We create an additional bit array D of the same size b as the variable length coded array that simply contains ones in all positions that a code begins in the rank array and zeros in all other positions. That is the ith rank in the variable length coded array occurs at position $\text{select}_1(D, i)$, where select_1 gives the position of the ith one in the array. We do not actually store the D array, but instead we build a more space efficient structure to answer select_1 queries. Due the distribution of n-gram frequencies, the D array is typically dense in containing a large proportion of ones, so we build a *rank9sel* dictionary structure (Vigna, 2008) to answer these queries in constant time. We can use this structure to identify the ith code in our variable length encoded rank array by querying for its starting position, $\text{select}_1(D, i)$, and compute its length using its ending position, $\text{select}_1(D, i+1) - 1$. The code and its length can then be decoded to obtain the original rank:

$$\text{rank} = \text{code} + 2^{(\text{length in bits})} - 2$$

5 Tiered MPHR

In this section we describe an alternative route to extending our basic S-MPHR model to achieve better space efficiency, by using multiple hash stores. The method exploits distributional characteristics of the data, i.e. that lower rank values (those assigned to values shared by very many n-grams) are sufficient for representing the value information of a disproportionately large subset of the data. For the Google Web 1T data, for example, we find that the first 256 ranks account for nearly 85% of distinct n-grams, so if we could store ranks for these n-grams using only the 8 bits they require, whilst allowing perhaps 20 bits per n-gram for the remaining 15%, we would achieve an average of just under 10 bits per n-gram to store all the rank values.

To achieve this gain, we might *partition* the n-gram data into subsets requiring different amounts of space for value storage, and put these subsets in separate MPHRs, e.g. for the example just mentioned, with two MPHRs having 8 and 20 bit value storage respectively. Partitioning to a larger number h of MPHRs might further reduce this average cost. This simple approach has several problems. Firstly, it potentially requires a *series* of look up steps (i.e. up to h) to retrieve the value for any n-gram, with *all* hashes needing to be addressed to determine the unseen status of an unseen n-gram. Secondly, multiple look ups will produce a *compounding* of error rates, since we have up to h opportunities to falsely construe an unseen n-gram as seen, or to construe a seen n-gram as being stored in the wrong MPHR and so return an incorrect count for it.

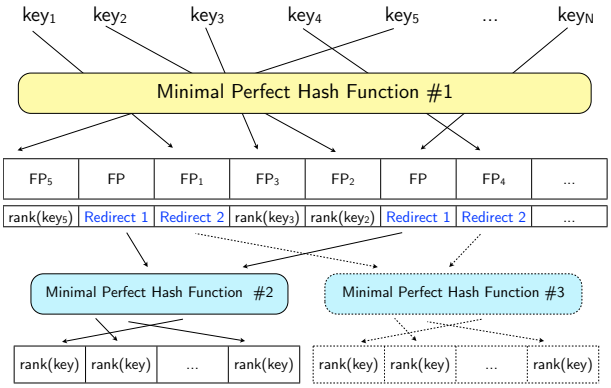

Figure 3: Tiered minimal perfect hash data structure

We will here explore an alternative approach that we call *Tiered* MPHR, which avoids this compounding of errors, and which limits the number of looks ups to a maximum of 2, irrespective of how many hashes are used. This approach employs a single *top-level* MPHR which has the full set of n-grams for its key-set, and stores a fingerprint for each. In addition, space is allocated to store rank values, but with some possible values being reserved to indicate *redirection* to other *secondary* hashes where values can be found. Each secondary hash has a minimal perfect hash function that is computed only for the n-grams whose values it stores. Secondary hashes do *not* need to record fingerprints, as fingerprint testing is done in the top-level hash.

For example, we might have a configuration of

three hashes, with the top-level MPHR having 8-bit storage, and with secondary hashes having 10 and 20 bit storage respectively. Two values of the 8-bit store (e.g. 0 and 1) are reserved to indicate redirection to the specific secondary hashes, with the remaining values $(2\,..\,255)$ representing ranks 1 to 254. The 10-bit secondary hash can store 1024 different values, which would then represent ranks 255 to 1278, with all ranks above this being represented in the 20-bit hash. To look up the count for an n-gram, we begin with the top-level hash, where fingerprint testing can immediately reject unseen n-grams. For some seen n-grams, the required rank value is provided directly by the top-level hash, but for others a redirection value is returned, indicating precisely the secondary hash in which the rank value will be found by simple look up (with no additional fingerprint testing). Figure 3 gives a generalized presentation of the structure of tiered MPHRs. Let us represent a configuration for a tiered MPHR as a sequence of bit values for their value stores, e.g. `(8,10,20)` for the example above, or $H = (b_1, \ldots .b_h)$ more generally (with b_1 being the top-level MPHR).

The overall memory cost of a particular configuration depends on distributional characteristics of the data stored. The top-level MPHR of configuration $(b_1, \ldots .b_h)$ stores all n-grams in its key-set, so its memory cost is calculated as before as $N \times (2.07 + m + b_1)$ (m the fingerprint size). The top-level MPHR must reserve $h - 1$ values for redirection, and so covers ranks $[1\,..\,(2^{b_1} - h + 1)]$. The second MPHR then covers the next 2^{b_2} ranks, starting at $(2^{b_1} - h + 2)$, and so on for further secondary MPHRs. This range of ranks determines the proportion μ_i of the overall n-gram set that each secondary MPHR b_i stores, and so the memory cost of each secondary MPHR is $N \times \mu_i \times (2.07 + b_i)$. The optimal T-MPHR configuration for a given data set is easily determined from distributional information (of the coverage of each rank), by a simple search.

6 Results

In this section, we present some results comparing the performance of our new storage methods to some of the existing methods, regarding the costs of storing LMs, and regarding the data access speeds that alternative systems allow.

Method	Gigaword		Web1T	
	full	quantized	full	quantized
Bloomier	6.00	3.08	7.53	3.08
S-MPHR	3.76	2.76	4.26	2.76
C-MPHR	2.19	2.09	3.40	2.09
T-MPHR	2.16	1.91	2.97	1.91

Table 3: Space usage in bytes/ngram using **12-bit** fingerprints and storing all 1 to 5 grams

Method	Gigaword		Web1T	
	full	quantized	full	quantized
Bloomier	5.38	2.46	6.91	2.46
S-MPHR	3.26	2.26	3.76	2.26
C-MPHR	1.69	1.59	2.90	1.59
T-MPHR	1.66	1.41	2.47	1.41

Table 4: Space usage in bytes/n-gram using **8-bit** fingerprints and storing all 1 to 5 grams

6.1 Comparison of memory costs

To test the effectiveness of our models we built models storing n-grams and full frequency counts for both the Gigaword and Google Web1T corpus storing all 1,2,3,4 and 5 grams. These corpora are very large, e.g. the Google Web1T corpus is 24.6GB when gzip compressed and contains over 3.7 billion n-grams, with frequency counts as large as 95 billion, requiring at least 37 bits to be stored. Using the Bloomier algorithm of Talbot and Brants (2008) with 37 bit values and 12 bit fingerprints would require 7.53 bytes/n-gram, so we would need 26.63GB to store a model for the entire corpus.

In comparison, our S-MPHR method requires only 4.26 bytes per n-gram to store full frequency count information and stores the entire Web1T corpus in just **15.05GB** or **57%** of the space required by the Bloomier method. This saving is mostly due to the ranking method allowing values to be stored at a cost of only 20 bits per n-gram. Applying the same rank array optimization to the Bloomier method significantly reduces its memory requirement, but S-MPHR still uses only 86% of the space that the Bloomier approach requires. Using T-MPHR instead, again with 12-bit fingerprints, we can store full counts for the Web 1T corpus in 10.50GB, which is small enough to be held in memory on many modern machines. Using 8-bit fingerprints, T-

Method	bytes/ ngram
SRILM Full, Compact	33.6
IRSTLM, 8-bit Quantized	9.1
Bloomier 12bit fp, 8bit Quantized	3.08
S-MPHR 12bit fp, 8bit Quantized	2.76
C-MPHR 12bit fp, 8bit Quantized	2.09
T-MPHR 12bit fp, 8bit Quantized	**1.91**

Table 5: Comparison between approaches for storing all 1 to 5 grams of the Gigaword Corpus

Test	Time (hr :min:sec)	Speed queries/sec
C-MPHR	00 : 01 : 50	507362
IRSTLM	00 : 02 : 12	422802
SRILM	00 : 01 : 29	627077
randLM	00 : 27 : 28	33865
MySQL 5	29 : 25 : 01	527

Table 6: Look-up speed performance comparison for C-MPHR and several other LM storage methods

MPHR can store this data in just 8.74GB.

Tables 3, 4 and 5 show results for all methods[2] on both corpora, for storing full counts, and for when 8-bit binning quantization of counts is used.

6.2 Access speed comparisons

The three models we present in this paper perform queries in O(1) time and are thus asymptotically optimal, but this does not guarantee they perform well in practice, therefore in this section we measure query speed on a large set of n-grams and compare it to that of modern language modeling toolkits. We build a model of all unigrams and bigrams in the Gigaword corpus (see Table 1) using the C-MPHR method, SRILM (Stolcke, 2002), IRSTLM (Federico and Cettolo, 2007), and randLM[3] (Talbot and Osborne, 2007a) toolkits. RandLM is a modern language modeling toolkit that uses Bloom filter based structures to store large language models and has been integrated so that it can be used as the language model storage for the Moses statistical machine translation system (Koehn et al., 2007). We use randLM with the *BloomMap* (Talbot and Talbot, 2008) storage structure option with 8 bit quantized values and an error rate equivalent to using 8 bit fingerprints (as recommended in the Moses documentation). All methods are implemented in C++ and are run on a machine with 2.80GHz Intel Xeon E5462 processor and 64 GB of RAM. In addition we show a comparison to using a modern database, MySQL 5.0, to store the same data. We measure the speed of querying all models for the 55 million distinct bigrams that occur in the Gigaword, these results are shown in Table 6. Unsurprisingly all methods perform significantly faster than using a database as they build models that reside completely in memory. The C-MPHR method tested here is slower than both S-MPHR and T-MPHR models due to the extra operations required for access to the variable length encoded array yet still performs similarly to SRILM and IRSTLM and is 14.99 times faster than using randLM.

7 Variable Length Fingerprints

To conclude our presentation of new methods for space-efficient language model storage, we suggest an additional possibility for reducing storage costs, which involves using different sizes of fingerprint for different n-grams. Recall that the only errors allowed by our approach are false-positives, i.e. where an unseen n-gram is falsely construed as being part of the model and a value returned for it. The idea behind using different sizes of fingerprint is that, intuitively, some possible errors seem worse than others, and in particular, it seems likely to be less damaging if we falsely construe an unseen n-gram as being a seen n-gram that has a low count or probability than as being one with a high count or probability.

False positives arise when our perfect hashing method maps an unseen n-gram to position where the stored n-gram fingerprint happens to coincide with that computed for the unseen n-gram. The risk of this occurring is a simple function of the size of fingerprints. To achieve a scheme that admits a higher risk of less damaging errors, but enforces a lower risk of more damaging errors, we need only store shorter fingerprints for n-grams in our model that have low counts or probabilities, and longer fingerprints for n-grams with higher values. This

[2] All T-MPHR results are for optimal configurations: Gigaword full: `(2,3,16)`, Gigaword quant: `(1,8)`, Web1T full: `(8,6,7,8,9,10,13,20)`, Web1T quant: `(1,8)`.

[3] `http://sourceforge.net/projects/randlm/`

idea could be implemented in different ways, e.g. by storing fingerprints of different lengths contiguously within a bit array, and constructing a 'selection structure' of the kind described in Section 4 to allow us to locate a given fingerprint within the bit array.

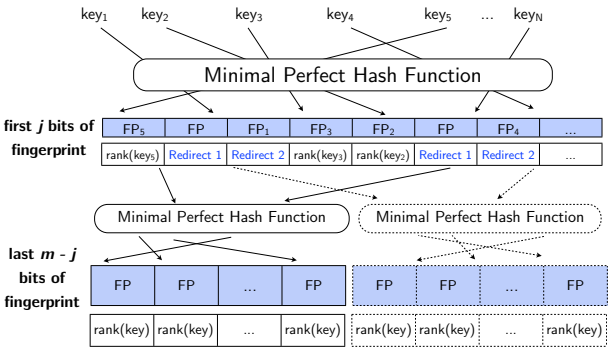

Figure 4: Variable length fingerprint T-MPHR structure using j bit fingerprints for the n-grams which are most rare and m bit fingerprints for all others.

We here instead consider an alternative implementation, based on the use of tiered structures. Recall that for T-MPHR, the top-level MPHR has all n-grams of the model as keys, and stores a fingerprint for each, plus a value that may represent an n-gram's count or probability, or that may redirect to a second-level hash where that information can be found. Redirection is done for items with higher counts or probabilities, so we can achieve lower error rates for precisely these items by storing *additional* fingerprint information for them in the second-level hash (see Figure 4). For example, we might have a top-level hash with only 4-bit fingerprints, but have an additional 8-bits of fingerprint for items also stored in a second-level hash, so there is quite a high risk (close to $\frac{1}{16}$) of returning a low count for an unseen n-gram, but a much lower risk of returning any higher count. Table 7 applies this idea to storing full and quantized counts of the Gigaword and Web 1T models, when fingerprints in the top-level MPHR have sizes in the range 1 to 6 bits, with the fingerprint information for items stored in secondary hashes being 'topped up' to 12 bits. This approach achieves storage costs of around 1 byte per n-gram or less for the quantized models.

Bits in lowest fingerprint	Gigaword Quantized	Web1T Quantized	Gigaword All	Web1T All
1	0.55	0.55	1.00	1.81
2	0.68	0.68	1.10	1.92
3	0.80	0.80	1.21	2.02
4	0.92	0.92	1.31	2.13
5	1.05	1.04	1.42	2.23
6	1.17	1.17	1.52	2.34

Table 7: Bytes per fingerprint for T-MPHR model using 1 to 6 bit fingerprints for rarest n-grams and 12 bit (in total) fingerprints for all other n-grams. (All configurations are as in Footnote 2.)

8 Conclusion

We have presented novel methods of storing large language models, consisting of billions of n-grams, that allow for quantized values or frequency counts to be accessed quickly and which require far less space than all known approaches. We show that it is possible to store all 1 to 5 grams in the Gigaword corpus, with full count information at a cost of just 1.66 bytes per n-gram, or with quantized counts for just 1.41 bytes per n-gram. We have shown that our models allow n-gram look-up at speeds comparable to modern language modeling toolkits (which have much greater storage costs), and at a rate approximately 15 times faster than a competitor approach for space-efficient storage.

References

Djamal Belazzougui, Fabiano Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. *Algorithms - ESA 2009*, pages 682–693.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Thorsten Brants and Alex Franz. 2006. Google Web 1T 5-gram Corpus, version 1. Linguistic Data Consortium, Philadelphia, Catalog Number LDC2006T13, September.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04*, pages 30–39, Philadelphia, PA, USA.

Philip Clarkson and Ronald Rosenfeld. 1997. Statistical language modeling using the CMU-cambridge toolkit. In *Proceedings of ESCA Eurospeech 1997*, pages 2707–2710.

Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation? In *StatMT '06: Proceedings of the Workshop on Statistical Machine Translation*, pages 94–101, Morristown, NJ, USA. Association for Computational Linguistics.

Marcello Federico and Mauro Cettolo. 2007. Efficient handling of n-gram language models for statistical machine translation. In *StatMT '07: Proceedings of the Second Workshop on Statistical Machine Translation*, pages 88–95, Morristown, NJ, USA. Association for Computational Linguistics.

Edward Fredkin. 1960. Trie memory. *Commun. ACM*, 3(9):490–499.

Kimmo Fredriksson and Fedor Nikitin. 2007. Simple compression code supporting random access and fast string matching. In *Proc. of the 6th International Workshop on Efficient and Experimental Algorithms (WEA'07)*, pages 203–216.

Ulrich Germann, Eric Joanis, and Samuel Larkin. 2009. Tightly packed tries: How to fit large models into memory, and make them load fast, too. *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language (SETQA-NLP 2009)*, pages 31–39.

Joshua Goodman and Jianfeng Gao. 2000. Language model size reduction by pruning and clustering. In *Proceedings of ICSLP'00*, pages 110–113.

David Graff. 2003. English Gigaword. Linguistic Data Consortium, catalog number LDC2003T05.

Boulos Harb, Ciprian Chelba, Jeffrey Dean, and Sanjay Ghemawat. 2009. Back-off language model compression. In *Proceedings of Interspeech*, pages 352–355.

Bo-June Hsu and James Glass. 2008. Iterative language model estimation:efficient data structure & algorithms. In *Proceedings of Interspeech*, pages 504–511.

F. Jelinek, B. Merialdo, S. Roukos, and M. Strauss I. 1990. Self-organized language modeling for speech recognition. In *Readings in Speech Recognition*, pages 450–506. Morgan Kaufmann.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: open source toolkit for statistical machine translation. In *ACL '07: Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 177–180, Morristown, NJ, USA. Association for Computational Linguistics.

Andreas Stolcke. 1998. Entropy-based pruning of backoff language models. In *Proceedings of DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274.

Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *Proceedings of the International Conference on Spoken Language Processing*, volume 2, pages 901–904, Denver.

David Talbot and Thorsten Brants. 2008. Randomized language models via perfect hash functions. *Proceedings of ACL-08 HLT*, pages 505–513.

David Talbot and Miles Osborne. 2007a. Randomised language modelling for statistical machine translation. In *Proceedings of ACL 07*, pages 512–519, Prague, Czech Republic, June.

David Talbot and Miles Osborne. 2007b. Smoothed bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of EMNLP*, pages 468–476.

David Talbot and John M. Talbot. 2008. Bloom maps. In *4th Workshop on Analytic Algorithmics and Combinatorics 2008 (ANALCO'08)*, pages 203—212, San Francisco, California.

David Talbot. 2009. Succinct approximate counting of skewed data. In *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*, pages 1243–1248, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *WEA'08: Proceedings of the 7th international conference on Experimental algorithms*, pages 154–168, Berlin, Heidelberg. Springer-Verlag.

Edward Whittaker and Bhiksha Raj. 2001. Quantization-based language model compression. Technical report, Mitsubishi Electric Research Laboratories, TR-2001-41.