Efficient Minimal Perfect Hash Language Models

David Guthrie, Mark Hepple, Wei Liu

Department of Computer Science, University of Sheffield
D.Guthrie@dcs.shef.ac.uk, M.Hepple@dcs.shef.ac.uk, W.Liu@dcs.shef.ac.uk

Abstract

The recent availability of large collections of text such as the Google 1T 5-gram corpus (Brants and Franz, 2006) and the Gigaword corpus of newswire (Graff, 2003) have made it possible to build language models that incorporate counts of billions of n-grams. This paper proposes two new methods of efficiently storing large language models that allow O(1) random access and use significantly less space than all known approaches. We introduce two novel data structures that take advantage of the distribution of n-grams in corpora and make use of various numbers of minimal perfect hashes to compactly store language models containing full frequency counts of billions of n-grams using 2.5 Bytes per n-gram and language models of quantized probabilities using 2.26 Bytes per n-gram. We show that our approaches are simple to implement and can easily be combined with pruning and quantization to achieve additional reductions in the size of the language model.

1. Introduction

Determining the most probable sequence of words is an integral part of many natural language processing tasks and the probabilities of these word sequences are often stored in a structure called a language model. Language models typically must hold n-grams of each distinct sequence of words up to length n that occur in a collection of training documents and associate with each of these sequences a frequency count or a probability. The use of language models for tasks such as machine translation and speech recognition have shown that increasing the size of the model is a constructive way of improving the performance on those tasks. Recently, Brants et al. (2007) showed that machine translation quality continues to improve even when increasing the size of the text used to build the language model above 1 trillion tokens.

Storing language models so that counts or probabilities can be accessed quickly has become problematic due to the space required to store these large models. Language models have commonly been stored in compact trie structures that allow fast searching and enumeration of n-grams, yet these structures do not scale well and require space proportional to both the number of n-grams and the vocabulary size. Recently, significant advances in reducing the amount of space required to store a language model have been made by the introduction of random access models by Talbot and Osborne (2007a), Talbot and Osborne (2007b), and Talbot and Brants (2008).

Random access language models make use of the idea that it is not necessary to actually store n-grams in the language model as long as when queried with an n-gram the model returns the correct count or probability associated with that n-gram. This has been achieved through the clever use of Bloom filters (Bloom, 1970) and Bloomier filters (Chazelle et al., 2004) that trade a very small probability of returning a false positive for the fact that they can represent data very efficiently. These techniques allow the storage of language models that are no longer dependent on the size of the vocabulary, but only on the number of n-grams.

In this paper we propose two new random access language models that introduce the use of tiers of minimal perfect hash functions to exploit the distribution of words in language and further reduce the size required to store language models. We achieve this reduction in space without increasing the time required to query the model for an n-gram's probability or increasing the false positive rate. Our method achieves storage of full n-gram counts using just 2.5 Bytes per n-gram, which is 36% of the size that would be required by the previously most compact random access model proposed by Talbot and Brants (2008).

2. Related Work

Several methods have been used to reduce the amount of storage required for language models by storing less information. For instance it is possible to reduce the number of n-grams that must be stored in the model using entropy pruning techniques (Stolcke, 1998), clustering (Jelinek et al., 1990; Goodman and Gao, 2000) or simply by throwing away n-grams that occurred infrequently in the training data. It is also possible to reduce the amount of bits required to store each n-gram's associated probability or count by sacrificing some precision using quantization (Whittaker and Raj, 2001). Quantization divides the possible range into Q discrete values so that probabilities can be stored using only $\lceil \log_2 Q \rceil$ bits. These techniques can be used with any structure (including ours) for further reductions in the size of the language model and so can be seen as complementary to ours. This paper focuses on the data structures that are used to store these n-grams and their probabilities whether or not they have first been pruned or quantized.

Two main approaches to storing language models have been used: compact trie structures and hash like data structures that allow random access. In this section we give a brief overview of both approaches, highlighting the advantages and disadvantages of each.

2.1. Trie based language models

Most modern language modeling toolkits including SRILM (Stolcke, 2002), CMU toolkit (Clarkson and Rosenfeld, 1997), MITLM (Hsu and Glass, 2008), and IRSTLM (Federico and Cettolo, 2007) currently store their language models using variations on a trie data structure. A trie (Fredkin, 1960) is a compact tree structure where each node stores the unique prefixes of the nodes below it in the tree.

For n-gram language models these prefixes are typically 24 or 32 bit integers that represent words (typically all words in the vocabulary are assigned a distinct integer representation). So every node in the trie along with its parent nodes represents a distinct n-gram. The probabilities or counts associated with an n-gram can be stored in the tree node along with the word. Very compact representations of these structures have been devised that do not require storing pointers and are used in the CMU toolkit (described in Whittaker and Raj (2001)) as well as the MITLM and others (Harb et al., 2009; Germann et al., 2009). Tries allow the model to be stored in relatively little space and they also permit enumeration over n-grams in the model. For instance it is possible to list all n-grams in the model or to query the model for n-grams that begin with certain words and iterate through all of the results.

The main drawback of the trie approach is that it needs to store a representation for every word in the n-gram and the probabilities. Even using 24 bit integers for words in the vocabulary and quantizing probabilities to 8-bit integers; this model requires significantly more space than the random access approach described in the next section. The size of the trie model can be reduced using block compression as in Harb et al. (2009), but this technique can equally be applied to random access models if the increase in the time required to query the model is acceptable.

2.2. Random Access Language Models

Random access language models (Talbot and Osborne, 2007b; Talbot and Brants, 2008; Talbot and Osborne, 2007a) make use of hash functions to map n-grams to their associated probabilities or counts. 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 and speed advantages make them extremely attractive. These methods store language models in relatively little space by not actually storing the n-gram key in the structure and allowing a small probability of returning a false positive. In the case of n-grams these models always return the correct probability associated with an n-gram if the n-gram is in the model, but for n-grams that are not in the model there is a small probability that the model will return some random probability instead of correctly reporting that the n-gram was not found. There have been two major approaches used for storing random access language models: Bloom Filters and Bloomier Filters. We give an overview of these approaches below.

A Bloom filter (Bloom, 1970) is a data structure used in membership queries. It can be used to answer simple queries of the form "Is this key in the Set?". This is a weaker structure than a dictionary or hash table which also can associate 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 1 to m. These values serve as indices to the bit array B and the bits at those positions are set to 1. So, $B[h_i(x)] = 1$ for $i = 1$ to k. 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 the set S contains a key, say w, we run our k hash function on w and check to see 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 .0027.

Talbot and Osborne (2007b) and Talbot and Osborne (2007a) made use of bloom filters to store a trigram language model by inserting into the bloom filter a concatenation of an n-gram and its associated count. To insert a trigram that occurred c times they inserted the trigram into the Bloom filter c times, each time appending a count from 1 to c. To retrieve a trigram count from the model, a trigram is first queried by appending a count of 1 and then if the filter returns true the trigram is queried again appending a count of 2. This process is repeated until the filter returns false. They limit this process by quantizing all counts to 8-bits so there are a maximum of 256 iterations that must be tried. The Zipf-like distribution of language means that many of the n-grams in the test data will occur a small number of times, so for most lookups the iteration count is small.

More recently Talbot and Brants (2008) proposed the use of the Bloomier filter for storing language models. A Bloomier filter (Chazelle et al., 2004) is a membership testing data structure that allows values associated with the keys to be retrieved from the filter. Bloomier filters generalize the Bloom Filters to encode arbitrary functions while maintaining their economical use of storage. For details on the construction of Bloomier Filters see Chazelle et al. (2004; Talbot and Brants (2008). The Bloomier filter is used by Talbot and Brants (2008) to store for each n-gram an associated probability. Like the Bloom filter, the Bloomier filter has a small probability of returning an incorrect result when queried with an n-gram that does not exist in the model, but will always return the correct probability if the n-gram is in the model.

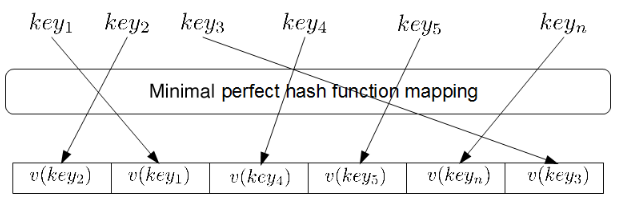

Figure 1: A minimal perfect hash function maps keys to integers in the range 0 to $|keys| - 1$ with no collisions

The Bloomier Filter can be thought of as a perfect hash function (Botelho et al., 2007) and this is a helpful way of understanding how it trades some probability of false positives for efficient storage. A perfect hash function is a function that maps the n distinct elements of S to distinct integers in the range 1 to r with no collisions. (A *minimal* perfect hash function, described in the next section, is a function that can do this with $n = r$, see Fig. 1.) We can think of a Bloomier language model as consisting a perfect hash function $ph()$ that maps n-grams to integers in the range 1 to r and then storing at that index a fingerprint and a value. The fingerprint is generated by a standard independent hash function and the value is the probability or count associated with that n-gram. The amount of bits used for the fingerprint determines the false positive rate for n-grams not in the model. Storage of the fingerprint is necessary if we know that the model might be queried for n-grams not in the model because the perfect hash function $ph()$ will likely map unseen trigrams to some random index and only by comparing the fingerprints we can say with high probability whether that n-gram was actually in the model. (Talbot and Brants, 2008) give results of using this model for Machine Translation experiments using 8 to 16 bit fingerprints and storing 5 to 8 bit quantized probability values. Using this approach it is possible to store 8-bit quantized probabilities (256 discrete values) for n-grams with 12-bit fingerprints, using only 3.08 Bytes per n-gram. This is significantly less than all known approaches, but we show that using our tiered minimal perfect hash structure, described in the next sections, it is possible to store full n-gram counts (no quantizing) using only 2.5 Bytes per n-gram (without increasing the false positive rate or the time required to query the model).

3. Single MPHR Approach

In this section we describe our basic Single Minimal Perfect Hash Rank approach (Single MPHR). Later in the paper we show that this model can be extended by using two tiers of minimal perfect hashes to save even more space while still keeping a constant look up time.

3.1. Storing Ranks of Frequencies

We describe our storage structure assuming we are storing frequency counts of n-grams. Instead of storing the frequency count of each n-gram, we store the 'rank' of the frequency count for each n-gram and have a separate array to store the actual frequency count values, where the index of this array is the 'rank'. This was motivated by the sparsity 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 1). 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. The memory savings in this step is thus due to the fact that the number of bits needed to store all the unique ranks is much less than the bits needed to store the maximum frequency count associated with an n-gram, $\lceil \log_2 K \rceil \ll \lceil \log_2 \text{maxcount} \rceil$.

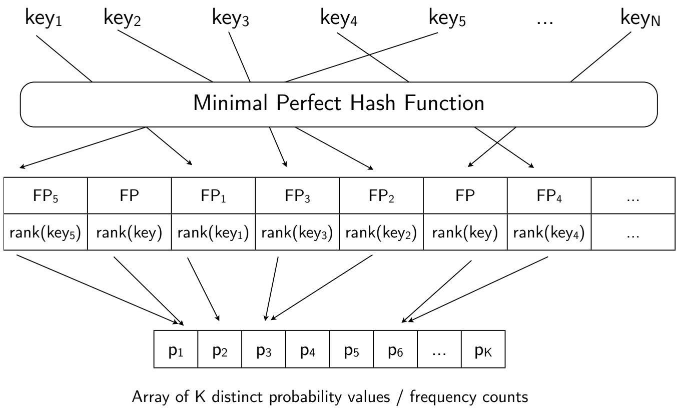

Figure 2: Single MPHR data structure

3.2. Structural Overview

Like the random access models of Talbot and Osborne (2007b; Talbot and Brants (2008; Talbot and Osborne (2007a) we do not store n-grams directly, but instead map each n-gram that we want to store using a perfect hash function. This hash function assigns to each n-gram an index that we use to store a $fingerprint$ and a *rank*. The hash function will always return the correct index for each stored n-gram, but it will also return an index for n-grams that were never stored, so it is necessary to store the additional fingerprint to reduce the probability of getting false positives.

To query an n-gram, we first look at the fingerprint stored in the position given by the perfect hash function, then we check whether the fingerprint of this n-gram matches the stored fingerprint. If they match then we return its $rank$, and another look-up into the value array is required to return the actual frequency count or probability of the n-gram. If the $fingerprints$ do not match, we can be sure that the query n-gram is not stored in our model. These lookups are order $O(1)$. The structure of this model is shown in Figure 2.

It should be noted that storing frequency counts is not a requirement of the model, if instead of storing frequency counts we were to store probabilities quantized to Q values, our method will at most require $\lceil \log_2 Q \rceil$ bits per n-gram and likely less.

3.3. Fingerprints

We use Austin Appleby's Murmurhash2[1] function to fingerprint each n-gram and then store the m highest order bits of the fingerprint. This is a simple function that produces a random sequence of bits given a key. We store this random bit sequence, or fingerprint, for every n-gram in the model and then later when the model is queried for an n-gram we check to see if the queried n-gram's fingerprint matches the

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

Google Web1T	maximum n-gram frequency count	unique frequency counts
1gm	$95,119,665,584$	$238,592$
2gm	$8,418,225,326$	$504,087$
3gm	$6,793,090,938$	$408,528$
4gm	$5,988,622,797$	$273,345$
5gm	$5,434,417,282$	$200,079$
Total	$95,119,665,584$	$771,058$

Table 1: unique n-gram frequency counts from Google Web1T corpus

fingerprint size (bits)	no. of FPs	FP Rate
8	5132312	3.906e-2
12	320574	2.440e-4
16	19804	1.507e-05

Table 2: False positives using the MPHR approach. We queried the Google Web 1T 1+2+3-gram model with 1.3 billion unseen 4-gram keys.

stored one. This is necessary because, as we mentioned in the previous section, even though the minimal perfect hash function always returns a distinct integer in the range 1 to N for every n-gram stored in the model, it will also return an integer for a previously unseen n-gram that was never stored in the model. A false positive occurs if the stored fingerprint located by the minimal perfect function happens to match the fingerprint of this *unseen* n-gram. If this happens the model will incorrectly return the value associated with that stored n-gram rather than reporting that the *unseen* n-gram was not found.
If the fingerprints were generated by a truly random hash function then the expected false positive rate of the model would be the same as selecting two identical fingerprints:

$$\frac{1}{2^m} \tag{1}$$

Where m is the number of bits used for the fingerprint. To test the actual false positive rates for our model we used it to store all 1 to 3 grams in the Google Web1T corpus, and then queried the model for all 4-grams from the Web1T corpus. These 4-grams are all unseen for this model, so when the fingerprints are compared, if the fingerprint of the query matches the stored fingerprint then a false positive has occurred. Table 2 shows that our actual false positive results are very close to the expected value.

3.4. Minimal Perfect Hashing

We use the *Hash, displace, and compress (CHD)* (Belazzougui et al., 2009) algorithm to generate a minimal perfect hash function that requires 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. Belazzougui et al. (2009) show that in practice it sufficient to use functions that behave similarly to fully random independent hash functions and they 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 an index j in T that contains a zero.

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

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 as proposed by Fredriksson and Nikitin (2007).

fingerprint	value	RPH	**S-MPHR**	savings
(bits)		bytes/n-gram		
8	8	2.46	2.26	8.18%
8	12	3.08	2.76	10.28%
8	20	4.31	3.76	12.69 %
8	32	6.15	5.26	14.49 %
12	8	3.08	2.76	10.28%
12	12	3.69	3.26	11.69%
12	20	4.92	4.26	13.44 %
12	32	6.77	5.75	14.87 %

Table 3: Comparison between Talbot and Brants (2008) Randomized (RPH) and our Single MPHR method (S-MPHR)

Method	Space bytes/n-gram
CMU 24 bit	6.2
IRSTLM	9.1
Randomized (Talbot&Brants)	3.08
Single MPHR	**2.76**

Table 4: Comparison between different language model storage representations all storing 8-bit quantized values and using 12 bit fingerprints for the random access methods.

3.5. Storage Requirement

Using the *Hash, displace, and compress (CHD)* algorithm, our hash function takes up $2.07 * N$ bits for an N sized n-gram language model. Let us assume that the N sized n-gram model has a total K distinct frequencies counts. An N sized array is needed to store the bits needed to keep m-bit $fingerprints$ and $\lceil log_2 K \rceil$ bits for the rank of each n-gram. Lastly, another K sized array to store the actual frequency count or probability associated with each n-gram (each being stored using say a 32 or 64 bit integer of size T). This array is very small compared to the other parts of the structure since $K \ll N$ and in a typically large n-gram language model its memory usage is insignificant. The total space requirement of our model can be calculated as shown in Equation 1. Table 3 and 4 show the bytes per n-gram requirement of our Single Minimal Perfect Hash Rank Model compared to other methods.

$$\begin{array}{lr} 2.07 \times N & \text{MPH function} \\ +(m + \lceil log_2 K \rceil) \times N & \text{fingerprints+ranks} \\ +K \times sizeof(T) & \text{value array} \\ = \text{ bits required} & \end{array}$$

Equation 1: Calculating space requirements of the Single MPHR structure

To demonstrate this we stored n-grams and full frequency counts for the entire Google Web1T corpus (Brants and Franz, 2006). This corpus is 24.6GB compressed and contains over 3.7 billion n-grams, so storing full frequency counts for every n-gram in a representation where they can be accessed quickly can be difficult. The Web1T corpus contains frequencies as large as 95 billion, so we would need at least 37 bits to store accurate counts for each n-gram. Using the RPH 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 Single Minimal Perfect Hash Rank method requires only 4.26 bytes per n-gram to store full frequency count information and so can store the entire corpus in just **14.98GB** or **57%** of the space required by the RPH method.

This savings is mostly due to the fact that we need only 20 bits per n-gram, instead of 37, to store the $ranks$ for every n-gram frequency count in the corpus. We can apply the same rank array optimization to the RPH method, so that it would also use only 20 bits to store $ranks$ and an additional array to hold the actual frequency counts; this significantly reduces the amount of memory required, but our Single MPHR structure still uses 86% of the space required by the RPH approach.

4. Tiered MPHR

We next describe our model can be elaborated into one that uses multiple hash stores to achieve even greater space efficiency. We go on to consider how this approach might provide a basis for storing even larger language models within a distributed architecture.

4.1. Improving space efficiency

Although the Single MPHR approach achieves a significant improvement in space efficiency over previous methods, there is still a considerable premium on achieving even more efficient space usage, given the size of current large language models and the larger ones that may be over the horizon. Our use of count *ranks* to record count information may be seen as exploiting distributional characteristics of the data in achieving more compact storage, i.e. the fact that in the range up to the maximum count found in some data, many of the possible count values are not used, and so replacing actual counts with ranks allows them to be represented using a much smaller numerical range.

In this section, we further exploit distributional characteristics of the data to achieve even more compact storage of this information, and specifically the fact that lower rank values (i.e. those assigned to count values shared by very many n-grams) are sufficient for representing the count information of a disproportionately large portion 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.

As a simple approach to achieving this gain, we might *partition* the n-gram data into subsets requiring different amounts of space for storing rank values, and store these subsets in separate MPHR structures, e.g. with two MPHRs having 8 and 20 bits respectively for storing the ranks for the example just mentioned. A more extensive partitioning of the data might further reduce this average cost, e.g. with subsets requiring 4, 8, 12, 16 and 20 bits, respectively. This simple approach has several problems. Firstly, it potentially requires a *series* of look up steps (i.e. up to 5 for the latter example) to retrieve the count of any n-gram, with *all* hashes needing to be addressed to determine the unseen status of an unseen n-gram. Secondly, and perhaps more seriously, such multiple look ups produce a *compounding* of false-positive error rates. Thus, we might falsely accept an unseen n-gram as seen in each look up step, and we may additionally construe a seen n-gram as being stored in the wrong MPHR and so return an incorrect count for it.

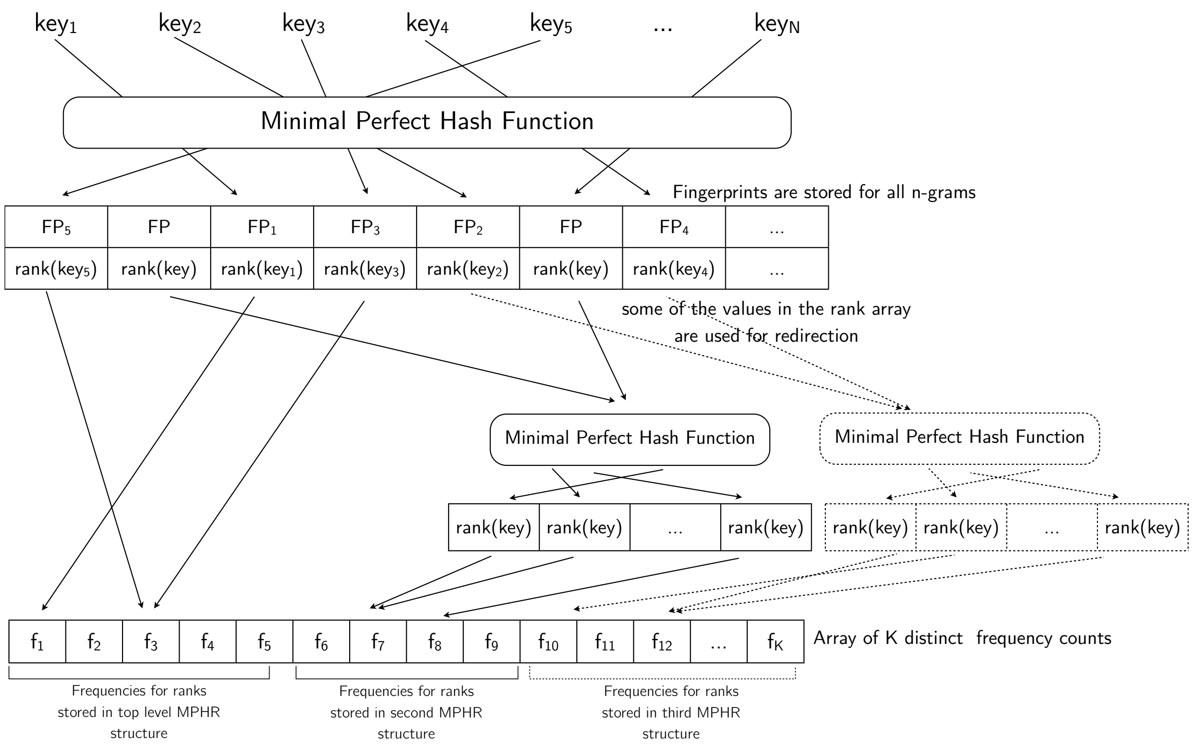
key1
key2
key3
key4
key5
...
keyN
Minimal Perfect Hash Function
Fingerprints are stored for all n-grams
FP5
FP
FP1
FP3
FP2
FP
FP4
rank(key5)
rank(key)
rank(key1)
rank(key3)
rank(key2)
rank(key)
rank(key4)
some of the values in the rank array are used for redirection
Minimal Perfect Hash Function
Minimal Perfect Hash Function
rank(key)
Array of K distinct frequency counts
f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 ... fK
Frequencies for ranks stored in top level MPHR structure
Frequencies for ranks stored in second MPHR structure
Frequencies for ranks stored in third MPHR structure

Figure 3: Tiered minimal perfect hash data structure

We will here explore an alternative approach to using multiple hashes that we call *Tiered* MPHR, which entirely avoids the compounding of false-positive errors, and which limits the maximum number of looks up steps to 2, irrespective of how many hashes are used. In this approach, there is a single *top-level* MPHR which has the full set of n-grams for its key-set, and which stores a fingerprint for every n-gram. In addition, space is allocated to store rank values, but with some possible values of this store being reserved to indicate *redirection* to other *secondary* hashes where count 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 two-level MPHRs. Let us represent a configuration for a two-level MPHR as a sequence of bit values for their rank storage components, e.g. `(8,10,20)` for the example above, or $H = (b_1, \ldots b_h)$ more generally.

The overall memory cost of a particular configuration depends crucially on distributional characteristics of the data to be stored. In particular, for each rank value r, we need to know the proportion of n-grams accounted for by ranks $[1..r]$, which we denote $\mu(r)$, which is easily computed from the data. The top-level MPHR of a configuration $H = (b_1, \ldots b_h)$ has all n-grams from the data in its key-set, so its memory cost is calculated as before as $N \times (2.07 + m + b_1)$ (where m is the fingerprint size). The memory cost for each secondary MPHR depends on the number of n-grams it stores, which in turn depends on the range of ranks that it covers. For example, a secondary hash with storage size b_i that covers ranks $r_j, .., r_k$ has $N \times (\mu(r_k) - \mu(r_{j-1}))$ n-grams in its key-set and so has memory cost $N \times (\mu(r_k) - \mu(r_{j-1})) \times (2.07 + b_i)$. The range of ranks covered by a secondary hash depends on the hashes that precede it in the configuration sequence and the overall number of hashes. In a configuration with h hashes overall, the top-level MPHR must reserve $h - 1$ values for redirection, and so covers ranks $[1..(2^{b_1} - h + 1)]$. The second hash will then cover the next 2^{b_2} ranks, starting at $(2^{b_1} - h + 2)$, and so on.

Table 5 shows two-level MPHR configurations that are optimally space-efficient for the Google Web1T data, for different numbers of hashes used (as determined by a simple brute-force search of alternative configurations). We see that even a single secondary hash is sufficient to bring the average memory cost below 25 bits per n-gram. Having more hashes allows the cost to be further reduced, but with diminishing returns for larger numbers of hashes. Having 5 hashes overall is sufficient to bring the cost per n-gram below 24 bits (3 Bytes) using 12 bit fingerprints. If we instead use only 8-bit fingerprints the space usage drops to 19.77 bits (**2.5 Bytes**) per n-gram. So, using 8 bit fingerprints and storing full n-gram counts this model is **36%** of the size of the RPH model proposed by Talbot and Brants (2008).

Number of hashes	Configuration	Bits per n-gram
2	`(9,20)`	24.96
3	`(8,11,20)`	24.29
4	`(8,9,12,20)`	24.05
5	`(8,7,9,12,20)`	23.94
6	`(8,7,8,10,13,20)`	23.87
7	`(8,6,7,8,10,13,20)`	23.82
8	`(8,6,7,8,9,10,13,20)`	23.77

Table 5: Optimal Tiered MPHR configurations for Google Web1T corpus (using 12-bit fingerprints).

4.2. Language models in distributed architectures

Although the two-level MPHR approach makes it feasible to handle the full Google Web1T data, with full count information, on machines with quite limited memory (requiring $\sim$ 10.45GB for the 12 bit fingerprint model), substantially larger data sets could still challenge the resources of individual machines. For example, the Web1T corpus itself could presumably be *much* larger than it is, had lower count thresholds been applied in filtering it down. The two-level MPHR approach looks promising for use within a distributed architecture, as a basis for distributing both memory and processing cost, i.e. with the multiple hashes being stored on different machines, and with the return of a *redirect* value from the top-level MPHR serving as an instruction to request the required count from a specific other machine. As the approach is described above, however, by far the largest memory requirement associates with the top-level MPHR, and so storing even just this single hash structure places most of the memory burden on a single machine. For example, the 8-bit top-level hash of the configurations in Table 5 uses up 22.07 bits of the overall cost in each case. So in those configurations the first level hash accounts for over 90% of the space.

A substantial redistribution of the memory burden can be achieved simply by moving all fingerprints from the top-level MPHR to the secondary hashes. This move forces the top-level hash to store *only* redirect values (i.e. no fingerprints or ranks). This is because without fingerprints to verify n-grams it also cannot reliably return the rank of any n-grams. This modification allows the memory cost per n-gram of the top-level MPHR to be reduced to just 2.07 plus the bits required for redirection values, i.e. 2 bits for up to 4 secondary hashes, 3 bits for up to 8. To illustrate, we can consider the cost of some configurations were this approach to be applied to the Google Web1T data. For example, the configuration `(2,4,6,9,20)` has overall average memory cost of 25.49 bits per n-gram, with the top-level MPHR requiring only 4.07 bits per n-gram, and with no secondary hash requiring more than 8 bits per n-gram. The configuration `(3,3,3,4,5,6,8,11,20)` has overall cost per n-gram at 24.89 bits, with the top-level requiring 5.07 bits per n-gram, and all secondary hashes less than 4. The downside of this modified model is that *every* n-gram will require two look up steps, including all unseen n-grams. To reduce the rate of mistaken redirects for unseen n-grams, we could *share* the fingerprint between the top-level hash and the secondary hash, e.g. store the first 4 bits of the fingerprint in the top-level hash, so only 1 in 16 unseen n-grams would be redirected, with the rest of the fingerprint being stored/tested in the secondary hash. This modification would reduce the rate of 'false' redirects for unseen n-grams without affecting the overall false-positive rate of the model.

5. Conclusion

We have presented two efficient methods of storing large language models. These models allow for the storage of frequency counts or probabilities using less space than all known approaches while retaining O(1) access. We make use of recent work in minimal perfect hashing and take advantage of the distribution of words in language to store language models with full n-gram frequency counts using just 2.5 Bytes per n-gram with 8 bit fingerprints (or 3 Bytes per n-gram with 12 bit fingerprints). These techniques make it possible to use full language models consisting of billions of n-grams without pruning or quantizing even on modest hardware. We additionally show how to use our minimal perfect hash storage structures in a distributed environment to store even larger language models efficiently. We are currently working on further reducing the storage required by our methods by incorporating compression techniques.

6. 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.

Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and space-efficient minimal perfect hash functions. In *Proceedings of 10th Workshop on Algorithms and Data Structures*, volume 4619, pages 139–150.

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

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of EMNLP-CoNLL*, pages 858–867.

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 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.

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.

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