Minimal Perfect Hash Rank: Compact Storage of Large N-gram Language Models

David Guthrie
Department of Computer Science
University of Sheffield
Sheffield, England, S1 4DP
D.Guthrie@dcs.shef.ac.uk

Mark Hepple
Department of Computer Science
University of Sheffield
Sheffield, England, S1 4DP
M.Hepple@dcs.shef.ac.uk

ABSTRACT

In this paper we propose a new method of compactly storing n-gram language models called Minimal Perfect Hash Rank (MPHR) that uses significantly less space than all known approaches. It requires $O(n)$ construction time and allows for $O(1)$ random access of probability values or frequency counts associated with n-grams. We make use of *minimal* perfect hashing to store fingerprints of n-grams in an approach that is similar to, but more space efficient than, that used by [17], and then exploit distributional characteristics of n-gram data to achieve even more compact storage. This approach can store an 8-bit quantized version of the full Google Web 1T data at a cost of only 1.41 bytes per n-gram. Furthermore, it can store *full count information* for the same Google data at a cost of just less than 2.5 bytes per n-gram, which is only 36% of the space required by the [17] approach. We compare this language model storage structure with several recent approaches, as well as evaluate the actual running time of MPHR when storing and querying billions of n-grams keys.

1. INTRODUCTION

The availability of very large text collections, such as the Gigaword corpus of newswire [11], and the Google 1T 1-5gram corpus [4], have made it possible to build models incorporating counts of billions of n-grams. The storage of such large language models, however, presents serious problems, given both their size and the need to provide rapid access. A standard 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 [17, 19] have involved the development of *randomized* 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 storage that the representation of even a large language model can be held totally within memory, avoiding the latencies of disk access. These randomized 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, and clever use of *hash functions* is made to achieve this end. 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.

The first method published for randomized storage of language models is that of [19, 18], which is based on the use of Bloom Filters [2]. As we shall see, this approach is limited in its applicability to being used with particular quantization schemes, where the range of values to be stored is quite limited, and where the majority of items are associated with lower values in the range (or speed of access will suffer). Furthermore, the favorable space characteristics of the approach are evident only where reasonably high error rates can be tolerated.

These problems are solved by the approach of [17], which is based on the *Bloomier Filter* technique of [5]. A Bloomier Filter can be thought of as a *perfect hash function* (PHF) [3], which is a function that maps a set of keys S of size n (i.e. $|S| = n$) to *distinct* integers in the range $[0 \,..\, m-1]$, $m \geq n$, i.e. which produces *no collisions* for the predetermined keyset S. The PHF of a *Bloomier filter* is *implicit*, in that the approach allows values to be stored and retrieved without there being an identifiable component that maps keys to integers in the range $[0 \,..\, m-1]$. Bloomier Filters, like Bloom Filters, bring a risk of *false positives*. This risk is managed by storing values in combination with a *fingerprint* of the key, The space requirement per stored item is then the sum of the bits needed to store values, plus the number of additional bits assigned to catch errors via fingerprinting. The downside of this approach is that the hash construction algorithm requires $m > n$, with a ratio $m/n = 1.23$ being recommended to ensure a high probability of successful construction.

In this paper, we advance three proposals for improving the effectiveness of language model storage. The first proposal is to replace the use of Bloomier Filters with an alternative perfect hashing method which generates PHFs that are *minimal*, i.e. which map n keys into a range of size m, such that $m = n$. These minimal PHFs are *explicit* and have an associated cost-per-key for the PHF itself, but this additional cost trades against the benefit of having $m = n$ for

 SIGIR 2010 WEB N-GRAM Workshop, July 23, 2010, Geneva, Switzerland.

storing values/fingerprints (rather than $m/n = 1.23$), giving space efficiency savings of around 10–16% for storing typical language models. The approach provides for random access of values that is both $O(1)$ and very fast in practice (as we demonstrate). Our two further proposals exploit distributional characteristics of the data to achieve more compact storage, using ranks to represent sparse count information, and using two-tiered hash structures to partition the data according to the space required for value storage. This combination of techniques allows us, for example, to represent the *full* count information of the Google Web1T corpus [4] (where count values range up to 95 billion) at a cost of just less than 2.5 bytes per n-gram (assuming 8-bit fingerprints, for excluding false positives). This is surprisingly little more than the 2.26 bytes per n-gram that our basic model would need to represent a language model employing 8-bit quantization (again with 8-bit fingerprints), or the 2.46 bytes per n-gram that would be needed by the Bloomier Filter approach of [17] to store 8-bit quantized values with 8-bit fingerprints.

2. RELATED WORK

A range of *lossy* methods have been proposed, that can reduce the storage requirement of a language model by discarding information from it. A key method is *quantization* [21], which reduces the information that can be associated with n-grams to a fixed set of *discrete* values. Each such value stands in place of a *range* of possible probabilities (or frequency counts), and so mapping probabilities (or counts) to quantized values discards information by losing precision. A common case is 8-bit quantization, which allows for 256 distinct 'quantum' values, but other schemes are possible, and different approaches may be taken for how initial values are mapped onto quantum values. Other lossy methods in the use of entropy pruning techniques [15] or clustering [14, 10] to reduce the number of n-grams that must be stored. In what follows, we focus on the methods that have been used for *storing* language models, irrespective of whether or not lossy methods have been applied to first 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 [8], 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 steps taken so far. Thus, 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). For nodes in the trie that correspond to *complete* n-grams, other information can be stored, e.g. a probability or count value. Most modern language modeling toolkits employ some version of a trie structure for storage, including SRILM [16], CMU toolkit [6], MITLM [13], and IRSTLM [7]. An advantage of this structure is that it allows the stored n-grams to be enumerated. However, although this approach achieves compactness of representation for sequences, its memory costs are still such that very large language models require very large storage space, far more than the randomized methods to be 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 [12], but such extensions of the approach present further obstacles to rapid access.

2.2 Randomized Language Models

Recent randomized language models [19, 17, 18] make make use of random hash functions to map n-grams to their associated probabilities or counts. 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. 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. 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.

2.2.1 Bloom Filters

A Bloom filter [2] 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 $[0\,..\,m-1]$. 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 the set S contains a key, say w, we run our k hash functions 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 0.0027.

In [19] and [18], Bloom filters are adapted to allow the storage of values with keys (i.e. n-grams) by concatenating the key and value to form 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. This approach is called the *optimal counting Bloom filter*. Talbot and Osborne use a simple logarithmic quantization of counts that will produce quite 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 an 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. This fact restricts the applicability of the approach to limited contexts of use. Furthermore, 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) can substantially reduce the space efficiency of the method. As mentioned, the case $k = 3, m = 20n$ produces a false positive rate of 0.0027, but with a dataset that required each n-gram to be stored, say, 3 times on average (i.e. as 3 key-value pairs), the true storage cost would become 60 bits per original n-gram. To achieve a lower error rate, we can either increase the number of hash functions used, bringing additional computation cost, or increase the number of bits allowed per item stored.

A variant of this approach, which we note for later comparisons, is the *simple Bloom map* [20]. This approach stores each key (n-gram) only *once*, using a different set of hash functions to store the key depending on its associated value. To retrieve the value for a key, we test for it in the filter using the hash functions for each different value in turn until we receive a positive answer, or have tested for all values unsuccessfully (i.e. for an unseen n-gram). This retrieval model makes the approach computationally expensive, again limiting its practical applicability, but the approach is optimally efficient amongst Bloom filter methods for space usage, and its minimum space requirements can be straightforwardly computed in relation to the entropy of the data set.

2.2.2 Bloomier Filters

More recently, [17] proposed using the *Bloomier Filter* technique of [5] as a basis for storing large language models. Bloomier Filters generalize the Bloom Filter to allows 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 *bit-wise XOR*, to return the stored value. The risk of false positives is managed by making bit vectors be longer than the minimum length required to store values, and 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.

As noted earlier, a Bloomier Filter can be thought of as a

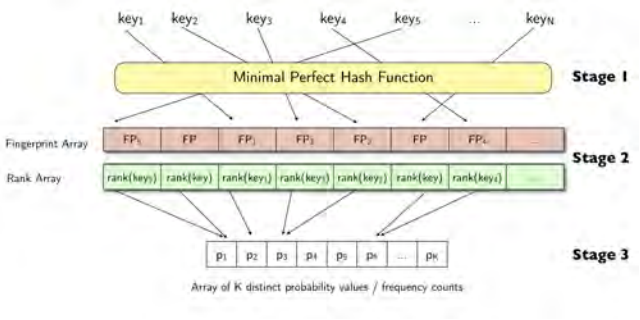

Figure 1: MPHR structure for storing n-gram language models

perfect hash function (PHF) [3], which is a function that maps a set of keys S of size n (i.e. $|S| = n$) to *distinct* integers in the range $[0 \,..\, m-1]$, $m \geq n$, i.e. which produces *no collisions* for the predetermined keyset S. When $m = n$ we have a *minimal* PHF. The PHF of a *Bloomier filter* is *implicit*, in that the approach allows values to correctly retrieved without there being an identifiable component that maps keys to integers in the range $[0 \,..\, m-1]$. The algorithm used to populate the Bloomier filter requires $m > n$, with a ratio $m/n = 1.23$ being recommended to ensure a high probability of successful construction. This factor contributes to the space required to store a language model, i.e. a model containing n n-grams, using v bits to represent values and e bits for error detection, required $1.23 \cdot n \cdot (v + e)$ bits of space.

3. MINIMAL PERFECT HASH RANKING APPROACH

We propose a data structure called Minimal Perfect Hash Rank (MPHR) that is more compact than that of [17] while still keeping a constant look up time. Our structure can be divided into 3 parts:

Stage 1 Minimal Perfect Hash Function

Stage 2 Fingerprint Rank Array

Stage 3 Unique Value Array

This structure is illustrated in Figure 1.

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)* [1] 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. In practice it sufficient to use functions that behave similarly to fully random independent hash functions and [1] 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 [9].

3.2 Fingerprint 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. 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 contains an array to store the *rank* for every n-gram. This rank is an index into the array of Stage 3 of our structure that holds the unique values associated with any n-gram.

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 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 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, $\lceil \log_2 K \rceil \ll \lceil \log_2 \text{maxcount} \rceil$.

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

4. STORAGE REQUIREMENTS

We next consider the storage requirements of our approach, and how they compare against those of other models, most particularly the Bloomier filter method of [17]. 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 Bloomier filter approach (RPH) [17] 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

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

detection), and a multiplying factor of 1.23, due to the non-miminality of the implicit PHF, giving an overall cost in bits of: $1.23 \cdot n \cdot (v + e)$
or of 1.23d *per n-gram*. The costs for our approach are similarly computed, but there is no multiplying factor, as our PHFs are minimal. However, the explicit minimal PHF computed using the CHD algorithm do bring an *additional* cost of 2.07 bits per n-gram for the PHF itself, and so the comparable overall cost to store a model is: $n \cdot (2.07 + d)$
or $2.07 + d$ per n-gram. For very small values of d, the Bloomier filter approach has the smaller cost, but the 'break-even' point occurs when $d = 9$, with our approach having the smaller cost, which in the limit could rise to a saving of around 18% for very large d. Table 2 shows some comparable costs of the two approaches for some plausible model settings. 5 compares the bytes per n-gram requirement of our model to a range of other methods.

fingerprint (bits)	value (bits)	RPH bytes/n-gram	**MPHR** bytes/n-gram	savings
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 %
16	8	3.69	3.26	11.69%
16	12	4.31	3.76	12.69%
16	20	5.54	4.76	14.02 %
16	32	7.38	6.26	15.19 %

Table 2: Comparison between RPH [17] and the MPHR method

The benefits that come from using the ranking method, for compactly storing count values, can only be evaluated in relation to the distributional characteristics specific corpora. To demonstrate this benefit, we stored n-grams and full frequency counts for the entire Google Web1T corpus [4]. 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 [17] 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 MPHR method requires only 4.26 bytes per n-gram to store full frequency count information and so can store the entire corpus in just **15.05GB** 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 (as shown in Section 3). 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 the MPHR structure still uses 86% of the space required by the RPH approach.

5. TIERED MPHR

In this section we describe how our basic model can be elaborated into one that uses multiple hash stores to achieve even greater space efficiency. Although the MPHR approach described in the last section achieves a significant improvement in space efficiency over previous methods, there is still a considerable incentive 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.

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

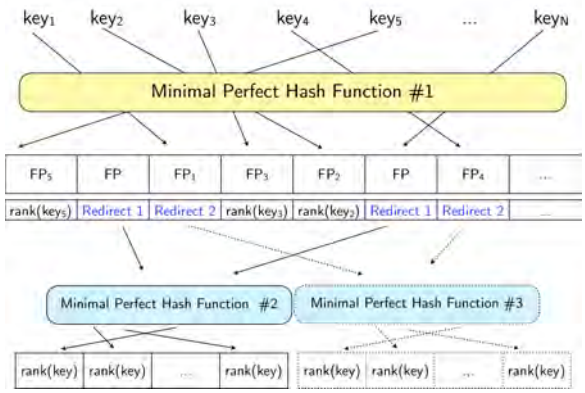

Figure 2: Tiered minimal perfect hash data structure

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 2 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 $r_j, \ldots, r_k$ 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 3 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 [17].

Number of hashes	Configuration	Counts in data	Bits per n-gram
2	`(9,20)`	full	24.96
3	`(8,11,20)`	full	24.29
5	`(8,7,9,12,20)`	full	23.94
8	`(8,6,7,8,9,10,13,20)`	full	23.77
2	`(1,8)`	quantized	15.24

Table 3: Optimal Tiered MPHR configurations for Google Web1T corpus (using 12-bit fingerprints).

6. FALSE POSITIVE RATES

The trade-offs of our compact storage model is the possibility of false positives. A false positive is when an *unseen* n-gram is queried, the model believes that this n-gram actually is stored in it and returns a (incorrect) value. This is because, as we mentioned in the previous section, the MPH function will return a distinct integer between $[0..N-1]$ for N stored n-grams, it will also, return an integer within that range for *unseen* n-grams. If the stored fingerprint located by the MPH function matches the fingerprint of this *unseen* n-grams, then the model will return the value associated with a *seen* n-gram (which is most likely incorrect). Assuming that

fingerprint size (bits)	no. of FPs	Actual FP Rate	Expected FP Rate
8	5132312	3.906e-2	3.906e-2
12	320574	2.440e-4	2.441e-4
16	19804	1.507e-05	1.525e-5

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

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: 2^{-m}.

Where m is the number of bits of the fingerprint. To test actual false positive rates we built a MPHR structure hold frequency counts for all 1 to 3 grams, and queried the model for all 4-grams from the Web1T corpus. These 4-grams are all unseen for this model, so when the array in Stage 2 is accessed, if the fingerprint of the query matches the stored

fingerprint then a false positive has occurred. Table 4 shows that these false positive results are very close to the expected value.

7. COMPARISON OF APPROACHES

We next consider some comparisons of our new methods to the alternative available approaches. Table 5 shows the cost of storing 8-bit quantized language models for different approaches, in bytes per n-gram, including trie-based LM toolkits (where figures are available). The figures for randomized methods assume error rates in line with 12-bit fingerprinting, and are calculated for the Google Web 1T data (when distributional characteristics affect costs). The cost for Bloom maps is calculated as $\log e(\log \frac{1}{\varepsilon} + H(\vec{v}))$ where $H_0(\vec{v})$ represents the zeroth order entropy of the distribution of values over keys. This entropy for the Web 1T data, when there is 8-bit uniform quantization of counts, is 1.8367 (or 8.3896 for the same data *without* quantization). Clearly the cost for trie-based methods is much greater than for the randomized methods.

We take a closer look at the latter case in Figure 4, to see how the cost varies across different error rates. The values plotted for the *optimal counting Bloom filter* are generously small, in that we do not take multiple storage of keys into account — we have instead simply looked up the minimum number of bits per item at which a given error rate can be achieved (by using enough hash functions). We see that the basic MPHR method has lower cost than the Bloomier filter approach, but greater cost than the Bloom filter methods, although the difference narrows with lower error rates. The Tiered MPHR method, however, achieves a significant cost saving over basic MPHR, such as to achieve lower cost than the Bloom filter methods for anything other the lowest error rates.

In Figure 3, we plot similar results for storing the *full count* information of the Google Web 1T data, including a line for the Bloom map. The relative space usage of methods here is closely in line with that observed with quantization. Although we plotted Bloom filter based approaches for comparison, neither the Bloom filter approach nor the Bloom Map approach plausibly extends to storing full counts because of the time that would be required to perform a query. To perform a single query, both structures in the worst case need $O(|\vec{v}| \cdot k)$ computations where $|\vec{v}|$ is the number of unique values stored and k is the number of hash functions as opposed to the constant time required by the MPHR, Bloomier, and Tiered MPHR structures. In addition to the infeasible time required for queries the false positive rates of these approaches also depends upon the number of hash functions computed and grows prohibitively large when storing full counts.

We tested the speed-of-access performance of the basic MPHR approach by building a model for the 1.3 billion n-grams of length 1–3 in the Web 1T data, and then querying it for all of its n-grams. This testing retrieved values at a rate of 564K queries per second — a rate of access that is around 1000 times faster than an alternative system storing the same data in a MySQL database.

To close this section, we note some further differences amongst

Method	Space bytes/n-gram
CMU 32b, 8-bit Quantized	7.2
CMU 24b, 8-bit Quantized	6.2
IRSTLM, 8-bit Quantized	9.1
RPH 12bit fp, 8bit Quantized	3.08
BloomMap $\varepsilon = 2^{12}$, 8bit Quantized	2.5
MPHR 12bit fp, 8bit Quantized	2.76
Tiered MPHR 12bit fp, 8bit Quantized	**1.91**

Table 5: Comparison between LM storage models to store 8 bit quantized values

the randomized approaches. Although all these methods suffer a problem of false positives, the Bloom filter methods also allow *misalignments*, where an incorrect value is returned for a *seen* n-gram. This does not arise for either RPH or MPHR methods. The RPH and MPHR methods have in common that the full set of keys must be available when the LM representation is constructed, but it is a characteristic of RPH that no values can be changed without invalidating the model. This is not true for MPHR, making it promising for use in adaptive LM contexts.

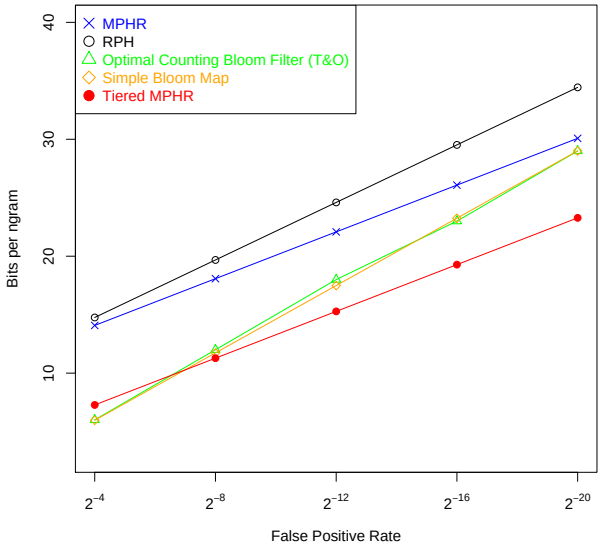

Figure 3: Comparison of space usage storing 8-bit quantized values

8. CONCLUSION

We have presented a two efficient methods of storing large language models, consisting of billions of n-grams, that allows for probability values or frequency counts to be accessed quickly and that store n-grams and full count information using less space than all know approaches. We show that using our Tiered MPHR structure we store all n-grams and values in the Google Web 1T dataset using 2.5 bytes per n-gram or 1.41 bytes per n-gram using quantized values. We have also shown that in addition to the efficient space usage

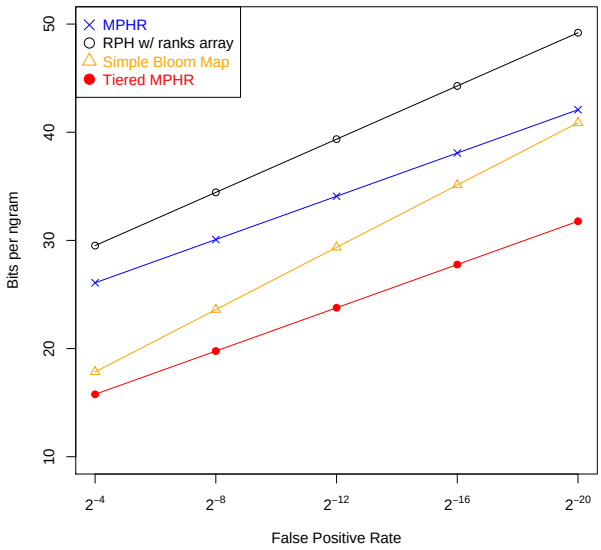

Figure 4: Comparison of space usage storing full counts

of our models, they have other advantages including O(1) query time, no possibility of missassignments, and ability to store full count information without increasing the query time.

9. REFERENCES

[1] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. *Algorithms - ESA 2009*, pages 682–693, 2009.

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

[3] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of 10th Workshop on Algorithms and Data Structures*, volume 4619, pages 139–150. 2007.

[4] T. Brants and A. Franz. Google Web 1T 5-gram Corpus, version 1. Linguistic Data Consortium, Philadelphia, Catalog Number LDC2006T13, September 2006.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04*, pages 30–39, Philadelphia, PA, USA, 2004.

[6] P. Clarkson and R. Rosenfeld. Statistical language modeling using the CMU-cambridge toolkit. In *Proceedings of ESCA Eurospeech 1997*, pages 2707–2710, 1997.

[7] M. Federico and M. Cettolo. 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, 2007. Association for Computational Linguistics.

[8] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[9] K. Fredriksson and F. Nikitin. 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, 2007.

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

[11] D. Graff. English Gigaword. Linguistic Data Consortium, catalog number LDC2003T05, 2003.

[12] B. Harb, C. Chelba, J. Dean, and S. Ghemawat. Back-off language model compression. In *Proceedings of Interspeech*, pages 352–355, 2009.

[13] B.-J. Hsu and J. Glass. Iterative language model estimation:efficient data structure & algorithms. In *Proceedings of Interspeech*, pages 504–511, 2008.

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

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

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

[17] D. Talbot and T. Brants. Randomized language models via perfect hash functions. *Proceedings of ACL-08 HLT*, pages 505–513, 2008.

[18] D. Talbot and M. Osborne. Randomised language modelling for statistical machine translation. In *Proceedings of ACL 07*, pages 512–519, Prague, Czech Republic, June 2007.

[19] D. Talbot and M. Osborne. Smoothed bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of EMNLP*, pages 468–476, 2007.

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

[21] E. Whittaker and B. Raj. Quantization-based language model compression. Technical report, Mitsubishi Electric Research Laboratories, TR-2001-41, 2001.