Ideally, we'd like to do as little pruning as possible and we'd like to use as large a *P* as possible, subject to the memory allocation. We don't have a principled argument for how to balance Stolcke pruning losses with hashing losses; this can be arrived at empirically on an application-specific basis. For example, to fix the storage per *n*-gram at around 13 bits:

$$13 = \left\lceil \frac{1}{\log_e 2} + \log_2 \frac{1}{\lambda} \right\rceil$$

If we solve for λ, we obtain $\lambda \approx 1/20{,}0000$. In other words, set *P* to a prime near $20{,}000N$ and then do as much Stolcke pruning as necessary to meet the memory constraint. Then measure your application's accuracy, and adjust accordingly.

6.2 HashTBO Values and Alphas

There are *N* log likelihood values, one for each key. These *N* values are quantized into a small number of distinct bins. They are written out as a sequence of *N* Huffman codes. If there are Katz backoff alphas, then they are also written out as a sequence of *N* Huffman codes. (Unigrams and bigrams have alphas, but trigrams don't.)

6.3 HashTBO Lookup

The lookup process is given an *n*-gram, $w_{i-2}w_{i-1}w_i$, and is asked to estimate a log likelihood, $\log \Pr(w_i | w_{i-2}\, w_{i-1})$. Using the standard backoff model, this depends on the likelihoods for the unigrams, bigrams and trigrams, as well as the alphas.

The lookup routine not only determines if the *n*-gram is in the table, but also determines the offset within that table. Using that offset, we can find the appropriate log likelihood and alpha. Side tables are maintained to speed up random access.

7 ZipTBO Format

ZipTBO is a well-established representation of trigrams. Detailed descriptions can be found in (Clarkson and Rosenfeld 1997; Whittaker and Raj 2001).

ZipTBO consumes 8 bytes per unigram, 5 bytes per bigram and 2.5 bytes per trigram. In practice, this comes to about 4 bytes per n-gram on average.

Note that there are some important interactions between ZipTBO and Stolcke pruning. ZipTBO is relatively efficient for trigrams, compared to bigrams. Unfortunately, aggressive Stolcke pruning generates bigram-heavy models, which don't compress well with ZipTBO.

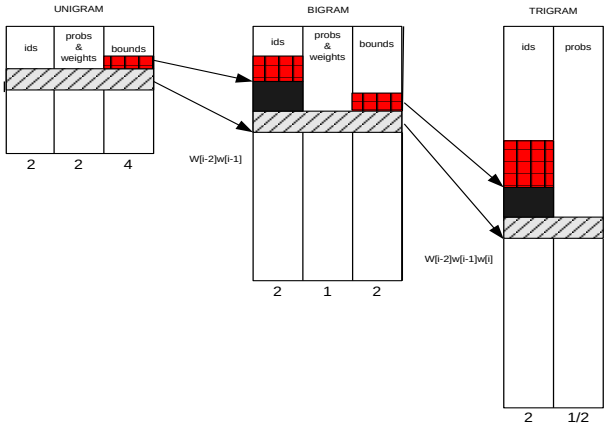

Figure 1. Tree structure of n-grams in ZipTBO format, following Whittaker and Ray (2001)

7.1 ZipTBO Keys

The tree structure of the trigram model is implemented using three arrays. As shown in Figure 1, from left to right, the first array (called *unigram array*) stores unigram nodes, each of which branches out into bigram nodes in the second array (*bigram array*). Each bigram node then branches out into trigram nodes in the third array (*trigram array*).

The length of the unigram array is determined by the vocabulary size (*V*). The lengths of the other two arrays depend on the number of bigrams and the number of trigrams, which depends on how aggressively they were pruned. (We do not prune unigrams.)

We store a 2-byte word id for each unigram, bigram and trigram.

The unigram nodes point to blocks of bigram nodes, and the bigram nodes point to blocks of trigram nodes. There are boundary symbols between blocks (denoted by the pointers in Figure 1). The boundary symbols consume 4 bytes for each unigram and 2 bytes for each bigram.

In each block, nodes are sorted by their word ids. Blocks are consecutive, so the boundary value