The *Virtual Corpus* Approach to Deriving Ngram Statistics from Large Scale Corpora*

Chunyu Kit†‡ Yorick Wilks†

Department of Computer Science, University of Sheffield†
{cykit, yorick}@dcs.shef.ac.uk
Department of Chinese, Translation and Linguistics, City University of Hong Kong‡
ctckit@cityu.edu.hk

Abstract

This paper reports our implementation of the *Virtual Corpus* approach to deriving ngram statistics for ngrams of any length from large-scale corpora based on the *suffix array* data structure. In order to enable the VC to accommodate corpora with a vocabulary of different size, we first convert corpus tokens into integer codes. To accelerate the processing, we employ a bucket-radixsort for sorting the VC indices (or pointers, each of which represents a sequence of corpus tokens from its position to the end of the corpus). The time complexity of the sorting algorithm is of $O(N \log_n N)$ in token code comparisons.

1 Introduction

It is known that ngram model is the simplest, most durable and successful statistical model for many natural language processing (NLP) and speech processing applications, e.g., as in [9, 2] and many others. Many high performance part-of-speech tagging systems, e.g., [10, 4, 3] and others, are based on ngram statistics. Multigram model [1, 6, 7] has become a practical language modelling technique.

Hitherto bigram and/or trigram models were used rather than any higher order ngrams. One of the reasons seems to be the unavailability of higher order ngrams, mainly because of the computational difficulty in acquiring long ngrams and their counts. However, the *suffix array* data structure [11], as a more economic implmentation of *PAT tree* [8], provides a basis for efficient approach to acquiring ngrams from large-scale corpora. Nagao and Mori [13] practise a position indexing technique similar to suffix tree, but the efficiency appears to need further enhancement.

In this paper, we report our implementation of the *Virtual Corpus* (VC) approach to deriving ngram statistics for ngrams of any length from large-scale corpora, with plenty of technical details. The system is called "virtual corpus" because it relies on a virtually existing sorted corpus for ngram counting (see next sections for detail). This approach first performs a pre-processing to convert corpus tokens into integer codes before constructing the VC, by which it gains the capacity to handle large-scale corpora with a vocabulary of various sizes. The VC employs a bucket-radixsort to facilitate the sorting of the VC pointers (or indices), each

*The first author gratefully acknowledges a University of Sheffield Research Scholarship awarded to him that enables him to undertake this work. We wish to thank Makoto Nagao and Shinsuke Mori for helpful communications, and to thank Hamish Cunningham, Ted Dunning, Rob Gaizauskas, Steve Renals and many other colleagues for various kinds of help and useful discussions.

of which represents a sequence of corpus tokens from a token position in the corpus to the end of the corpus. The time complexity of the sorting algorithm is $O(N \log_n N)$ in token comparisons, where N is the corpus length and a token is an integer code. This algorithm compares favourably with other sorting algorithms that are of $O(N \log N)$ complexity in string comparisons. Experiments show that it takes 1.5 minutes to sort the VC for the 1.29 million PTB-II WSJ POS tag corpus on a Sun Sparc 4 and 1.5 minutes to sort the VC for the 6-million-character Brown corpus on a Sun Ultra 2.

2 Virtual Corpus Based on Suffix Array

A virtual corpus is realised by a sequence of pointers (or indices), each of which points to a corpus token (e.g., a word, a character, etc.) in the real corpus and stands, virtually, for the sequence of tokens from the token to the end of the corpus. This is the essence of the suffix array data structure. The pointers are as many as the number of tokens in the corpus, i.e., the corpus length N. With these pointers we have a VC consisting of N token sequences. After sorting these pointers, the count of an ngram item consisting of any n tokens in the corpus is simply the count of the number of adjacent pointers that take the n tokens as prefix.

This VC approach based on the suffix array data structure can be exemplified by the following 3 stages with a very simple artificial corpus:

Stage 1 Construct a virtual corpus with pointers to each token's position.

```
real corpus:  b  a  b  a  a  c  b  a  a  b ....[] ([]: end of corpus)
              ^  ^  ^  ^  ^  ^  ^  ^  ^  ^
pointers:     p1 p2 p3 p4 p5 p6 p7 p8 p9 p10....[]

virtual corpus:

      p1  -> b a b a a c b a a b ....[]
      p2  -> a b a a c b a a b ....[]
      p3  -> b a a c b a a b ....[]
      p4  -> a a c b a a b ....[]
      p5  -> a c b a a b ....[]
      p6  -> c b a a b ....[]
      p7  -> b a a b ....[]
      p8  -> a a b ....[]
      p9  -> a b ....[]
      p10 -> b ....[]
          :
```

Stage 2 Sort the virtual corpus.

```
      p8  -> a a b ....[]
      p4  -> a a c b a a b ....[]
      p2  -> a b a a c b a a b ....[]
      p9  -> a b ....[]
      p5  -> a c b a a b ....[]
      p7  -> b a a b ....[]
      p3  -> b a a c b a a b ....[]
      p1  -> b a b a a c b a a b ....[]
      p10 -> b ....[]
      p6  -> c b a a b ....[]
          :
```

After the sorting, all VC pointers with an identical prefix (e.g., [b a] or [b a a] above) of any length have become adjacent to one another.

Stage 3 Count the occurrences of ngram items of any length n. This is simply to count the number of adjacent sequences (i.e., pointers) with the same prefix of the length n. It is rather straightforward to get the following ngrams and their counts from the above sorted VC:

```
uni-gram    bi-gram     tri-gram      quad-gram     .....

[a]: 5      [a a]: 2    [a a b]: 1    ....
[b]: 4      [a b]: 2    [a a c]: 1    [a a c b]: 1
[c]: 1      [a c]: 1    ....          ....
....        [b a]: 3    [b a a]: 2    [b a a b]: 1
            ....        [b a b]: 1    [b a a c]: 1
                        ....          ....
```

After the VC is sorted, you may either output the ngram results, or just simply keep the VC as an ngram resource such that an ngram is counted only when it is needed.

3 Implementation

Our implementation of the *Virtual Corpus* system for deriving ngrams of any length follows the notion of suffix array in principle, as illustrated in the preceding section. There are a number of technical details in the implementation that are worth noting to researchers interested in utilising ngram statistics in their research.

First, the corpus tokens are converted into digit codes before a VC is constructed. There are two major reasons for this. One is that it can make efficient use of the working space if it can determine the length of a code in terms of the vocabulary size (i.e., the number of distinct tokens in the corpus). Saving working space is a critical issue while dealing with large-scale corpora of millions of words. For example, for a PTB tagged corpus, since the tag number (which is 48) is smaller we can use a `char` for a PTB tag, which is usually of 2-3 letters. In this way, to retrieve a tag in the corpus is to read a `char`, instead of a string. This treatment saves both space and time. It also leads to another favourite consequence: it can speed up the VC pointer comparison in the sorting. It is obvious that the comparison of two `char` codes is much faster than that of two corpus tokens as strings of several characters (e.g., English words). The token-code conversion is rather simple and fast: read through the corpus, once a new token shows up, assign a code to it.

Second, a bucket-radixsort is employed for sorting VC pointers. Other sorting algorithms, e.g., the combsort and disc mergesort that are used in [13], are assumed to have the time complexity $O(N \log N)$ in comparisons of the VC pointers. However, as we can see it, this complexity measure seems inappropriate in the context of sorting VC pointers, because comparing two VC pointers is to compare the two sequences of corpus tokens they represent. For example, to compare two VC pointers as follows,

```
pi -> ti_0 ti_1 ti_2 ti_3 ... ti_k ... []
pj -> tj_0 tj_1 tj_2 tj_3 ... tj_k ... []
```

where `ti_k = crps[pi+k]`, assuming the corpus in the array `crps[]`, the comparison procedure can be characterised as the pseudo-C++ code below, which is rather straightforward and highly similar to string comparison. The first `tok_cmp()` is for token code comparison and the second one for token string comparison. Which one is used depends on the type of `crps[]`. Also, notice that there is no need to break the `for` loop, because one of `crps[pi+k]` and `crps[pj+k]` must hit the end-of-corpus `[]` (whose code is 0) as `k` increases, and then the result `rslt` is returned.

```
int ptr_cmp(int pi, int pj) // comparison of two VC pointers
{
  if (pi == pj) return 0;
  for (int rslt, k=0; ; k++)
    if (rslt = tok_comp(crps[pi + k], crps[pj + k])) return rslt;
}

int tok_cmp(int ti, int tj) { return ti - tj; }
int tok_cmp(str ti, str tj) { return strcmp(ti, tj); }
```

The number of token readings and token comparisons in VC pointer comparison as above actually depends on the length of the common prefix of the two pointers in question. This raises two problems. First, it appears to be problematic to characterise the complexity of combsort and mergesort on a VC as $O(N \log N)$ in VC pointer comparison, because the time for comparing two VC pointers is not constant. Consider an extreme corpus such as `[a a...a b]`, of N `a`'s and a `b`. Sorting its VC will take $O(N \log N)$ time in VC pointer comparisons, and any two pointers in the VC have a common prefix of length $N/2$ on average, i.e., comparing two VC pointers takes $O(N)$ time in token comparisons. Consequently, the combsort and mergesort for VC sorting have a complexity of $O(N^2 \log N)$ in token comparisons.

Second, the `ptr_comp()` procedure (or its equivalent) always starts comparing two VC pointers from the first token, then the second, the third, and so on, regardless of how many common prefix tokens have been found in the two VC pointers by previous comparisons. It is obvious that many comparisons of identical token pairs are unnecessarily repetitious.

However, this deficiency can be remedied by a bucket-radixsort. The pseudo-C++ code given below is the recursive version of the algorithm for sorting a VC with a vocabulary size n, where tokens are in integer codes. Since the outputting takes place during the recursion, there is no need to have a merge operation; otherwise the running time could last significantly longer.

```
void VC::brsort(int lvl) // bucket-radixsort a VC at level 'lvl'
{
  if (is_singleton()) output and return;
  VC *subVCs = new VC[n+1];                               // buckets
  put each ptr in *this VC into subVCs[crps(ptr + lvl)];  // division
  for (i=0; i<=n; i++)
     if (!subVC[i].is_empty()) subVCs[i].brsort(lvl + 1); // recursion
  delete [] subVCs;
}
```

It can be observed that each corpus token in each VC pointer is read only once during the sorting, because `lvl` increases incrementally and never repeats when the recursion goes deeper. The recurrence for this algorithm is $C_N = nC_{N/n} + N$. So, its complexity can be appropriately characterised as $O(N \log_n N)$ in token comparisons.

The bucket-radixsort we use in the VC program is an iterative version of the above algorithm, where a stack for subVCs is used as working space in place of `subVCs[]`, so as to avoid frequent memory allocation in each recursion as the above, and, more importantly, to avoid scanning through the n `subVCs` when most of them are empty. This can speed up the program to a great extent, particularly, when a corpus with a very large vocabulary-size is processed.

We have also tried an alternative sorting algorithm, namely the `qsort()` in standard C library, as suggested in [5]. What we need to do is to provide an appropriate function `int` `ptr_cmp(const void* p1, const void* p2)` for VC pointer comparison. Experiments show that the `qsort()` works basically as well as the `brsort()`. It runs slower than `brsort()` on a small-vocabulary corpus (e.g., POS tag or character corpus) but faster on a large-vocabulary corpus (e.g., word corpus). In the latter case, the above recursive version of the `brsort()` runs severely slower than its iterative version, because when n is large, the `for` loop takes a long time to go through all n `subVCs`. In particular, when the recursion goes deeper, the time wasted on this loop is more serious, because most of the `subVCs` are empty.

3.1 Output Ngrams

A simple program is implemented with a small trick for outputting ngrams of any length and their counts from a sorted VC. It is designed to output ngrams with respect to a specified minimum count (≥ 2) and minimum length. It is efficient in the sense that it scans through the entire VC once and outputs all ngrams. It is trivial to modify this program to output ngrams in a given range of length. The pseudo-C++ code of the algorithm is given below.

```
void VC::output_ngrams (int minC, int minL) const
     // output ngrams whose count >= minC (>= 2) & length >= minL
{
  int i, j, curcpl, tmpcnt;
  int *cnts = new int[maxcpl+1] & init: cnts[i] = 0; // cnts[0] not used
  for (i = 0, curcpl = cpl[i]; i < crpsLg; i++) {    // scan through the VC
     if (cpl[i] >= curcpl) cnts[cpl[i]]++;
     else { // cpl[i] < curcpl
        for (tmpcnt = 1, j = curcpl; j > cpl[i]; j--) {
           tmpcnt += cnts[j];    // count accumulation
           cnts[j] = 0;          // reset
           if (tmpcnt >= minC && j >= minL)
              output: ngram = crps[ptrs[i] .. ptrs[i]+j-1] & its count = tmpcnt.
        }
        cnts[cpl[i]] += tmpcnt;  // count accumulation
        cnts[curcpl]  = 0;       // reset
     }
     curcpl = cpl[i];
  }
  delete [] cnts;
}
```

The array `cpl[]` is used to facilitate the algorithm, where `cpl[i]` is the common prefix length of the `i_th` and the `i+1_th` VC pointers. The trick is the accumulation of counts with the aid of the array `cnts[]`, where `cnts[j]` is the count of the ngram item of the first `j` tokens in the VC pointer `ptrs[i]`, i.e., the ngram item `crps[ptrs[i]]...crps[ptrs[i]+j-1]`. `maxcpl` is the maximum common prefix length in the corpus.

It is worth noting that outputting ngrams takes many times longer than VC construction and sorting. If ngrams[1] of count 1 are also output, the required space can be as many times as the original corpus, depending on all ngrams up to which length are output.

4 Experiments

A number of experiments have been conducted on large-scale text corpora, including the PTB-II WSJ corpus and Brown corpus [12], to derive ngram statistics for ngrams of various lengths

[1]All fragments of the corpus that are not in the output by the call `output_ngrams(2, 1)` are of count 1.

and token types using the VC system. The experiments are carried out on different models of Sun station, in order to examine the program's performance objectively. On Sun Sparc 4, it takes about 1.5 minutes to sort the 1.29M PTB-II WSJ tag corpus. The table below is the experimental results on Sun Sparc 20 and Ultra 2. All running time data are obtained in this way: we run the program on each corpus for several times, and take the average of the two fastest running times as the final result.

On Sun Sparc 20				Time (sec.)		
Corpus	Size	Token Type	Vocabulary Size	`brsort`	`qsort`	Output VC
Brown	1.17M	POS Tag	89	32.5	45.0	25.0
		Word	47705	50.0	34.0	24.0
WSJ	1.29M	POS Tag	85	33.0	52.0	27.5
		Word	45649	48.0	40.0	27.0
On Sun Ultra 2				**Time (sec.)**		
Corpus	Size	Token Type	Vocabulary Size	`brsort`	`qsort`	Output VC
Brown	1.17M	POS Tag	89	11.0	15.0	6.5
		Word	47705	16.0	12.0	6.0
	6.13M	Char	58	86.0	106.5	36.5
WSJ	1.29M	POS Tag	85	13.0	17.0	6.5
		Word	45649	17.0	13.5	7.0
	5.73M	Char	57	86.5	96.0	32.5

One may ask why there are only 48 POS tags in PTB but we have 89 tags in Brown POS tag corpus and 85 tags in WSJ POS tag corpus. The reason is that there are a number of multiple tags used in the PTB corpus [12], e.g., JJ|NN, JJ|VBG, to avoid forcing annotators to make decisions they are unsure. These multiple tags are treated as different tokens from other individual POS tags by the VC system.

From the experimental results, we can see that the system takes about 11-12 seconds on Sun Ultra 2 and roughly half a minute on Sun Sparc 20 to prepare a sorted VC for a 1.17 to 1.29 million POS tag (or word) corpus for ngram counting. On Ultra 2, it takes about 1.5 minutes to do this for a 5.73 to 6.13 million character corpus. Furthermore, we also observe that, on Ultra 2, the `brsort()` is about 30-36% faster than the `qsort()` on a medium-sized corpus with a small-sized vocabulary (e.g., the WSJ and Brown POS tag corpus), but about 25-33% slower on a corpus of similar size with a large-sized vocabulary (e.g., WSJ and Brown word corpus). On a very large-scale corpus with a small-sized vocabulary, e.g., the WSJ and Brown character corpus, the `brsort()` is about 11-23% faster than `qsort()`. Nevertheless, both of them appear to be adequately fast for the purpose of practical use.

5 Conclusion

We have constructed a *Virtual Corpus* system for deriving statistics for ngrams of any length, based on the suffix array data structure. It can work on a tokenised corpus of any vocabulary size. Since an efficient sorting algorithm, whose complexity is $O(N \log_n N)$, is employed to do the VC sorting, it runs very fast. It takes less than 1.5 minutes to sort a VC for a corpus of 6 million characters on a Sun Ultra 2. It can output ngrams of various lengths at the same time. Since outputting all ngrams takes many times larger space than the original corpus, an alternative option other than outputting ngrams is to keep a sorted VC as the ngram resource.

Finally, a more important point we would like to point out is that the virtual corpus is not limited to serving as a tool for deriving ngram statistics for linguistic studies and language modelling. It can be employed as a basis to develop many other corpus tools where term (or position) indexing is critical, including concordancers, corpus tools (e.g., annotation systems for word segmentation, POS tagging, sentence bracketing, etc.) that a concordance program can facilitate, information retrieval systems, collocation extractor, etc. Also, a utility for extracting discontinuous co-occurrences of corpus tokens, of any distance from each other, can be implemented based on this program.

References

[1] Bimbot, F, R. Pieraccini, E. Levin and B. Atal. 1995. Variable-length sequence modelling: multigrams. In *IEEE Signal Processing Letters*, **2**(6):111-113.

[2] Brown, P. F., V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer. 1990. Class-based n-gram models of natural language. *Computational Linguistics*, **18**(4):467-479.

[3] Charniak, E., C. Hendrickson, N. Jacobson, and M. Perkowitz. 1993. Equations for part-of-speech tagging. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pp.784-789, AAAI Press/MIT Press.

[4] Church, K. W. 1989. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of International Conference on Acoustic, Speech and Signal Processing 1989*, pp.134-143, Glasgow, Scotland.

[5] Church, K. W. 1995. *ACL-95 Tutorial on NGRAMS*. June 16, 1995. MIT, MA.

[6] Deligne, S., and F. Bimbot. 1995. Language modelling by variable length sequences: theoretical formulation and evaluation of multigrams. In em ICASSP-95, Vol. 1: Speech, pp.169-172. May 9-12, 1995, Detroit, Michigan.

[7] Deligne, S., F. Yvon and F. Bimbot. 1995. Variable-length sequence matching for phonetic transcription using joint multigrams. In *European Conference on Speech Communication and Technology*, pp.2243-2246. September, 1995, Madrid.

[8] Gonnet, G. H., and R. Baeza-Yates. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

[9] Jelinek, F. 1985. Self-organized language modeling for speech recognition. IBM T.J. Watson Research Center, Continuous Speech Recognition Group, Yorktown Heights, New York, 1985. Also in A. Waibel and K. F. Lee (eds.), *Readings in Speech Recognition*, 1990, pp.450-506, Morgan Kaufmann, San Mateo, California.

[10] Leech, G., R. Garside and E. Atwell. 1983. The automatic grammatical tagging of the LOB corpus. *ICAME News*, **7**:13-33.

[11] Manber, U., and E. Myers. 1990. Suffix array: a new method for on-line string searches. In *First ASM-SIAM Symposium on Discrete Algorithms*, pp.319-327, American Mathematical Society, Providence.

[12] Marcus, M., B. Santorini and M. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. em Computational Linguistics, **19**(2):313-330.

[13] Nagao, M., and S. Mori. 1994. A new method of N-gram statistics for large number of n and automatic extraction of words and phrases from large text data of Japanese. In *COLING-94*, pp.611-615.