K-Best Suffix Arrays

Kenneth Church
Microsoft
One Microsoft Way
Redmond, WA 98052
church@microsoft.com

Bo Thiesson
Microsoft
One Microsoft Way
Redmond, WA 98052
thiesson@microsoft.com

Robert Ragno
Microsoft
One Microsoft Way
Redmond, WA 98052
rragno@microsoft.com

Abstract

Suppose we have a large dictionary of strings. Each entry starts with a figure of merit (popularity). We wish to find the *k*-best matches for a substring, *s*, in a dictionary, *dict*. That is, grep *s dict* | sort –n | head –*k*, but we would like to do this in sublinear time. Example applications: (1) web queries with popularities, (2) products with prices and (3) ads with click through rates. This paper proposes a novel index, *k-best suffix arrays*, based on ideas borrowed from suffix arrays and kd-trees. A standard suffix array sorts the suffixes by a single order (lexicographic) whereas k-best suffix arrays are sorted by two orders (lexicographic and popularity). Lookup time is between *log N* and *sqrt N*.

1 Standard Suffix Arrays

This paper will introduce *k-best suffix arrays*, which are similar to standard suffix arrays (Manber and Myers, 1990), an index that makes it convenient to compute the frequency and location of a substring, *s*, in a long sequence, *corpus*. A suffix array, *suf*, is an array of all *N* suffixes, sorted alphabetically. A suffix, *suf*[*i*], also known as a semi-infinite string, is a string that starts at position *j* in the corpus and continues to the end of the corpus. In practical implementations, a suffix is a 4-byte integer, *j*. In this way, an int (constant space) denotes a long string (*N* bytes).

The *make_standard_suf* program below creates a standard suffix array. The program starts with a *corpus*, a global variable containing a long string of *N* characters. The program allocates the suffix array *suf* and initializes it to a vector of *N* ints (suffixes) ranging from 0 to *N*−1. The suffix array is sorted by lexicographic order and returned.

```
int* make_standard_suf () {
  int N = strlen(corpus);
  int* suf = (int*)malloc(N * sizeof(int));
  for (int i=0; i<N; i++) suf[i] = i;
  qsort(suf, N, sizeof(int), lexcomp);
  return suf;}

int lexcomp(int* a, int* b)
{ return strcmp(corpus + *a, corpus + *b);}
```

This program is simple to describe (but inefficient, at least in theory) because *strcmp* can take O(*N*) time in the worst case (where the corpus contains two copies of an arbitrarily long string). See http://cm.bell-labs.com/cm/cs/who/doug/ssort.c for an implementation of the O(*N* log *N*) Manber and Myers algorithm. However, in practice, when the corpus is a dictionary of relatively short entries (such as web queries), the worst case is unlikely to come up. In which case, the simple *make_suf* program above is good enough, and maybe even better than the O(*N* log *N*) solution.

1.1 Standard Suffix Array Lookup

To compute the frequency and locations of a substring *s*, use a pair of binary searches to find *i* and *j*, the locations of the first and last suffix in the suffix array that start with *s*. Each suffix between *i* and *j* point to a location of *s* in the corpus. The frequency is simply: $j - i + 1$.

Here is some simple code. We show how to find the first suffix. The last suffix is left as an exercise. As above, we ignore the unlikely worst

Proceedings of NAACL HLT 2007, Companion Volume, pages 17–20,
Rochester, NY, April 2007.

case (two copies of a long string). See references mentioned above for worst case solutions.

```
void standard_lookup(char* s, int* suf, int N){
  int* i = find_first_suf(s, suf, N);
  int* j = find_last_suf(s, suf, N);
  for (int* k=i; k<=j; k++) output(*k);}

int* find_first_suf(char* s, int* suf, int N) {
  int len = strlen(s);
  int* high = suf + N;
  while (suf + 2 < high) {
    int* mid = suf + (high−suf)/2;
    int c = strncmp(s, corpus + *mid, len);
    if (c == 0) high = mid+1;
    else if (c < 0) high = mid;
    else suf = mid;}
  for ( ; suf < high; suf++)
    if (strncmp(s, corpus + *suf, len) == 0)
      return suf;
  return NULL;}          // not found
```

2 K-Best Suffix Arrays

K-best suffix arrays are like standard suffix arrays, except there are two orders instead of one. In addition to lexicographic order, we assume a figure of merit, which we will refer to as popularity. For example, the popularity of a string could be its frequency in a search log. The code below assumes that the corpus is a sequence of strings that comes pre-sorted by popularity, and then the popularities have been stripped off. These assumptions make it very easy to compare two strings by popularity. All *popcomp* has to do is to compare the two positions in the corpus.[1]

The *make_kbest_suf* program below is similar to the *make_standard_suf* program above except we now sort by the two orders at alternating depths in the tree. First we sort lexicographically and then we sort by popularity and so on, using a construction similar to KD-Trees (Bentley, 1975). The code below is simple to describe (though there are more efficient implementations that avoid unnecessary qsorts).

```
int* make_kbest_suf () {
  int N = strlen(corpus);
  int* suf = (int*)malloc(N * sizeof(int));
  for (int i=0; i<N; i++) suf[i]=i;
  process(suf, suf+N, 0);
  return suf;}

void process(int* start, int* end, int depth) {
  int* mid = start + (end − start)/2;
  if (end <= start+1) return;
  qsort(start, end-start, sizeof(int),
        (depth & 1) ? popcomp : lexcomp);
  process(start, mid, depth+1);
  process(mid+1, end, depth+1);}

int popcomp(int* a, int* b) {
  if (*a > *b) return 1;
  if (*a < *b) return −1;
  return 0;}
```

2.1 K-Best Suffix Array Lookup

To find the k-best matches for a particular substring *s*, we do what we would normally do for standard suffix arrays on lexicographic splits. However, on popularity splits, we search the more popular half first and then we search the less popular half, if necessary.

An implementation of *kbest-lookup* is given below. *D* denotes the depth of the search thus far. *Kbest-lookup* is initially called with *D* of 0. *Propose* maintains a heap of the k-best matches found thus far. *Done* returns true if its argument is less popular than the k^{th} best match found thus far.

```
void kbest_lookup(char* s, int* suf, int N, int D){
  int* mid = suf + N/2;
  int len = strlen(s);

  if (N==1 && strncmp(s, corpus+*suf, len)==0)
      propose(*suf);
  if (N <= 1) return;

  if (D&1) {              // popularity split
    kbest_lookup(s, suf, mid−suf, D+1);
    if (done(*mid)) return;
    if (strncmp(s, corpus + *mid, len) == 0)
      propose(*mid);
    kbest_lookup(s, mid+1, (suf+N)−mid−1,
                           D+1);}
  else {                  // lexicographic split
    int c = strncmp(s, corpus + *mid, len);
    int n = (suf+N)−mid−1;
    if (c < 0) kbest_lookup(s, suf, mid-suf, D+1);
    else if (c > 0) kbest_lookup(s, mid+1, n, D+1);
    else { kbest_lookup(s, suf, mid-suf, depth+1);
           propose(*mid);
           kbest_lookup(s, mid+1, n, D+1); }}}
```

[1] With a little extra book keeping, one can keep a table on the side that makes it possible to map back and forth between popularity rank and the actual popularity. This turns out to be useful for some applications.

2.2 A Short Example: To be or not to be

Suppose we were given the text, "to be or not to be." We could then generate the following dictionary with frequencies (popularities).

Popularity	Word
2	to
2	be
1	or
1	not

The dictionary is sorted by popularity. We treat the second column as an N=13 byte corpus (with underscores at record boundaries): to_be_or_not_

Standard		K-Best	
suf	corpus + suf[i]	suf	corpus + suf[i]
12	_	2	_be_or_not_
2	_be_or_not_	3	be_or_not_
8	_not_	4	e_or_not_
5	_or_not_	5	_or_not_
3	be_or_not_	8	_not_
4	e_or_not_	12	_
9	not_	9	not_
1	o_be_or_not_	1	o_be_or_not_
6	or_not_	6	or_not_
10	ot_	0	to_be_or_not_
7	r_not_	7	r_not_
11	t_	10	ot_
0	to_be_or_not_	11	t_

The standard suffix array is the 1st column of the table above. For illustrative convenience, we show the corresponding strings in the 2nd column. Note that the 2nd column is sorted lexicographically.

The k-best suffix array is the 3rd column with the corresponding strings in the 4th column. The first split is a lexicographic split at 9 ("not_"). On both sides of that split we have a popularity split at 5 ("_or_not_") and 7 ("r_not_"). (Recall that relative popularity depends on corpus position.) Following there are 4 lexicographic splits, and so on.

If k-best lookup were given the query string *s* = "o," then it would find 1 (o_be_or_not_), 6 (or_not_) and 10 (ot_) as the best choices (in that order). The first split is a lexicographic split. All the matches are below 9 (not_). The next split is on popularity. The matches above this split (1&6) are as popular as the matches below this split (10).

It is often desirable to output matching records (rather than suffixes). Records are output in popularity order. The actual popularity can be output, using the side table mentioned in footnote 1:

Popularity	Record
2	to
1	or
1	not

2.3 Time and Space Complexity

The space requirements are the same for both standard and k-best suffix arrays. Both indexes are permutations of the same suffixes.

The time requirements are quite different. Standard suffix arrays were designed to find all matches, not the k-best. Standard suffix arrays can find all matches in O(*log N*) time. However, if we attempt to use standard suffix arrays to find the k-best, something they were not designed to do, then it could take a long time to sort through the worst case (an embarrassment of riches with lots of matches). When the query matches every string in the dictionary, standard suffix arrays do not help us find the best matches. K-best suffix arrays were designed to handle an embarrassment of riches, which is quite common, especially when the substring *s* is short. Each popularity split cuts the search space in half when there are lots of lexicographic matches.

The best case for k-best suffix arrays is when the popularity splits always work in our favor and we never have to search the less popular half. The worst case is when the popularity splits always fail, such as when the query string *s* is not in the corpus. In this case, we must always check both the popular half and the unpopular half at each split, since the failure to find a lexicographic match in the first tells us nothing about the existence of matches in the second.

Asymptotically, k-best lookup takes between log N and sqrt N time. To see this complexity result, let P(N) be the work to process N items starting with a popularity splits and let L(N) be the work to process N items starting with a lexicographic splits.

Thus,

$P(N) = \alpha L(N/2) + C_1$
$L(N) = P(N/2) + C_2$

where $\alpha = 2-p$, when p is the probability that the popular half contains sufficient matches. α lies between 1 (best case) and 2 (worst case). C_1 and C_2 are constants. Thus,

$$P(N) = \alpha\, P(N/4) + C \quad (1)$$

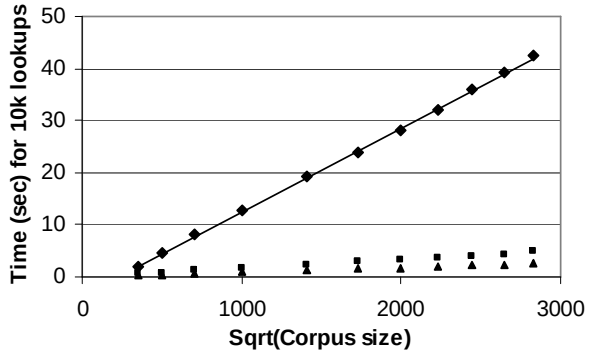
where $C = C_1 + \alpha C_2$. Using the master method (Cormen *et al*, 2001), $P(N) = O(log_2 N)$ in the best case (α=1). In the worst case (α=2), $P(N) = O(sqrt\ N)$. In general, for $\alpha > 1$, $P(N) = O(N^{(log_2 \alpha)/2})$.

In practical applications, we expect popularity splits to work more often than not, and therefore we expect the typical case to be closer to the best case than the worst case.

3 Empirical Study

The plot below shows the k-best lookup time as a function of square root of corpus size. We extracted sub-corpora from a 150 MB collection of 8M queries, sorted by popularity, according to the logs from Microsoft www.live.com. All experiments were performed on a Pentium 4, 3.2GHz dual processor machine with enough memory to avoid paging.

The line of diamonds shows the worst case, where we the query string is not in the index. Note that the diamonds fit the regression line quite well, confirming the theory in the previous section: The worst case lookup is *O*(*sqrt N*).



To simulate a more typical scenario, we constructed random samples of queries by popularity, represented by squares in the figure. Note that the squares are well below the line, demonstrating that these queries are considerably easier than the worst case.

K-best suffix arrays have been used in auto-complete applications (Church and Thiesson, 2005). The triangles with the fastest lookup times demonstrate the effectiveness of the index for this application. We started with the random sample above, but replaced each query q in the sample with a substring of q (of random size).

4 Conclusion

A new data structure, *k-best suffix arrays*, was proposed. K-best suffix arrays are sorted by two orders, lexicographic and popularity, which make it convenient to find the most popular matches, especially when there are lots of matches. In many applications, such as the web, there are often embarrassments of riches (lots of matches).

Lookup time varies from *log N* to *sqrt N*, depending on the effectiveness of the popularity splits. In the best case (e.g., very short query strings that match nearly everything), the popularity splits work nearly every time and we rarely have to search the less popular side of a popularity split. In this case, the time is close to *log N*. On the other hand, in the worst case (e.g., query strings that match nothing), the popularity splits never work, and we always have to search both sides of a popularity split. In this case, lookup time is *sqrt N*. In many cases, popularity splits work more often than not, and therefore, performance is closer to *log N* than *sqrt N*.

References

Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM*, 18:9, pp. 509-517.

Kenneth Church and Bo Thiesson. 2005. The Wild Thing, *ACL*, pp. 93-96.

Udi Manber and Gene Myers. 1990. Suffix Arrays: A New Method for On-line String Searches, *SODA*, pp. 319-327.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. Introduction to Algorithms, Second Edition. *MIT Press and McGraw-Hill*, pp.73–90.