APPROXIMATE MULTIPLE STRING SEARCH

Robert Muth and Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
{muth | udi}@cs.arizona.edu

Abstract. This paper presents a fast algorithm for searching a large text for multiple strings allowing one error. On a fast workstation, the algorithm can process a megabyte of text searching for 1000 patterns (with one error) in less than a second. Although we combine several interesting techniques, overall the algorithm is not deep theoretically. The emphasis of this paper is on the experimental side of algorithm design. We show the importance of careful design, experimentation, and utilization of current architectures. In particular, we discuss the issues of locality and cache performance, fast hash functions, and incremental hashing techniques. We introduce the notion of *two-level hashing*, which utilizes cache behavior to speed up hashing, especially in cases where unsuccessful searches are not uncommon. Two-level hashing may be useful for many other applications. The end result is also interesting by itself. We show that multiple search with one error is fast enough for most text applications.

1. Introduction

The problem we address is easy to define: Given a set of patterns P_1, P_2, ... P_k and a large text $T = t_1, t_2, \ldots t_n$, we want to find all occurrences of all patterns in the text, allowing for one error (insertion, deletion, or substitution). The problem arises in some applications of DNA searching, OCR, and other pattern matching, but the

---

[1] Supported in part by NSF grant CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

emphasis of this paper is on regular text filtering. We assume that the text is an English text, for example a collection of newswire stories, and the patterns are regular strings, for example, names. One wants to filter all relevant stories, and allowing misspellings is important.

We have in the past designed and implemented algorithms for fast approximate string searching (agrep [WM92]), and fast multiple string searching ([WM93], also available as part of the agrep package). We received many requests for a combination of the two, and this paper is a first attempt to answer these requests, at least with one error. (We also designed algorithms to search with one error in a fixed text [MW94], which addressed a particular security application.) We are not familiar with any other comparable algorithms, except for the notion of ''fuzzy search'' which several information retrieval packages employ, which typically uses heuristics to identify close spellings, but cannot guarantee following the exact definitions of insertion/deletion/substitution. These heuristics also typically work within words rather than arbitrary strings, and they use a fixed dictionary to help with the heuristics.

This paper is organized as follows. We first describe the basic algorithm and show some experimental results. This algorithm is faster than we expected, but not fast enough. We then describe several improvements and show their merits, some of which are rather surprising. The final algorithm is more than 3 times faster than the first one. Its performance is studied in detail in Section 5. The last section presents some general conclusions already learned from this work.

2. The Basic Algorithm

The basic algorithm follows ideas from Mor and Fraenkel [MF82] (some of which we rediscovered in [MW94]). Let P be the number of patterns, and n the size of the text. We want to find all occurrences of all patterns with one insertion, deletion, or substitution error.

We cut all patterns to an equal length by taking their prefixes of size r ($r = 6$ proved best often, but not always). Then, for each pattern A, we take all possible combinations of A with one deletion. For example, if the pattern is ''example'' and we decided on prefixes of size 6, the prefix is ''exampl''; we take the 6 strings (each of size 5) ''xampl,'' ''eampl,'' ''exmpl,'' ''exapl,'' ''examl,'' and ''examp.'' We compute the hash values of these strings and store a pointer to the original full length string in the hash table. The number of entries in the table will be $r \cdot P$.

We then look at the text r characters at a time. For each subtext of size r, we follow the same procedure — construct the r strings resulting from deleting one character at a time — and check whether any of the resulting strings appear in the hash table. A deletion error in the pattern will, of course, be directly discovered, but in addition (as is shown in [MF82]) an insertion error corresponds to a deletion in the text, and a substitution error corresponds to both deletions, so all errors will be discovered by this method. If none of the patterns appears in the hash table, then there is no match, we shift by one, and continue. If any of the strings appear in the hash table, then there may be a match and we verify this directly.

Overall, this is an $O(n)$ algorithm if we consider r to be a constant and we assume that the verification step is also a constant-time procedure (or that it is rare enough). (Of course, in the worst case, the verification stage can take a long time if many of the patterns are almost equal and all have to be checked often, but this will not occur in practice for English text.) For much longer strings, r would not have to be much longer (for the same alphabet size), to make the verification step sufficiently rare. (For example, there are 24.3 million 5-letter combinations in an alphabet of size 30. With 1000 patterns, each generating 5 strings, we still touch on a very small percentage.) For smaller-sized alphabets, we will have to rethink some of the settings, but as we said, we concentrate in this paper only on English text.

As we mentioned, this algorithm is rather slow, although not too slow. Our first implementation, with 1000 patterns, already beats by a factor of 2 the standard egrep package, with only 25 patterns (egrep cannot handle more than 500 patterns) even though egrep does not allow errors! This implementation is not competitive, however, with very fast multiple string matching without errors, such as the ones used in agrep [WM92] or in Gnu-grep [Ha93] (which uses an algorithm based on Commentz-Walter [CW79]). We cannot hope to compete with string matching without errors, but we can hope to come close, which we did. Next, we will show how to improve this basic scheme. We call this basic algorithm *Algorithm 00*.

3. Better and Faster Hash Functions

The choice of a hash function is a critical choice in any implementation. In our experience, this is one of the most common causes of bad performance and sometimes incorrectness. Selecting hash functions involves a basic tradeoff. On the one hand, we want to minimize collisions by designing the hash functions to generate as random results as possible. But on the other hand, computing hash values is typically

done in the most inner loop of the algorithm, therefore it is essential to do it efficiently. The less time spent on computing hash functions the more collisions or other overheads will be incurred. But any overhead in computing the hash values themselves will be incurred all the time. We need to strike the right balance. While there has been extensive research on the organization of hash tables — in particular, collision resolution schemes — there has not been sufficient work on the tradeoffs of selecting hash functions. This area is still mostly ad-hoc art.

The first hash function we used is similar to the one used in agrep, where it was found to strike the right balance between minimizing collisions and maximizing speed. It is very simple. The characters of the pattern are combined into one value by shifting and adding as we add more characters. How much to shift depends on the size of the string and the size of the hash function. For example, for strings $x_1x_2x_3x_4x_5$ of size 5 (which corresponds to many of the experiments we ran) and hash tables of size 2^{20} the function is

$$Hash = (x_1 << 16) + (x_2 << 12) + (x_3 << 8) + (x_4 << 4) + (x_5)$$

where ''<<*x*'' stands for ''shift *x* bits to the left.''

We improved this scheme in two ways. First, instead of taking the bits directly out of each character, we added a step of randomization. We first constructed a table of size 256 mapping each character to a random number. Then, whenever we read a character we first consult the table and take its random number. Then, the numbers of all the characters are shifted and xor'ed together to form the hash value. Let R[x] be the random number associated with character x. The hash value for $x_1x_2x_3x_4x_5$ is

$$Hash = ((((R[x_1] << 1) \hat{} R[x_2]) << 1) \hat{} R[x_3]) << 1) \hat{} R[x_4]) << 1) \hat{} R[x_5]$$

(where ^ is the XOR function). The added overhead was minimal — one indirect addressing — and the extra randomization was effective. (This trick is quite known in the ''folklore,'' but we are not sure who invented it first.)

The second improvement was much more complex. It comes from the fact that the 6 strings that are generated by one string in the text (after deleting one character at a time) are not random. They are quite similar to one another. Instead of computing each hash function from scratch, we utilize the similarities and manage to sort of ''cut and paste'' different parts of the hash functions to minimize total running time. Not only that, but the strings that are generated in the next step, after shifting by one character, are also quite similar, and we utilize that too. We will describe the idea through an example.

Let's assume again that $r = 6$, and that the text starts with abcdefg. The first string we need to consider is abcdef, which means that we need to compute Hash(''abcde''), Hash(''abcdf''), Hash(''abcef''), Hash(''abdef''), Hash(''acdef''), and Hash(''bcdef''). The next string will be bcdefg, which requires Hash(''bcdef''), Hash(''bcdeg''), Hash(''bcdfg''), Hash(''bcefg''), Hash(''bdefg'') and Hash(''cdefg''). Notice that Hash(''bcdef'') appears in both of these windows, and needs to be evaluated only once. Therefore, we need to evaluate hashing for only 5 strings per window. Assume that the following variables are initialized as stated:

hash = Hash(''abcde'')
x01 = (R['a'] ^ R['b']) << 4
x12 = (R['b'] ^ R['c']) << 3
x23 = (R['c'] ^ R['d']) << 2
x34 = (R['d'] ^ R['e']) << 1
x45 = (R['e'] ^ R['f']) << 0
x5 = R['f']

Now we can quickly compute the other hash values for the current window:

Hash(''abcdf'') = hash ^ x45
Hash(''abcef'') = hash ^ x45 ^ x34
Hash(''abdef'') = hash ^ x45 ^ x34 ^ x23
Hash(''acdef'') = hash ^ x45 ^ x34 ^ x23 ^ x12

More importantly we also can easily compute the initial values for the next window:

hash' = hash ^ x45 ^ x34 ^ x23 ^ x12 ^ x01
x01' = x12 << 1
x12' = x23 << 1
x23' = x34 << 1
x34' = x45 << 1
x45' = x5 ^ R['g']
x5' = R['g']

Overall, even though each hash function is of a string of size *r*, we compute all *r* hash values with only two XOR's and one shift per value. In essence, instead of spending $O(r^2)$ time computing *r* hash functions of strings of size *r*, we do it in time $O(r)$. Performance results of these two improvements, which yield Algorithm 01, are shown in Section 5. Surprisingly, they do not improve algorithm 00 significantly. But they are much more effective in combination with the other improvement.

4. Two-Level Hashing: Using Locality to Improve Hashing

Improving the hash functions was very helpful but not as much as the next method, which is counter-intuitive at first. We call this method *two-level hashing*. We have not seen it in print, although it is likely to have been used before by others. Instead of one hash table, we use two of them, with the first being only a bitmap, such that each bit is set to 1 iff something is mapped to the corresponding entry in the ''real'' hash table. Let's call the first table the ''bitmap table'' and the second one the ''real'' table. Instead of going directly to the original hash table, we first consult the bitmap table. If the bit is 0, we know that the string is not in the hash table. If the bit is 1, we still have to go to the real table to find the corresponding pattern. It seems that we keep the same amount of work for unsuccessful queries and double the work for successful queries, but in fact we make unsuccessful queries, which are much more common, significantly faster.

There are two reasons for the speedup. First and most important, the bitmap table is much smaller than the real table. The real table requires, in each entry, a pointer to the string that was mapped into that entry. We use a 64-bit machine, so the bitmap table is 64 times smaller. Since the table is smaller, there is a much better chance that it will be kept in the cache. Therefore, accessing the bitmap table can be an order of magnitude faster than accessing the real table. This by itself gives more than a 2-fold speedup for the whole algorithm, even taking into account that for successful queries the access is slower (because of the need for two levels). Furthermore, the less patterns the more speedup, because the more chance we have for unsuccessful queries. We present several experiments on the effects of the cache here, and we expect to expand on that in future work. We believe that a better understanding of the effects of today's large caches is sorely missing from the analysis of algorithms.

The second reason is that secondary collisions can be avoided in the bitmap table. Since the use of chaining requires an even larger hash table, we use linear probing which leads to secondary collisions. If the bitmap table has at least as many entries as the real table, then secondary collisions in the real table would not affect the bitmap table. This was one of our first reasons to use a bitmap table, but the experiments showed an interesting effect. Once it became clear that the total size of the tables affects performance much more than extra collisions, we experimented with compressing the bitmap table even more by allocating one bit to several entries in the real table (and setting that bit to 1 iff any of the original entries is set). In

Section 5, we will show that such compressed tables often gave the best performance.

Incorporating this change into algorithm 00, but without changing the hash functions, yields algorithm 10. It presents a two-fold improvement in running times. Adding the new hash functions to algorithm 10, giving the final algorithm which we call algorithm 11, gives more than 3-fold improvement.

5. Detailed Experiments

We performed many experiments, studying the effects of hash table sizes (both of them in the case of two-level hashing), the number of patterns, the sizes of the prefixes taken from each pattern, different types of texts, and different machines. (Not all results are listed in this paper.) We present the main results in order of importance. (A general note: grep-type programs output lines where matches occur. We have found in the past that this can skew performance results, because when matches occur in the beginning of a line the rest of the line is just skipped. Therefore, more patterns can lead to more matches, which in turn may seem to lead to faster running times. In all the tests we performed here, we continued the matching process after finding a match, not skipping the rest of the lines. The results here are therefore somewhat slower than actual results for the real implementation of this program.)

Table 1 presents the running times of the algorithms with the different improvements. These are in a sense the bottom line (which is why we put them on top). We used a DEC AlphaStation 200 4/233 (233 Mhz) with 96 MB of memory and 1MB of second-level cache. The text was a collection of articles from the Wall Street Journal of size exactly 10MB. We selected 1000 patterns at random from the text. (If we had chosen some patterns that do not appear in the text, which is more realistic, the running times would have been better; we decided to remain as conservative as we could.) The times are given in seconds, and they are the elapsed times given by UNIX time routine (‘‘user times’’ were very similar). The instructions field counts the total number of instructions computed by a tool called ATOM made by Digital (on a separate run, of course). Recall that algorithm 00 is the basic one, algorithm 01 employs better hash functions, algorithm 10 employs two-level hashing (with the same hash functions as algorithm 00), and algorithm 11 employs both techniques together. The hash table sizes are the ones that gave the best performance for algorithm 11.

Algorithm	Hash table size	Hash bitmap	seconds	instructions
00	2^{19}	-	25.62	1,671,775,504
01	2^{19}	-	24.70	834,559,519
10	2^{19}	2^{17}	12.85	1,753,496,436
11	2^{19}	2^{17}	7.32	991,896,252

Table 1: Running times and instruction counts for multiple approximate matching for 1000 patterns on 10MB text

Looking only at running times, we see that having better hash functions does not make a big difference going from algorithm 00 to 01. This may be surprising because there are clearly less calculations in algorithm 01. The reason becomes clear when we look at the instruction counts. As expected, they drop by almost a half going from algorithm 00 to 01. This did not affect the running times significantly, because the dominant factor was not the CPU but out-of-cache access.[2] Algorithm 10, which does not employ the better hash functions but does employ the two-level hash tables leads to *more* instructions but *better* running times. The better hash functions do make a difference when going from algorithm 10 to algorithm 11.

The other interesting results are those that study the effects of the caches. In Table 2, we vary the sizes of the hash tables, both the real one (indicated by h1, one per row) and the bitmap (indicated by h2, one per column). It is clear that when both tables become too big, there is a performance penalty even though there are less collisions in the hash table. These runs were made with algorithm 11, same text, patterns, and machine as the one for table 1. The sizes of the hash tables are given in logarithms base 2.

It is not easy to identify all the different factors that affect the performance. For example, since we use English text, which is not random, it is possible that some strings appear quite often leading to better performance because the corresponding

[2] Another factor may have been the fact that the hash functions in algorithm 01 are also better in the sense that they spread the hash values more evenly. The number of empty hash locations was significantly higher for algorithm 00 than it was for 01, which means that there were more opportunities for caching in algorithm 00. Ironically, a worse hash function can sometimes actually lead to *better* running times, which we observed in some instances comparing algorithm 00 to 01 for smaller number of patterns. The differences, however, were small.

h1 | h2	15	16	17	18	19	20	21
15	11.17	08.70	07.82	07.92	08.19	08.55	10.21
16	08.86	08.50	07.77	07.97	08.21	08.44	11.06
17	08.70	08.06	07.76	08.49	08.44	08.55	11.38
18	08.73	07.83	07.50	07.98	08.17	09.56	10.81
19	08.97	07.83	07.32	07.80	08.14	08.58	13.15
20	09.13	07.94	07.72	07.90	08.33	08.95	11.71
21	09.39	08.19	07.54	08.10	08.50	10.36	13.13
22	10.95	09.56	08.84	09.59	09.91	11.47	12.91

Table 2: Running times for different hash table sizes

hash entries were more often in the cache. On the other hand, the patterns are not random either, and common strings in the patterns will lead to performance penalties because they will require more verifications. Therefore, we ran two additional experiments: one which skips the verification stages and one with completely random text and patterns. The results of these tests, which are not presented here for lack of space, were quite similar to those presented in Table 2, although some jumps were more clear. For example, with 100 random patterns, using algorithm 00 with no verification, it was about twice as slow (18.2 seconds vs. 9.6) to use a hash table of size 2^{17} compared with one of size 2^{16} (the same experiment with real text yielded a difference of only about 40% — 13.45 vs. 9.46). This is consistent with the fact that the second-level cache was 1MB, which accommodates a table of size 2^{16} (each entry is 8 bytes since this is a 64-bit machine) comfortably, but runs over (since it probably caches some pieces of the text as well) with size 2^{17}.

To study these effects further, we also used tools to count the total of instructions for all runs. Figure 3 shows how changing the size of the bitmap table (for a fixed real table) affects the running times (above, scale on the left) and number of instructions (below, scale on the right). Although the number of instructions does not change by much, the running time increases when the hash tables become too large, because of cache and memory performances. (The small spike in the number of instructions going from tables of size 2^{16} to 2^{17} can be explained by a possible added

instruction needed to deal with numbers greater than 2^{16}.)

Figure 4 shows the running times for different number of patterns, ranging from 25 to 800. Since the running time is dominated by the hashing, more patterns do not contribute too much until the number of hits become significant, in which case the time spent on verification starts to dominate. We have not yet put sufficient effort into trying to optimize the algorithm to beyond 1000 patterns.

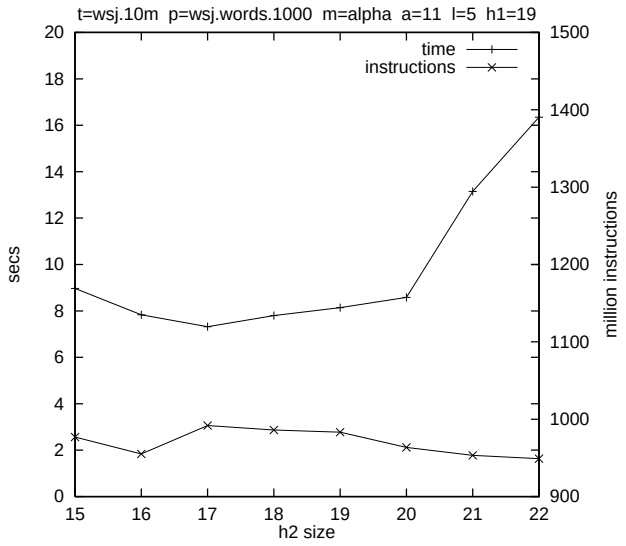

Figure 3: Running times and number of instructions for different bitmap sizes

6. Conclusions

The effects of locality are extremely important; this is, of course, well known, although not yet sufficiently studied in the algorithms area ([AACS87] was a very good start). The differences, in today's architectures, between fetching from first-level cache and fetching from main memory can be 100 fold. The sizes of memory caches (and in mainframe computers disk caches as well) have grown significantly in recent years, and the gap between cache access time and memory access time has grown too. It was very surprising, at least to us, to see how much this affects hashing, even though due to the randomness of hashing, it was thought to be immune to locality. Caches just grew in size to a degree that they can accommodate reasonably large-size hash tables, and it is important to take this into account. We believe this observation is especially important in the string matching area, because using hashing

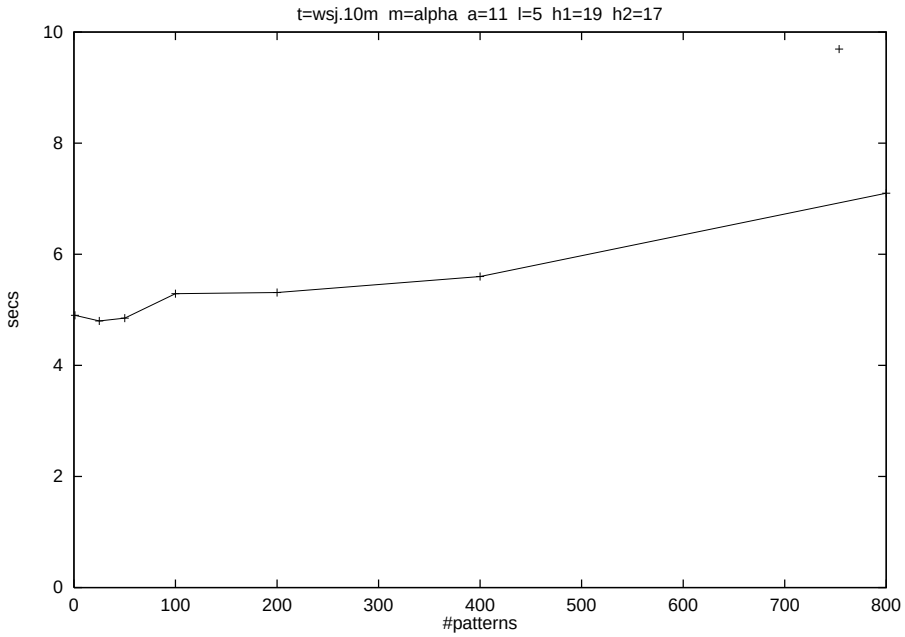

Figure 4: Running times with different number of patterns

on strings of size 4-6 is very common, and this kind of hashing leads to hash tables of the size we handled in this paper. As we have shown, it may be time to adjust other algorithms to current architectures, which may change some policies and folklore.

The final algorithm we presented is fast enough, we believe, to make multiple search with one error feasible for most text applications. This was not obvious to us when we started this work.

7. References

[AACS1987] Aggarwal A., B. Alpern, A. K. Chandra, and M. Snir, ‘‘A Model for Hierarchical Memory,’’ *ACM Symposium on Theory of Computing*, New York City (May 1987), pp. 305–314.

[CW79] Commentz-Walter, B, ‘‘A string matching algorithm fast on the average,’’ *Proc. 6th International Colloquium on Automata, Languages, and Programming* (1979), pp. 118–132.

[Ha93] Haertel, M., ‘‘Gnugrep-2.0,’’ Usenet archive *comp.sources.reviewed*, Volume 3 (July, 1993).

[KMP77] Knuth D. E., J. H. Morris, and V. R. Pratt, ‘‘Fast pattern matching in strings,’’ *SIAM Journal on Computing* **6** (June 1977), pp. 323–350.

[MF82] Mor M., and S. Fraenkel, ‘‘A Hash Code Method for Detecting and Correcting Spelling Errors,’’ *Comm. of the ACM*, **25** (December 1982), pp. 935–938.

[MW94] Manber U., and S. Wu, ‘‘An Algorithm for Approximate Membership Checking With Application to Password Security,’’ *Information Processing Letters* **50** (May 1994), pp. 191–197.

[WM92a] Wu S., and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ *Usenix Winter 1992 Technical Conference*, San Francisco (January 1992), pp. 153–162.

[WM92b] Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the ACM* **35** (October 1992), pp. 83–91.

[WM93] Wu S., and U. Manber, ‘‘A Fast Algorithm for Multi-Pattern Searching,’’ Technical Report TR-94-17, Department of Computer Science, University of Arizona (May 1993).