Some Perils of Performance Prediction : a case study on Pattern Matching

Peter Fenwick
Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
email : p.fenwick@auckland.ac.nz

October 29, 2007

Abstract

Recent experience with string searching or pattern matching algorithms revealed wide discrepancies between predicted and observed performance. Further observations on a variety of computers revealed even greater variations between those algorithms on different computers. These observations are collected here as an example of problems in real-world comparison of algorithms.

Keywords string-searching, pattern matching, timings, performance, prediction, observation

1 Introduction

A recent paper[1] presented a new string searching or pattern matching algorithm of high performance and concluded with some experimental measurements to confirm that performance. Perhaps unfortunately, but after the paper was submitted, the timing experiments were repeated on several other computers. These gave rather different ratios between predicted times and observed speeds. It was clear that comparisons which were valid on one computer were much less valid on others; in some cases

they were significantly different and quite inapplicable to the different environment.

This paper presents some observations on those results. Although it does include some plausible analysis of the results, its main intention is cautionary, to show that algorithm performance can be very difficult to predict. It is all too easy to make predictions which are quite at variance with observed performance. It is also easy to assume that relative performance on one computer will apply to another computer. Perhaps even worse it is possible to optimise away the worst case (which seldom if ever occurs) at the cost of penalising the usual case.

This paper is *not* meant to compare different string searching or pattern matching algorithms; neither does it pretend to test examples of best current practice. The important point is that *ratios* of execution times on one computer do not necessarily translate to another computer. The actual times and relative merits of the algorithms are very much a secondary consideration.

2 Summary of the search algorithms.

We now give a summary of the algorithms used for the tests. For more details of the algorithms, readers should refer to the paper by Fenwick[1], and the report by Charras and Lecroq[2] or the book by Stephen[3]. In the descriptions we take a *text* string of n characters (which may be hundreds of kilobytes), and search it for occurrences of a *pattern* of m bytes, perhaps a dozen bytes or less.

Simple (Sometimes called "Brute Force" or "Naive".) In this simple and obvious algorithm, the text is searched for occurrences of the first character of the pattern. Whenever a match is found, the following characters are compared against those of the pattern. The standard analysis is that in the worst case we may get a match on each of the n text characters and at each position have to initiate a comparison to the full pattern length of m characters, for a total of about mn character comparisons.

Knuth-Morris-Pratt A problem with the Simple algorithm is that overlapping matches may force repeated comparisons of portions of the text.

This problem is addressed by the Knuth-Morris-Pratt algorithm[4] which constructs a control array from the pattern. In the event of a partially matched pattern, it allows comparisons to be resumed without back tracking.

Boyer-Moore-Horspool The BMH algorithm[5] also constructs a control table, but one which allows the text to be examined at only every m-th character. The number of text characters examined is then of the order of n/m.

Digram The new digram algorithm is rather more complex and readers should consult the original paper[1] for details. The algorithm proceeds in several phases –

- The entire text is read in to a buffer.
- Links are constructed to connect occurrences of each digram or character pair.
- For each search an appropriate digram is selected from the pattern. That list is traversed as an initial estimate of likely search targets. Areas of text which do not contain the chosen digram need never be examined for possible matches.
- Strategically selected "guard" characters from the pattern are compared against corresponding characters in the text.
- Only after a potential match survives both digram and guard tests is it fully examined for a complete match.

This algorithm was noted as being typically two orders of magnitude "better" than the others, as judged by the number of character comparisons. The extra storage over other algorithms is about 256k bytes for the digram table and 4 bytes per text character, assuming 4-byte integers.

The traditional figure-of-merit in pattern matching is the number of character comparisons in searching the text. It is an obvious metric, easily measured, and independent of the computer. Unfortunately, we will see that it has little relationship to execution speed.

3 The test computers

It is important to remember that programs are not just "run on computers" but are processed by compilers and subject to optimisation. Caches, pipelining and instruction parallelism provide another form of optimisation, but one largely uncontrollable by the user. Tests were run on various computers and at different compiler optimisation levels. In one case the cache could be disabled, as a way of pessimising the performance . The differences *between* computers are not that important, but we can certainly see the relative effects on different algorithms as optimisation and other parameters are changed for a single computer.

- Macintosh 540C Powerbook, 60 MHz 68040LC processor. Programs were compiled with the Metrowerks 1.4 C compiler, with peephole optimisation. One set of tests was run with the cache disabled, giving a computer of much lower performance and probably fewer runtime idiosyncracies.

- Macintosh G4, 400MHz PowerPC processor. The compiler and runtime system was Metrowerks 4.0, using three levels of optimisation. The first two were the minimum and maximum levels of machine independent optimisation, while the third (noted as "Altivec") added processor specific optimisations to the full level.

- DEC/Compaq Alpha processors, one with a 275 MHz clock and the other 533 MHz. Both were used with the GNU gcc compiler, at optimisation levels of 0 and 3, using identical code files for the two computers.

- A Pentium-PRO processor, with the Microsoft C/C++ compiler.

- An SGI "Indy" system, with 100MHz R4000 processor, 64MB RAM, and caches 2×8k + 1MByte. Its GNU gcc compiler had optimisation levels of 0 and 2.

- A 550MHz Pentium III, 512 Mbyte RAM, caches 2×1k + 512k byte, running Linux with gcc compiler and optimisation levels 0 and 3.

	Read	Link	Simple	KMP	BMH	Digram
540C no cache	567	712	159.34	367.84	70.52	0.6665
540C	142	134	19.173	72.429	16.514	0.1519
G4min	11	4	1.087	3.042	0.748	0.0117
G4full	6	4	0.829	2.560	0.664	0.0110
G4altivec	6	3	0.846	2.216	0.630	0.0109
Alpha-275 opt0	416,650	916,630	155,693	346,492	74,836	2,429.6
Alpha-275 opt3	133,328	249,990	49,729	109,189	34,192	1,318.3
Alpha-533 opt0	199,992	399,984	68,008	173,111	31,397	1,405.7
Alpha-533 opt3	66,664	66,664	16,827	41,826	13,180	1,272.7
Pentium-pro	31	47	6.23	8.41	2.96	0.0450
SGI INDY opt0	810,000	1,780,000	289,935	789,871	165,258	5,413
SGI INDY opt2	440,000	820,000	110,935	96,032	60,871	3,179
550 Pent opt0	140,000	110,000	14,161	37,581	12,484	792.69
550 Pent opt3	90,000	90,000	11,008	16,742	8,591	674.29

Table 1: Raw time in clock units, as reported by each system

4 Test procedure

In all cases the "text" is the file "book1" from the Calgary Compression corpus which is searched for about 30 words selected from early paragraphs. Times are initially measured in clock ticks for the subject computer. The actual time unit is irrelevant as we take only the ratio of times for each computer and do not compare times between computers. (Some computers used a 1/60 second clock tick, and others a 1μs tick.)

The combination of long clock periods and fast computers meant that many tests completed within just few clock ticks, or even within one clock tick, making timings and comparisons very difficult. Each search on a pattern was repeated until at least 10 clock ticks had been accumulated. The whole suite of searches was repeated 10 times so that the final times are based on at least 100 ticks for each pattern, or about 3000 ticks for the whole test. Table 1 shows times for actual computers. Note that there is no attempt to reduce these times to seconds or other compatible units; all are left in system-specific units. Apart from the times for the four test algorithms, two columns show other important times –

Read This is the time required to read the file into memory before starting the actual searches. It is an unavoidable overhead which must be

included in the search time if a file is to be searched for only one or two patterns. The read time is determined by the input-output system and software. It is outside the control of the user.

Link The digram algorithm has a phase, following or concurrent with reading, which constructs the digram links to facilitate fast traversal of the text. It is only if the initial linking time can be amortised over perhaps several searches that the digram algorithm really shows to advantage over Simple.

The raw data of Table 1 are of little importance in their own right. The more useful information is in Table 2 where all values are scaled relative to the Simple search for that computer. The linking cost is shown as a time relative to the Simple search, with times equivalent to 3.5–7.6 searches. This means that the digram algorithm, the only one which needs the preliminary linking phase, is useful only if more than 5–10 searches are needed on the same data. Fewer searches are better done with one of the simpler algorithms. The search algorithms themselves are given as the *speeds* relative to Simple.

In the "ideal" case (perhaps more correctly called the "naive" case) the values in each column would be the same, or at least similar with each algorithm have a well-defined performance relative to the others. But the values are far from similar, showing that comparisons on one computer do not scale to other computers.

Thus we see that the "improved" Knuth-Morris-Pratt algorithm is only about 25% – 75% the speed of the Simple algorithm which it is supposed to better. The Boyer-Moore-Horspool algorithm is somewhat faster than Simple, but certainly not the 4 times improvement that the ratio of 25% as many character comparisons would imply (see [1] for details). The new, "digram", algorithm shows the widest variations. In some cases its improvement over Simple is about 240 times, while in others it is only about 13 times better, a variation of 18:1 in improvement.

The last two lines of Table 2 are based on the "obvious" metric of character comparisons and provide the figure of merit which is often published. The penultimate line gives the actual number of comparisons, while the last line gives the predicted improvement over Simple. The ratio of comparisons has little relation to the ratio of measured times except to note that the observed improvement is seldom better than one half of the improvement predicted by counting character comparisons.

	Link (Time)	Simple	KMP	BMH	Digram
			(speeds)		
540C no cache	4.47		0.43	2.26	239.09
540C	6.99		0.26	1.16	126.21
G4min	3.68		0.36	1.45	92.85
G4full	4.82		0.32	1.25	75.56
G4altivec	3.55		0.38	1.34	77.73
Alpha-275 opt0	5.89		0.45	2.08	64.08
Alpha-275 opt3	5.03		0.46	1.45	37.72
Alpha-533 opt0	5.88		0.39	2.17	48.38
Alpha-533 opt3	3.96		0.40	1.28	13.22
Pentium-pro	7.55		0.74	2.10	138.37
SGI INDY opt0	6.14		0.37	1.75	53.56
SGI INDY opt2	7.39		0.57	1.82	34.89
550 Pent opt0	7.77		0.38	1.13	17.86
550 Pent opt3	8.18		0.66	1.28	16.33
Character Compares		24,678,938	24,564,952	6,066,659	63,929
Improve over Simple		—	1.0046	4.068	386.04

Table 2: Times and speeds, relative to "Simple"

5 Discussion

In this discussion we abstract the essential and costly details of the inner loops of the algorithms. While purists might claim that the algorithm is not like that at all, we assert that the code shown is equivalent to what is relevant.

Simple The main loop is just a reading and comparison of two characters; one reference is constant and the other steps uniformly through the text array.

```
for (textIx = 0; textIx < textLen-pattLen; textIx++)
  if (text[textIx] == pattern[0])
    .....   // full string comparison
```

Both operations are good candidates for compiler optimisation as is shown by the times in Table 1. The two DEC Alpha computers are particularly impressive here, with speeds improving by 3 or 4 times

with full optimisation. This improvement of course penalises other algorithms which are measured with respect to the improved times under optimisation.

KMP The Knuth-Morris-Pratt algorithm innermost loop has the general form –

```
while (pattern[patIx] != text[textIx++])
  patIx = next[patIx];
```

While there is other essential code in the algorithm, it is this loop or code of similar nature which performs most of the searching. It requires three array accesses per iteration, with subscript patterns which are difficult to optimise. This effect is seen in the relatively poor performance, worse than the Simple algorithm. Knuth-Morris-Pratt can work much better than Simple in the worst case, but the worst case seldom occurs with normal text.

BMH The Boyer-Moore-Horspool algorithm is based on the loop –

```
while (text[textIx] != pattern[patIx])
  textIx += skip[text[textIx]];
.....   // full string comparison
```

The problem here is that the code `skip[text[textIx]]` involves *two* array references. Except for the repeating reference to `pattern[patIx]` none of the array references can be optimised and their irregular nature means that they are unlikely to get much cache assistance. The decreased number of *character* comparisons as compared with Simple is partly at the expense of extra arbitrary array references. As a skip will often stride right out of a cache line, each comparison may require a full memory reference and gain no benefit from the cache. While in this case the performance was always better than Simple, the improvement is not large and certainly not as large as a count of character comparisons might indicate.

Digram With its linked lists of digrams, this algorithm has completely unpredictable memory reference patterns, which will defeat most

	Simple	K M P	B M H	Digram
G4	1.311	1.188	1.127	1.064
Alpha-275	3.131	3.173	2.189	1.843
Alpha-533	4.042	4.139	2.382	1.104
SGI INDY	2.613	8.225	2.715	1.703
Pentium-550	1.286	2.245	1.453	1.176

Table 3: Speed gains : best optimisation vs least optimisation

caches. There is certainly little spatial locality (at least in the main loop) and with 3.9Mbyte of data structure for the Book1 file, probably little scope for temporal locality either. Although the digram algorithm has a *very* impressive performance from counting character comparisons, much of the gain is cancelled by its expensive memory references.

The whole matter of which algorithm is the best one, and by "how much" it is better, is very complex and difficult to answer. Certainly the basic Simple search seems to suit most computers and in its simplicity works very well on most real data. More complex searches require more memory references, with corresponding reductions in performance especially if the reference patterns are unpredictable.

Another aspect which we can compare is the improvement of algorithms under optimisation. Again we take each computer in isolation, comparing its speed with best optimisation against its speed with least optimisation. (Some other comparisons can be drawn from the data of Table 1 but are not included.) While we include the Macintosh G4 in this comparison, the most telling comparisons are for the two DEC Alpha computers, shown in Table 3. Here the "opt0" and "opt3" codefiles were the same for the two computers. Not only is the improvement different for two different versions of the same computer, but the improvement tends to be less with the"better" algorithms. The greater improvement of the simpler algorithms decreases the apparent advantage of the better ones.

6 Restricted Alphabets

Many of the algorithms are sensitive to the size of the alphabet. Furthermore, most searching or matching algorithms are demonstrated with very

small alphabets, the better to illustrate the method; with large alphabets interesting situations may be too rare for interest. Charras and Lecroq [2] use examples which seem to be related to DNA sequencing with an alphabet of 4 symbols.

- With a smaller alphabet the Simple algorithm has a greater chance of matching the initial character and entering the slower compare phase. It then slows down in comparison with a large alphabet, which increases the *apparent* link speed. It likewise increases the apparent speed of Knuth-Morris-Pratt which tends to maintain a constant speed, irrespective of the alphabet size.

- Boyer-Moore-Horspool depends on the probability of finding a particular character within the pattern and skipping if the character is not found. With a small alphabet, the probability of a particular character occurring in a given length of text increases and the BMH search is less likely to skip. More comparisons are needed and the operation should slow down.

- The digram algorithm depends on its speed for distributing digram links over many possible digrams. If the alphabet is small, there are fewer digrams and therefore fewer lists. Each active list will be longer, with more elements to traverse and characters to compare. The relative speed advantage also decreases. With DNA the situation is even worse because of the nearly equiprobable symbols. The digram lists (only 16 of them) are of nearly the same length and there is little advantage in choosing one over another.

This discussion is still based on the number of character comparisons, which is a most unreliable metric and does not translate well to execution time.

To test the operation on a smaller alphabet, the test file ("book1") and its test patterns had each character converted into its two hexadecimal digits (in ASCII) with line terminators left unchanged. Thus an ASCII 'A' $\rightarrow$ '4' and '1' (two ASCII characters), and '6' $\rightarrow$ '3' and '6'. The alphabet is thereby reduced from about 60 symbols to 16.

The tests were repeated using these converted files for two computers, both at minimum optimisation, with the results shown in Table 4[1].

[1]Tests with small alphabets could not be run on all of the computers.

	Link	KMP	BMH	digram
	(Time)	(speeds)		
G4 min	3.68	0.36	1.45	92.91
G4 hex	2.71	0.45	2.39	15.15
G4 DNA	2.72	0.50	1.47	5.71
Alpha-533 Opt0	5.88	0.39	2.17	48.38
Alpha-533 Hex	4.41	0.51	4.90	23.90
Alpha-533 DNA	7.00	0.34	1.00	4.96

Table 4: Alphabetical, hexadecimal and DNA data

While the Knuth-Morris-Pratt search shows a small change, the Boyer-Moore-Horspool search again shows a useful performance improvement. The new digram algorithm however shows a significant deterioration (6 times on G4 and 2 times on Alpha) when moving to the small-alphabet file. This is largely because there are only 256 digram chains instead of the earlier 4,096, with corresponding increases in the number of comparisons for each pattern.

For an even smaller alphabet we use a file of DNA data, about 200 kBytes from the Genbank human genome data base. The file is searched for about 30 patterns, of length 6 – 12 symbols, with the results also shown in Table 4. It is interesting to observe that the frequency of pattern matches is almost what one would expect from random data. Knuth-Morris-Pratt is still much slower than Simple. Boyer-Moore-Horspool is about 50% faster than Simple on the G4 computer, but the same speed on the Alpha, in complete contrast to its behaviour on the other two data sets. The digram algorithm is only about 5 or 6 times faster than Simple. With only 4 digram lists we would expect the search length to be 25% that of naive and still probably 1.375 character comparisons for each candidate. The probable speed-up is then 5.5 times, much in line with the results. (For good speed-up, the algorithm should probably use groups of 5 or 6 characters in forming lists, to give 1,024 or 4,096 lists.)

In none of these tests is Knuth-Morris-Pratt faster than Simple. Boyer-Moore-Horspool has rather unpredictable behaviour with rather more improvement on the hexadecimal file and little or none on DNA. While the digram algorithm is penalised by the absence of the short lists of digrams from which it really gains benefit, its performance is still in line with an

approximate prediction.

7 Conclusions

This paper has been presented as an case study in performance prediction. Its main lessons are –

- Simple prediction measures based on the *appearance* of the algorithm may be of little real value.
- The relative "qualities" of two algorithms may vary widely between computers and even between compilers or optimisation levels on a single computer. In one case a speed ratio of 240:1 on one computer became only 13:1 on another.
- Very simple algorithms are often in a form which is easily optimised by compilers or accelerated by run-time mechanisms such as caches.
- Complex algorithms tend to use more memory references, and to use them in unpredictable ways. They often get less benefit from optimisation (either software or hardware) and appear to slow down relative to the better-optimised simple algorithms.
- Algorithms may have strong data dependencies. Ones which show great benefit on some data may be much less useful on other data. Is the test data appropriate to the intended use?

In general, be very very careful when attempting to compare algorithms on the basis of convenient measures such as character comparisons, memory references, and so on. The pattern and interaction of references may be at least as important as their mere existence.

8 Acknowledgements

This work was started at the University of Auckland, and completed at the University of Vermont while the author was on Study Leave. The author thanks both institutions for their support.

References

[1] Fenwick, P.M., "Fast string matching for multiple searches", *Software – Practice and Experience* submitted Feb 2000.

[2] Charras, C. and Lecroq, T., "Exact String Matching Algorithms", Laboratoire d'Informatique de Rouen, Université de Rouen.

[3] Stephen, G.A., *String Searching Algorithms*, World Scientific, Singapore, 1994.

[4] Knuth, D.E., Morris, J.H. Jr, and Pratt V.R., "Fast Pattern Matching in Strings", *SIAM Journ. Computing*, Vol 6, No 2, pp 323–350 , June 1977

[5] Horspool, R.N., "Practical fast searching in strings", *Software – Practice and Experience*, Vol. 10, No. 6, pp 501–506, 1980.