Very fast pattern matching for highly repetitive text

David Kulp

December 22, 1992

Department of Computer Science
University of Canterbury
Christchurch
New Zealand

Abstract

This paper describes two searching methods for locating longest string matches in source texts of low entropy. A modification of the Boyer-Moore scanning algorithm and a statistical method, which searches for less likely symbols, are presented. Both algorithms have been implemented as part of the searching strategy for an LZ77 type encoder. Experimental results are included.

1 Introduction

LZ77 data compression schemes [ZL77] search through a buffer of previously encoded symbols to locate the largest string of symbols that match the current phrase of symbols to be coded. For example, given the text in Figure 1, the *lookahead buffer* contains the symbols "ABDC" to be coded, and the *window* contains the symbols which have already been coded and form the current context. An LZ77 encoder must locate the largest match in the window, which, in this case, is "ABD" at position 5. Bell and Kulp [BK] describe several data structures to accelerate searching for longest matches. Most of the data structures presented performed poorly for highly repetitive source such as bit patterns in a picture. The two techniques discussed here perform better than other known methods for searching repetitive, low entropy source, but perform poorly for most types of source.

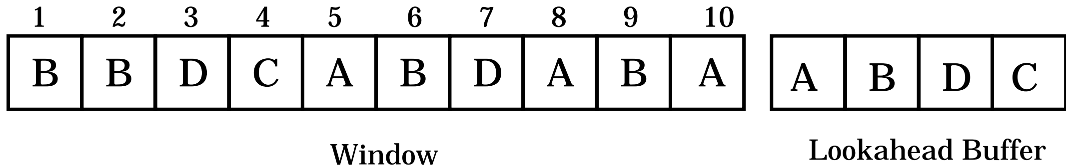

Figure 1: sample LZ77 window and lookahead buffer

2 The Boyer-Moore String Searching Algorithm

Boyer and Moore [BM77] designed a scanning algorithm for exact pattern matching in an un-indexed source. Worst case performance is guaranteed to be no worse than $O(M + N)$, where M is the size of the pattern and N is the size of the source. On average, the algorithm performs in *sub-linear* time, that is, less than N symbols are inspected. The algorithm maintains a data structure for identifying occurrences in the pattern of each symbol in the alphabet (δ_1) and a second data structure for identifying self-repetition within the pattern (δ_2). Conceptually, the pattern is overlayed onto the source text, comparisons are performed from right to left, and the pattern is shifted forward (to the right) along the source when comparisons fail. The number of positions that the pattern shifts is determined by referencing δ_1 and δ_2.

For example, Figure 2 shows a pattern, "ADBADABA", to be matched against the given source. The pattern is of length 8. To start, the pattern is overlayed along the 8 leftmost symbols, and comparisons begin at the *eighth* symbol ("A"). The eighth and seventh symbols match, but comparison fails at the sixth symbol. The algorithm must determine the maximum number of positions that the pattern can be shifted without missing a potential match.

δ_1 is an array of offsets which indicate the position, counting from the right, of the rightmost occurrence of every symbol; when comparisons fail for symbol c in the source, δ_1 prescribes the number of positions to shift the pattern to align the occurrence of c, if any, in the pattern with the same symbol in the source. In the example, comparisons failed at the sixth position when a "D" was read from the source. δ_1 indicates that a "D" is in the fourth position from the right in the pattern. The number of positions to jump is computed by subtracting the number of comparison

already made from the value of δ_1. In the example, $\delta_1('D') = 4$ and three comparisons have been made, so the pattern is shifted one position.

δ_2 is an array of M offsets that indicate the number of positions, j, to jump given a failure at position i. δ_2 is calculated by searching the pattern to identify repetitions within the pattern. More specifically, strings of one or more symbols in the pattern that are identical to the string beginning at the right side of the pattern are identified. In the sample pattern, the string "BA" beginning at the third position matches the final two symbols. On the other hand, the string "AD" occurs twice, but is of no interest because an occurrence of the pattern does not begin at the right side of the pattern. According to δ_2, the pattern may be shifted five positions to align the occurrence of "BA" in the source code with the "BA" in the pattern.

The Boyer-Moore algorithm selects the larger of the two values, δ_1 or δ_2, and shifts the pattern that number of positions. At this point comparisons resume again from the rightmost symbol in the pattern. In this way, large sections of text may be ignored. Baase[Baa88] gives an excellent discussion of the Boyer-Moore algorithm and provides source code for generating δ_1 and δ_2. Note that δ_1 and δ_2 can be generated in $O(M)$ time.

Although the Boyer-Moore algorithm was designed for exact pattern matching it can be modified for longest matching. Given the example in Figure 2, the first two *leftmost* symbols agree, and this may be the longest existing match in the source, but the match is never identified. The solution is to search the source from right to left and compare symbols beginning at the leftmost symbol of the pattern. Figure 3 shows the reverse Boyer-Moore scenario where the pattern is now "ABAD". Symbol comparisons are performed from left to right; when a comparison fails, the algorithm checks if the current (partial) match is larger than any found before.

The algorithm must also be modified to ensure that no partial matches are skipped. In Figure 3, δ_1 would dictate that the pattern should be shifted three positions to the left to align the "D" in the source with the "D" in the fourth position of the pattern. The partial match "AB" in the source text would be missed altogether. The solution is to modify the jump algorithm such that the *minimum* of the current maximum partial match length and the maximum of δ_1 and δ_2 is chosen as the jump offset, i.e. $MIN(curr_match_len, MAX(\delta_1, \delta_2))$. In the example, if no partial matches have been found, then the pattern shifts only one position; if a pattern of, say, length six has already been found, then the pattern shifts

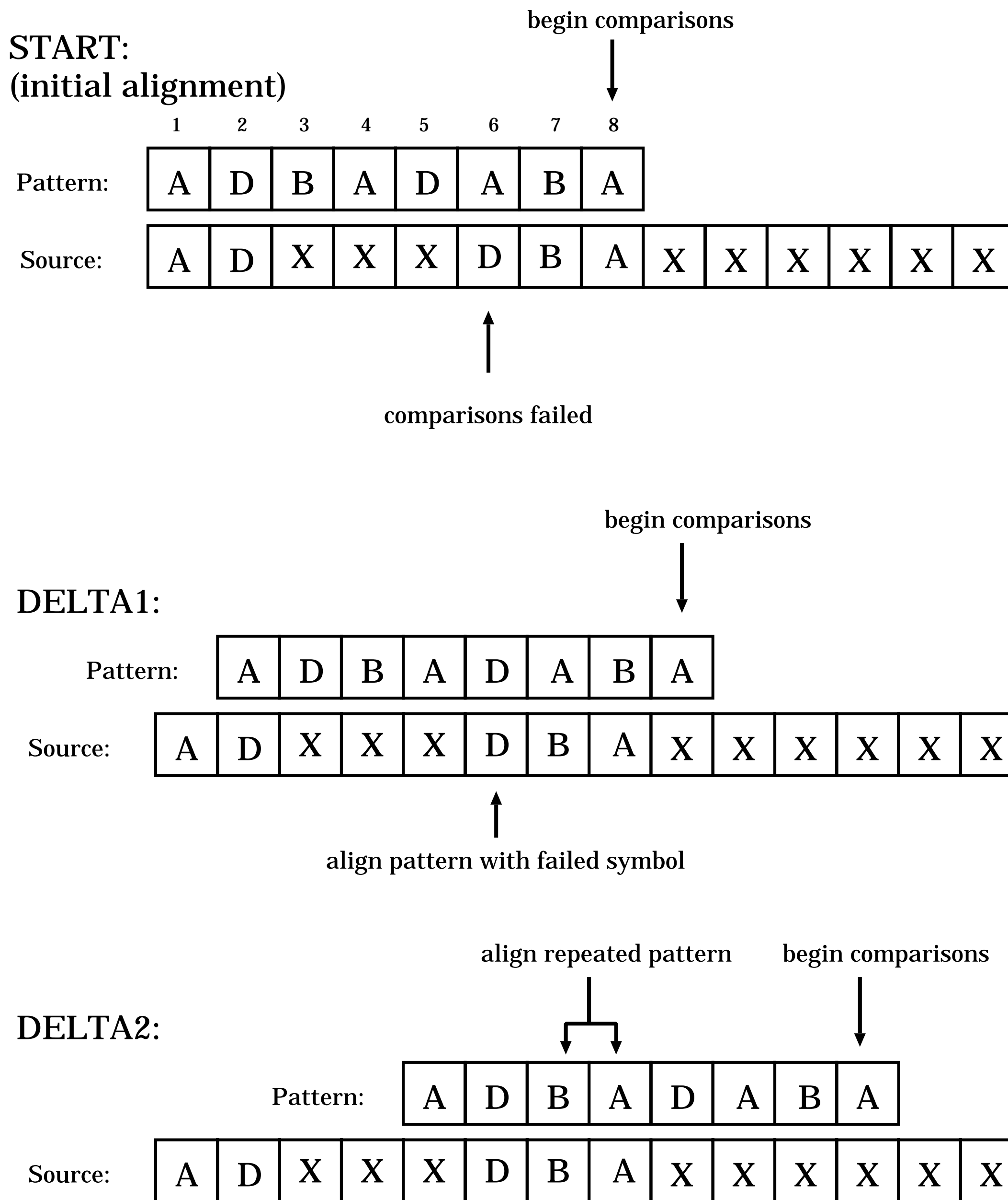

Figure 2: The Boyer-Moore algorithm

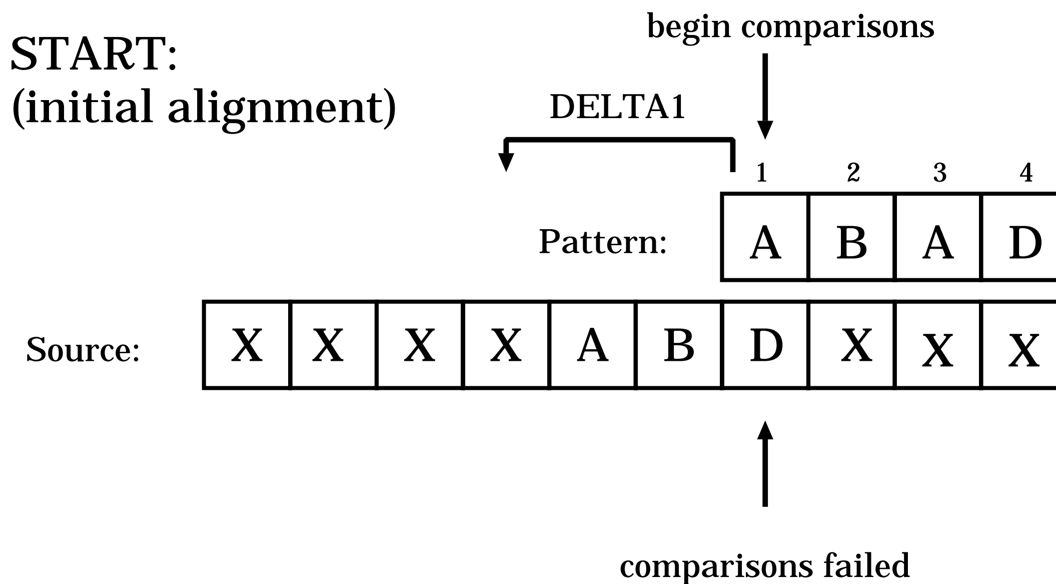

Figure 3: Reverse Boyer-Moore for finding largest prefix matches

three positions.

On average, the modified Boyer-Moore will perform slower than the original exact match algorithm because smaller jumps are often made to ensure that no partial matches are skipped. Nevertheless, worst case performance remains at $O(N + M)$ because there is still at most one comparison per symbol in the source and the time to construct δ_1 and δ_2 remains the same.

3 Statistical List

A simple indexing data structure called List1 is described in [BK] that maintains linked lists of the position of each occurrence of every symbol in the alphabet. For some pattern, the position of each occurrence in the source of the first symbol in the pattern is inspected as a potential match. The data structure is designed such that insertions and deletions are performed in $O(1)$ time, while searching is performed in sublinear time on average, since usually only a fraction of the N substrings in the window are compared. Only $O(q + N)$ memory is used, where q is the size of the alphabet, since there are q pointers to the first occurrence of each symbol, and exactly N pointers in the linked lists for the N symbols in the window.

Given $p(c)$ as the probability of the symbol c occurring in the source text, then for some pattern, k, $O(p(k_1) \cdot N)$ different string comparisons are performed, where k_1 is the first symbol of k. If a symbol is unlikely, then little time will be spent searching for possible matches, whereas a highly likely symbol will require many string comparisons.

Sedgewick [Sed83] describes a modification to the *exact match* simple linear scanning algorithm that searches for the least likely symbol in the pattern. A static table of probabilities for English text is used.

Sedgewick's technique of searching for less likely symbols can be combined with the linked list searching method above to locate the longest match. If some symbol in the pattern, k_i, is unlikely, then little time will be lost searching for occurrences of k_i, but if i is large, then it is likely that the length of the largest match in the source will be less than i. Generally, searches for less likely symbols that are within the size of the expected match length are favored. The technique presented here involves weighing each position in the pattern, favoring less likely symbols but taking into account the probability that the symbol position will be in the largest match.

Two data structures are required: the first holds the probability of each symbol in the alphabet, $p_A(c)$, and the second holds the cumulative probability of match lengths, $p_B(i)$, that is, the probability of a match of length i or greater. Before a search is made, the pattern is pre-processed, and for each position i and symbol k_i the function $f(i) = \frac{p_B(i)}{p_A(k_i)}$ is calculated. If $f(i)$ is greater than all $f(j), j < i$, then position i is placed in a LIFO queue as a search candidate.

Given a pattern of length m and queue operations $push(x)$ and $top()$, where $push(x)$ inserts x in the queue and $top()$ examines the contents of the top element in the queue without removing it, then the preprocessing algorithm can be described as:

```
push(1);
for i = 2 to m do
    if f(i) > f(top()) then push(x);
```

Figure 4 shows typical cumulative match length probabilities for some different sources. For geographic data, matches of size 4 or greater are unlikely, and a search for a symbol in the the fourth position or higher is unlikely to yield a pattern match. For the picture data, matches of size 4

or greater are much more likely. Therefore, a search for a symbol in the fourth position for picture data would be given a greater weight than the same positioned symbol in geographic data.

The search algorithm then performs a search for each symbol, k_i, for every i in the queue. If all searches for a match containing the symbol k_i (at position i) fails, then it is known that no match exists in the source that is of length i or greater. Therefore, subsequent searches need only perform comparisons of the symbols in the pattern from position 1 to $i-1$.

Given string comparison function $match(s_1, s_2)$, where $match$ returns true if strings s_1 and s_2 are identical, queue functions *pop*() and $empty()$, where *pop*() removes and returns the top element from the queue and $empty()$ returns true if there are no elements remaining in the queue, and list operations $pos(c)$, $next(c)$, and $done(c)$, where $pos(c)$ returns the position in the source of an occurrence of symbol c, $next(c)$ advances the pointer in the linked list to the next occurrence of the symbol c, and $done(c)$ returns true if there are no further occurrences of c in the source, then the search algorithm can be described as:

```
max := M;
while not empty() do
begin
   i := pop; c := PAT[i];
   while not done(c) do
      if match(PAT[1...max], SOURCE[pos(c)−i+1...pos(c)−i+max])
         then return(pos(c)−i+1,max)
         else next(c);
   max := x−1;
end;
```

The above function returns the position and length of the longest match in the source.

4 Experimental Results

Figure 5 shows the average time required to code each symbol using five different search algorithms for nine different sources[1]. Binary Tree and

[1] "book1" is a sample English text, "pic" is a binary picture file, "geo" is geographical data, "paper1" is a research paper, "progc", "progl", and "progp" are "C", lisp,

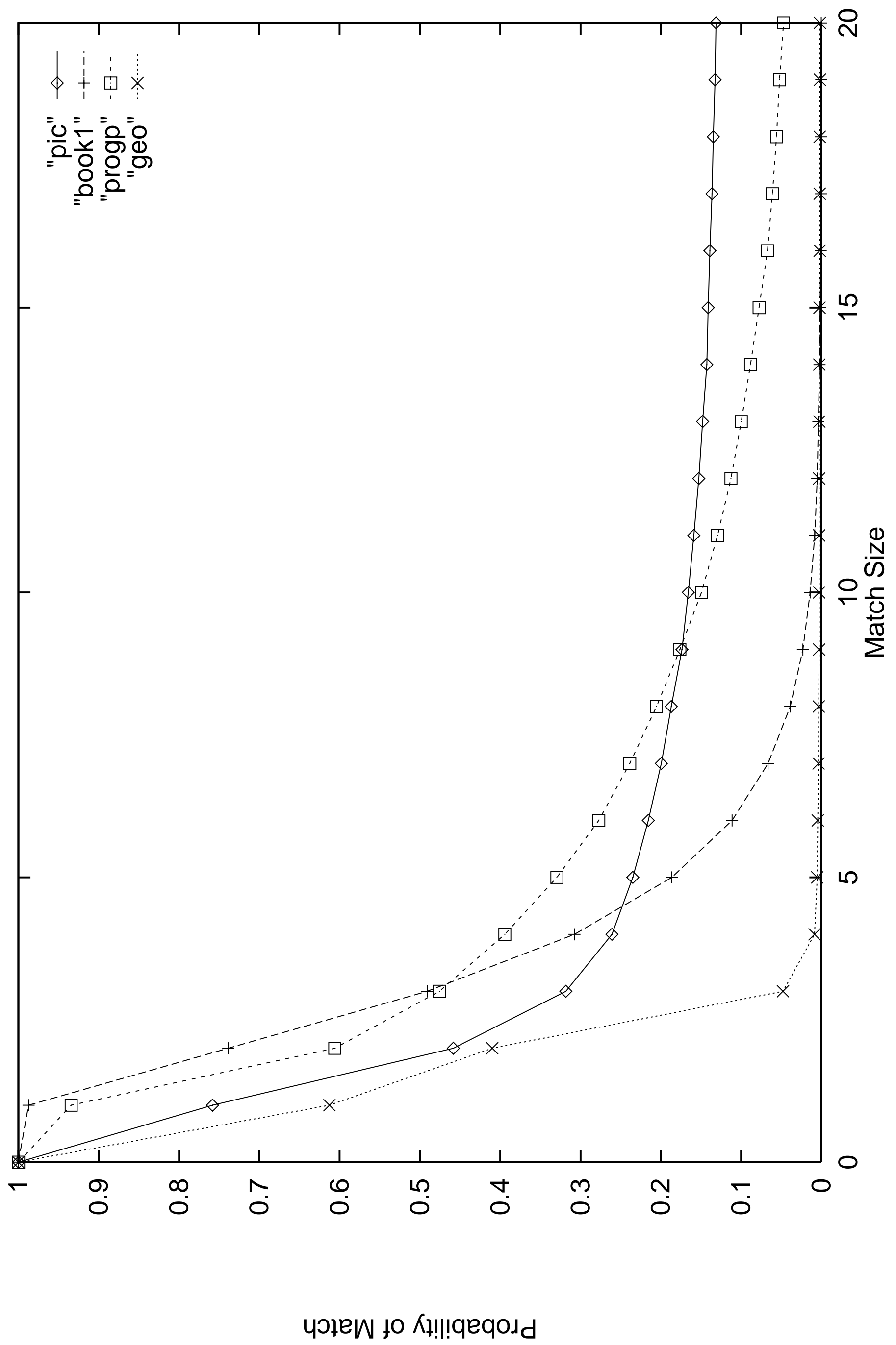

Figure 4: Match length probabilities for four different sources

Figure 5: Average Search Times

List2 are presented in [BK] as the two most favorable search strategies. The experiments were run using a maximum match size of 64 and window size of 8192, sufficient to give good compression for all sources. Of significant interest here is the good performance of both Boyer-Moore and Statistical on the "pic" file, while the other methods demonstrated very poor behaviour for the "pic" file. For most texts, however, the Boyer-Moore and Statistical algorithms perform poorly. Although Statistical is generally slow, it may offer acceptable performance, on average, since it has no severe worst case performance.

Figure 6 shows the number of comparisons made per search by each of the five methods. Note that Statistical consistently performs fewer symbol comparisons than List1, but usually performs slower than List1, which suggests that the overhead involved in choosing which symbol to search for, i.e. calculating $f(i)$, did not pay off in improved search time.

---

and Pascal programs, respectively, and "trans" is a terminal transaction.

Figure 6: Average Number of Symbol Comparisons

5 Conclusion

The Boyer-Moore exact match scanning algorithm is a simple and fast technique for locating patterns in highly repetitive sources, and is particular good considering its low memory requirements. The algorithm can be modified to locate the longest match in a source while retaining the same performance characteristics.

The Statistical List performs better than any known searching algorithm for the binary picture file and performs fewer symbol comparisons than List1 for most sources. The overhead required in pre-processing the search pattern, however, results in relatively poor speed performance for most sources. Because the Statistical List adapts its search strategy to the symbol probabilities and match length probabilities of the source, then severe worst case performance, as experienced by the Binary Tree, List1, and List2, is unlikely. The algorithm is appropriate for use in searching pictures and other highly repetitive text or as part of a general purpose searching utility for all types of sources when worst case execution time should be only slightly worse than the average case.

References

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1988.

[BK] T. C. Bell and D. C. Kulp. Longest match string searching for Ziv-Lempel compression. *Software - Practice and Experience.* (to appear).

[BM77] R. S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm ACM*, 20(10):762–772, 1977.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans Information Theory*, 23(3):337–343, 1977.