The Efficient Implementation of Common Lisp's `SEARCH` Function on Bit-vectors

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

August, 1989; revised April, 1991

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

Abstract

The efficient implementation of Common Lisp's `SEARCH` function specialized to bit-vectors is discussed. With its non-word-aligned search patterns and its small 2-element alphabet, a bit-vector `SEARCH` can often be the most inefficient of all `SEARCH`'es. Techniques, some of which we believe are novel, are discussed for overcoming these problems and achieving excellent performance on standard computer hardware. To reduce the amortized pattern pre-processing costs, we use four distinct search algorithms for pattern lengths of 1, 2 to `w`, `w`+1 to 2`w`-2, and larger, where `w` is the algorithm byte size.

Fast searching of a bit-string for an exact match of a shorter pattern bit-string can be required for some "bit-stuffing" communication protocols, and for the 1-dimensional portion of a 2-dimensional search within a binary image, such as those found on a bit-mapped display or transmitted by facsimile. Some of the techniques discussed are also relevant for searching with other small alphabets, e.g., the 2-bit ATGC alphabet used in DNA databases.

Introduction

This paper is a companion to [Baker90] which discusses the efficient implementation of the Common Lisp "sequence functions" [Steele90,s.14] on bit-vectors. The implementation of the function `SEARCH` [Steele90,s.14.4] was left out of that paper, because the techniques involved in speeding up the other sequence functions are mostly ineffective for speeding up `SEARCH`.

SEARCH

`SEARCH` (with the default `:test #'eql`) of two bit-strings is the hardest sequence function of bit-vectors of all to efficiently implement. It is hard because an efficient implementation of search is quite difficult even for character strings ([Boyer77], [Knuth77], [Davies86]). It is doubly difficult because we would like it to be of about the same efficiency as a character string search, in terms of the number of bits processed per second, even though the pattern may not be an integral number of bytes long, and even though an non-byte-aligned instance of the pattern may occur in the string. Achieving this level of efficiency is a tall order.

The most naive subsequence *exact match* search of a *pattern* of length `m` in a *string* of length `n` requires O(`m`•`n`) operations in the worst case, and this worst case unfortunately comes up quite often in bit-vectors. Linear O(`m`+`n`) algorithms are known, however, such as Knuth-Morris-Pratt ("KMP") [Knuth77], which scans the pattern from left-to-right and Boyer-Moore ("BM") [Boyer77] [Galil79], which scans the pattern from right-to-left (the string itself is always searched from left-to-right, because we desire the left-most match of the pattern within the string).

These algorithms are based on the fact that matching at any position is highly unlikely, and therefore the algorithm should be highly tuned for the quick rejection of mismatches, and for the skipping over without even trying of certain positions which cannot possibly match.

On average, BM is several times the speed of KMP on character strings, because it can skip distances up to the length of the pattern, while KMP must attempt a portion of a match for every character in the string. Therefore, BM is the logical algorithm to start from when attempting to build a fast searcher for bit-vectors.

Unfortunately, Boyer and Moore's own data show that the binary alphabet (i.e., bit-vectors) are the worst case for their algorithm, since one of their most powerful heuristics for skipping involves the likely absence of certain characters from the pattern. This heuristic fails for bit-vectors, however, and the BM algorithm on bit-vectors degenerates to a slightly more efficient version of KMP, because it starts with the rightmost part of the pattern instead of the leftmost.

Furthermore, none of the three algorithms—naive, KMP, BM—show any obvious `w`-fold speedup through the use of byte-wide parallelism (the `EQUAL` or `MISMATCH` used within the naive algorithm is speeded up, but only for bit-vectors which are almost equal, and we have seen that all but one of the `EQUAL`'s or `MISMATCH`'s will fail, and on the average will fail within a few bits—long before the speedup can be beneficial).

Nevertheless, we pursued the possibility of a faster bit-vector search because it was an interesting algorithmic question as to whether byte-parallelism could help on this problem.

Before considering a winning strategy, we first consider a losing strategy. One possibility would be to take the pattern, and position it on all `w` possible byte boundaries, and utilize a character-oriented search strategy on each of these `w` possibilities, and return the one with the smallest index. The time required is proportional to the byte-size `w` times `m+n`. While we are now doing byte-size chunks during the match, we now have to do byte-size times as many matches, so we have made no improvement. Furthermore, our worse case is longer, since we must now examine the whole string, whereas in the straightforward bit-oriented implementation, we can stop as soon as the first match is found.

We would like to preserve the speed of the BM "fast" case, wherein characters are looked up in a table, and the increment of the index to the next character to be examined is found in that table. In the ideal case, we would have a byte-indexed table and be able to skip through the string in byte-sized chunks. The major problem with this scenario is that byte boundaries get in the way.

An additional complication arises when trying to match byte-size elements when the underlying pattern is not located on a byte boundary. We must now ignore some of the bits in the process of trying to perform the match—i.e., we may now have some "don't care" items at the beginning and the end of the pattern. While "don't care" elements in the pattern can be handled—Pratt shows one way in [Pratt]—they add a great deal of complexity to the process for no gain in speed.

While the idea of using character-based processing for bit-strings doesn't work in its simple form, it provides the germ of an idea that *does* work. During the "fast" portion of the BM string searcher, we do a table lookup on every string character examined, and based on the results of this table lookup, we can either skip forward, or start matching. The key to the speed of our algorithm is to use this same idea, but our table allows for the possibility of matches at any bit location within the byte.

`SEARCH` for Consecutive 0's

Before tackling the general search problem, we will first "warm up" on a search for a consecutive pattern of 0's of length `m` in the string of length `n`. Let us first consider the case where `m` is considerably larger than the byte size `w`, so that we know that a consecutive sequence of 0's of length `m` *must* contain at least one aligned byte of all zeros. Since we will be using the Boyer-Moore strategy of matching from the right-hand end of the pattern, our search for a zero byte will commence at approximately `m` bits into the string. There are then two cases. If the test byte is zero, then we will commence searching backwards for the first 1, so that we can determine if there are `m` 0's in this region. If the test byte is non-zero, then we will advance the pointer by an amount approximately equal to `m`, because a non-zero byte implies at least one 1, and this would imply that the earliest we could possibly end a string of consecutive 0's of length `m` would be approximately `m` bits further down the string. Thus, we will have achieved essentially the Boyer-Moore speedup of examining only about `w/m` of the bits in the string when there are no zero bytes.

We must now be more precise about our algorithm. Let `m` be the length of the pattern (which in this case will be `m` consecutive 0's), `n` be the length of the string, and `w` be the byte size. Let `i` be the bit-number within the string of the proposed start of the pattern; clearly `i` starts out at zero. At any point in the algorithm, we will be checking to see whether the `m`-bit pattern matches the string in bit-positions `i`:`i+m` (we number subsequences in the same manner as Common Lisp's `subseq` function). Since we will be checking the right-most part of the pattern first (in typical Boyer-Moore fashion), and since we want to check (aligned) full bytes first, because they are more efficiently checked, we first check the *last* full byte against the pattern starting from `i`. The last full byte of the pattern in this position is the subsequence `j=floor((i+m)/w)*w-w`:`j+w`. If we find a non-zero byte at this location (the most likely case), then we cannot possibly have `m` consecutive zeros starting from `i` because all bit locations in this byte are within `m` of `i`, so if we find a non-zero bit, then we must abandon our attempt to find the pattern at location `i`. Now we must increment `i` to attempt another match at a new location. What is the maximum distance by which we can increment `i` without missing a possible match for the pattern? Since the pattern is all zeros, and since we have already shown that there is no possibility of a match at `i`, we have also eliminated any possibilities of starting a match anywhere between `i` and the full test byte just examined. Therefore, the earliest place a pattern of `m` consecutive 0's could possibly start is right after the last 1 in the test byte just examined. We therefore update `i` to this location, and attempt another match, as above.

If we skip through the entire string in this fashion (e.g., if no large strings of 0's exist in the string), then we will have examined only about `n/(m-w)` bytes, with only a minimal amount of work per byte examined, for a full speedup over a bit-oriented version of Boyer-Moore of a factor of about `w`.

We must also handle the case where the test byte examined during the "fast" loop is zero, however. In this case, we must do some more matching to see whether the pattern will match at this location. We first check the partial byte at the end of the pattern `j=floor((i+m)/w)*w`:`i+m`. If this partial byte is non-zero, then we determine the rightmost 1 found within that partial byte, update `i` to that position, and continue with the fast loop. If this partial byte is zero, we then embark upon a backward match of the string with the pattern starting from the predecessor of the full byte that we just checked. If this match succeeds, we have located the pattern within the string. If, on the other hand, we find a 1, we will have found the rightmost 1 within the region of the attempted match, and that becomes the new `i` for starting the next iteration of our "fast" loop.

```
/* Search for first position of m consecutive 0's within a byte-aligned string. */[1]
Initialize: i:=0;                                  /* Work from beginning of string */
Fast:       j:=floor((i+m)/w)*w-w;                 /* j:=((i+m) logand -w)-w when w=2^k */
            byte:=getbyte(string,j);               /* Test full byte from string at bit pos. j. */
            if byte=0 then goto Slow;
            i:=position1-from-end[2](byte)+j;      /* Simple table lookup */
            goto Fast;
Slow:       j:=position1-from-end[3](string,i,i+m); /* Check whole pattern. */
            if j then (i:=j; goto Fast);           /* Start searching again at rightmost 1. */
            return i;
```

---

[1] We apologize for non-Lisp pseudocode in a paper about Lisp.

[2] This is a highly specialized version of `POSITION` for a single word discussed in [Baker90]. We violate slightly the Boyer-Moore "fast" heuristic by setting `i` to be one past the last 1 in the test byte; we should instead set `i:=j+1` and immediately go back to "fast", since a nonzero byte at the right-hand end of the pattern skips many bytes instead of a few bits. We believe our approach is faster for shorter patterns, however.

[3] This is Common Lisp `POSITION` specialized for finding 1 in a bit-vector searching from the end [Baker90].

`SEARCH` for Arbitrary Patterns Large Enough to Always Contain a Full Byte

Now that we have solved the problem of searching for `m` consecutive 0's, we now take up the harder problem of finding an arbitrary pattern of `m` bits within a string of length `n`.

We still want to test a *full* byte within the fast loop, but since the pattern is no longer all zeros, we have more work to do. What is almost as fast as a zero test is a table lookup. We need to quickly determine whether the full byte occurs anywhere within the pattern—regardless of byte boundaries—and if it occurs, what its rightmost position is.

We do this by constructing a table of 2^w bytes which are indexed by `w`-bit bytes. We initialize all table entries to a large number (how large we will later see), and then move a window of size `w` through the pattern from beginning to end. At each window position, we enter the position number as the value of the table entry whose key is the bit sequence within the window. When we are done, we will have a table which indicates for every substring of size `w` within the pattern the bit index of the rightmost occurrence of that `w`-bit substring within the pattern.

Now within our fast loop we compute the position of the last (aligned) full byte in the current pattern position, and use this test byte as an index to our table. There are four possible cases:

1) the test byte does not occur within the pattern anywhere (table entry is "large");
2) the test byte occurs at this position in the pattern;
3) the test byte occurs to the left of this position in the pattern;
4) the test byte occurs to the right of this position in the pattern.

If the test byte does not occur anywhere within the pattern, then we may reposition our pattern to start at the second bit (bit#1) within the test byte and start the fast loop again.

If the test byte occurs in the pattern at this position, then we enter the slow loop and perform a more laborious match.

If the test byte occurs in the pattern, but to the left (earlier in the pattern), then this value tells us how far to shift the pattern to the right to align the bytes before proceeding with the slow loop.

If the byte occurs in the pattern, but to the right (later in the pattern), then we should extract the actual last full (unaligned) byte, update `j`, and use this unaligned byte as our test byte. If this new test byte does not occur within the pattern, then we can set `i` to be `i+m`.[4]

```
Initialize:  For j:=0 upto 2^w do table(j):=m;
             For j:=0 upto m-w do table(getbyte(pattern,j)):=j;     /* Unaligned getbyte's. */
             Build skip table as in normal Boyer-Moore algorithm.
             i:=0;
Fast:        j:=floor((i+m)/w)*w-w;
             k:=table(getbyte(string,j));                           /* Aligned getbyte. */
             if i+k<j then (i:=j-k; goto Fast);
             if i+k=j then goto Slow;
             if i+k>j then (j:=i+m-w; k:=table(getbyte(string,j)));  /* Unaligned getbyte. */
             if i+k<j then (i:=j-k; goto Fast);
             if i+k>j  then (i:=j+w; goto Fast);
Slow:        k:=mismatch-from-end(pattern,string,i);
             if k then (i:=i+skip(k); goto Fast);                   /* Traditional BM skip. */
             return i;
```

[4] We should also investigate the use of the "CanEnd" and "CanStart" tables developed in the next section to speed this case.

`SEARCH` for Short Patterns

Our previous algorithms dealt with patterns which were guaranteed to contain at least one full byte. Patterns of at least 2`w`-1 bits, where `w` is the byte-size, always contain one aligned full byte, no matter what the bit alignment of the pattern is. When `m` is less than 2`w`-1 (i.e., 15 bits when `w`=8), however, we must use a completely different algorithm.

Efficient searching of short patterns is a real problem, since we can no longer skip great distances in the string as a result of a mismatch. Furthermore, any fixed overhead (e.g., table-building) becomes relatively more expensive when amortized over a short pattern length; we will ignore this overhead for the moment—i.e., we consider the case of very long strings.

Efficient searching of the shortest patterns—of length 1—is handled by the sequence function `position` already described in [Baker90]. Patterns of lengths 2 to `w` occupy one or two bytes, while patterns of lengths `w`+1 to 2`w`-2 occupy exactly two bytes.

Patterns of length `w`+1 to 2`w`-2

We first tackle patterns of lengths `w`+1 to 2`w`-2, which extend over two bytes. The first byte contains 1 to `w` bits, right-aligned, while the next byte contains 1 to `w` bits, left-aligned. We then set up two byte-indexed tables "CanStart" and "CanEnd". We sequentially scan the string bytes looking for a byte which "CanStart" the pattern. If such a byte is found, we then look up the next byte in the "CanEnd" table. If one byte "CanStart" and the next byte "CanEnd", then we have a good potential for a match of the pattern within the two-byte subsequence.

But we can do better than this. If the "CanStart" table entry is a bit-vector whose "on" bits indicate the leftmost position of a succeeding match; i.e., if bit #'s 0,3, and 4 are all on, then the pattern can start in those bit positions within the index byte. Similarly, the "CanEnd" table entry can also be a bit-vector whose on bits indicate the leftmost position of a successful ending. Thus, when a non-zero CanStart:CanEnd sequence is found, we need only shift the CanEnd table entry left by `m-w-1` bits and logically "and" the two bit-vectors together. If the "and" is non-zero, then the pattern definitely occurs within the two-byte sequence, and the leftmost occurrence is indicated by the first "on" bit within this logical "and".

Unfortunately, this scheme requires two 2^{w}-byte tables, whose building can take a significant amount of time. If the pattern is known in advance, then these tables can be precomputed at compile time, or if the string is very long, then the table-building time becomes insignificant. At a cost of 2^{w+1} 2^{w}-byte tables, however, we can precompute all of the tables we will need, and quickly choose the correct one when the pattern becomes known; if `w`=8, then this will cost 128K 8-bit bytes for table storage. By doing a bit more computation during search time, we can save half of this room. We notice that the entry for the CanEnd table is identical to the reversed entry in the CanStart table for the reversed pattern, so we can utilize a (different) CanStart table in place of our CanEnd table by reversing the bits in the byte before using it to index the table and then reversing the entry. Thus, a single 2^{w}-byte byte-reversing table (already required by the `REVERSE` sequence function [Baker90]) can eliminate the need for a separate set of precomputed CanEnd tables, and we therefore need only about 64K 8-bit bytes for table storage.

Thus, to search for a `m`-bit pattern in a `n`-bit string, where `w`+1≤`m`≤2`w`-2, we find the CanStart table corresponding to the first `w` bits of the pattern, and the CanEnd table corresponding to the last `w` bits of the pattern, reversed. We then sequentially scan the string looking for a byte which produces a non-zero value when indexed into CanStart. When such a byte is found, we reverse the bits in the next byte and index the CanEnd table, reverse, shift left by `m-w-1` bits, and logically "and" the entry with the CanStart entry. If this byte is non-zero, then a match has occurred, and the left-most bit indicates the location of the match.

```
/*  Search for a pattern of length m, where w+1≤m≤2w-2.  */
Initialize:   i:=0;
              Select CanStart/CanEnd tables based on initial/final w bits of pattern.
Fast:         startmask:=CanStart(getbyte(string,i));[5]
              i:=i+w;
              if startmask=0 then goto Fast;
Slow:         endmask:=reverse(CanEnd(reverse(getbyte(string,i))));
              posmask:=startmask&(endmask<<m-w-1);
              if posmask=0 then goto Fast;
              return find1(posmask)+i-w;
```

Patterns of length 2 to w

The handling of length m in the range $2 \le m \le w$ requires a bit more complexity, because the pattern may be positioned in a byte in such a way that we have "don't care" bits at *both* ends of the byte. We can again use a CanStart table to scan the string, and when the table entry is non-zero, a possible match is indicated. Before accessing another byte from the string, however, we first "and" the table entry with a left-aligned mask of w-m+1 bits, and if the result is non-zero, then we have a confirmed match within the single string byte. If not, we access the next string byte and compute its CanEnd entry and shift and "and" as before.

We cannot use the previous CanStart tables from above for these lengths, but require a different set, thus requiring an additional 64K of 8-bit bytes when w=8.

```
/*  Search for a pattern of length m, where 2≤m≤w-1.  */
Initialize:   i:=0;
              Select CanStart/CanEnd tables based on the pattern.
              mMask is left-justified mask of w-m+1 1's.
Fast:         startmask:=CanStart(getbyte(string,i));
              i:=i+w;
              if startmask=0 then goto Fast;
Slow:         if (startmask&mMask)≠0 then return find1(startmask)+i-w;
              endmask:=reverse(CanEnd(reverse(getbyte(string,i))));
              posmask:=startmask&(endmask>>w-m+1);
              if posmask=0 then goto Fast;
              return find1(posmask)+i-w;
```

Conclusions

We have exhibited an algorithm for the fast searching of bit-strings for a bit-string pattern which approaches character-string search algorithms in efficiency, in terms of bits of the bit-string processed per second. This efficiency is gained by processing the bit-strings an aligned byte at a time and using table lookup, whenever possible. The algorithm breaks into four cases dependent upon the pattern length—1, 2 to w, w+1 to 2w-2, and larger patterns, where w is the byte-size in bits.

Our general bit-vector SEARCH algorithm requires 128K bytes of preprocessed CanStart tables when w=8, and could be speeded up a bit more through the use of a preprocessed CanEnd table, as well. Given the low cost of DRAM and disk memory, we believe that the cost of these tables is worth paying if any amount of bit-vector searching is envisioned. Alternatively, one could SEARCH a string for a while (perhaps 1000 bytes or so), and if the pattern is still not found, go into "turbo"

---

[5]In the finest Boyer-Moore tradition, we should check CanEnd of the next byte first, but we believe that the speed up would be minimal, given the short pattern length.

mode by pre-processing a CanEnd table at that point. Of course, one must pay the "turbo lag" cost of building the specialized CanEnd table.

The general desire of any efficient search algorithm to preprocess the pattern would argue for a slight change in the definition of the Common Lisp `SEARCH` function. Instead of the simple syntax `(search pattern string)`, Common Lisp should recognize the need for preprocessing by "currying" the `SEARCH` function like so: `(funcall (search pattern) string)`. In other words, the expression `(search pattern)` would itself return a search function which is specialized to search for the particular pattern. Currying the function in this way would enable simple "constant propagation" techniques to preprocess the pattern at compile time instead of at run time, so that the common case of searching for a constant pattern would be optimized.[6]

References

Allison, L, and Dix, T.I. "A Bit-String Longest-Common-Subsequence Algorithm". *Info. Proc. Let. 23* (1986), 305-310.

Baeza-Yates, Ricardo A. "Improved String Searching". *SW—Prac. & Exper. 19*,3 (March 1989), 257-271.

Bailey, T.A., and Dromey, R.G. "Fast String Searching by Finding Subkeys in Subtext". *Info. Proc. Let. 11*,3 (1980), 130-133.

Baker, Henry G. "Efficient Implementation of Bit-vector Operations in Common Lisp". *ACM Sigplan LISP Pointers 3*,2-4 (April-June 1990), 8-22.

Boyer, Robert, S., and Moore, J. Strother. "A Fast String Searching Algorithm". *CACM 20*,10 (Oct. 1977), 762-772.

Davies, G., and Bowsher, S. "Algorithms for Pattern matching". *SW—Practise & Exper. 16*,6 (June 1986), 575-601. (Rabin and Karp).

Galil, Zvi. "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm". *CACM 22*,9 (Sept. 1979), 505-508.

Knuth, D.E., Morris, Jr., J.H., and Pratt, V.B. "Fast Pattern Matching in Strings". *SIAM J. Comput. 6*,2 (1977), 323-350.

Kuck, David J. *The Structure of Computers and Computations, Vol. I.* John Wiley & Sons, NY, 1978.

Semba, Ichiro "An Efficient String Searching Algorithm". *J. Info. Proc. 8*,2 (1985)

Steele, G.L. *Common Lisp, the Language: Second Edition*. Digital Press, Bedford, MA, 1990.

Zhu, R.F., and Takaoka, T. "On improving the average case of the Boyer-Moore string matching algorithm". *J. Inf. Proc. 10*,3 (March 1987), 173-177.

[6] `SEARCH` is not the only Common Lisp function which should be so curried; `FORMAT` is also an excellent candidate for currying to allow for compiler preprocessing of its usually constant formatting string.

