 Version 1.1.

Notes on the MiscTBMK String Searching Routines

Tips on Using these Routines

You've got to search a 10MB memory stream for 30 character literal strings? No problem! You need to search a 256 character string for an occurrence of the word *and*? Well... Here are a few hints on how and when to use these routines.

1. The larger, the better. The algorithm exhibits two interesting properties that result in the same thing: the larger the text to search, the faster the apparent search rate; the

larger the pattern for which to search, the faster the real search rate. Both of these (counter-intuitive?) properties are a product of the fast skip loop [*see discussion below*]. What this means in quantitative terms is that the algorithm is best- suited to searching for patterns of length >= 5 (or so), and searching texts greater than 5 kilobytes (very roughly). Patterns and texts shorter than that do not take as great an advantage of the skip loop, and the overhead of the pre-processing ("compiling") on the pattern string begins to be noticeable and, as pattern and text size shrinks, this algorithm will be slower than the simple brute-force string searching easily imagined. I have no canonical data though; there are many variables and the best suggestion is to experiment.

2. Regular expressions. These routines will only search for an array of literal characters. There is no regular expression support. Note that the search is for a fixed-length block of characters, not just strings. There may be nul characters imbedded in the pattern, if desired.

3. Reading and seeking for searching. Clearly, only streams that are readable can be

searched. Unfortunately, the Boyer-Moore algorithm also requires buffering of the text. Thus, only streams that are seekable can be searched with the stream-searching routine (at present). Mach port streams and streams on pipes/sockets and FIFOs are not seekable. This applies particularly to the C library standard I/O descriptor *stdin*, which may be reading from a file, a terminal, a pipe, etc. So sometimes you may be able to search *stdin* with these routines, others not. See the UNIX manual page for *lseek(2)* for more information.

4. Big contiguous blocks of memory. The memory searching routine is about six times faster than the stream searching routine. A lot of work goes into maintaining the stream header (structure). Use the memory searching routine whenever you can. The practical suggestions that arise from this:

- For a memory stream, use *NXGetMemoryBuffer()* to get the stream's memory buffer, and pass this buffer to *Misc_TBMKsearch_memory()*. *NXOpenMemory()* and *NXMapFile()* create memory streams.
- If you want to search a file on disk, use *NXMapFile()* to get a stream on the file,

rather than opening the file and using *NXOpenFile()* on the descriptor. *NXMapFile()* will (for most cases) call *map_fd()* to have the file mapped into memory on demand. If you don't need a stream, use *open()* and *map_fd()* (or *mmap()* on non-NeXTs if available) yourself. Use *Misc_TBMKsearch_memory()* on the block of data that results.

- The stream returned by the Text class's *-stream* instance method is *not* a memory stream, nor would be a seekable stream opened on *stdin*. *Misc_TBMKsearch_stream()* must be used for these. Alternatively, it may be faster to read the material into another (memory) stream, and then get the memory buffer and search in it.

5. More is better. Finally, don't recompile a pattern if you don't have to. If the user is doing a search, they may want to do a "find next"-type search for their next search. You may not want to throw that compiled pattern structure away after a search operation. Remember, though, that the case sensitivity and direction of a search can only be specified when a pattern is compiled. You may find that the extra overhead of caching a

few compiled patterns outweighs any efficiency advantage.

Notes on the Implementation

The string searching algorithm implemented here has its foundations in the work of Boyer and Moore published over 15 years ago. A few enhancements to their algorithm have been made in that time, but it remains fundamentally the same. There are several fast string-searching algorithms, but many rely on knowledge of the structure text to be searched (for instance, character frequency). This algorithm was chosen as the basis for this implementation because it is fast, relatively simple, and general. It began with the fast version of the Boyer-Moore algorithm, presented by Hume and Sunday [Hum91] which they called the Tuned Boyer-Moore algorithm. I have taken this algorithm, then the fastest known general version of Boyer ands Moore's, generalized it to both forward and backward text searching and optional case insensitivity, and coded a version in C.

The result was quite fast, considering all the "extra" functionality loaded into the code (as compared with the algorithms that are designed and tested in the literature). Version 1.1 is even faster than
the original version, as more decisions have been pushed into the pre-processing step and the skip loop has been unrolled a bit.

It's in the skip loop that the code takes typically 70-90% of its time. The skip loop uses the skip table calculated during pattern "compilation". The skip table contains the shift that should be applied to the current text pointer to align to the next possible match, one value for each character; the shift is the distance from the last occurance of a particular character in the pattern to the end of the pattern. The comparison of the pattern against the text proceeds from right to left (the innovation of Boyer and Moore). Now for an example (forward search):

```
pattern: nation
         skip['n']=0, skip['o']=1, skip['i']=2, etc.; patlen=6; jump=5
```

```
text: Hath yoked a nation strong, trained up in arms.[2]
1     *****^                                          skip['y']=6
2           *****^                                    skip['a']=4
3               *****^                                skip['t']=3
4                  *****^                             skip['n']=0
5                        *****^                       skip['n']=0; jump=5
6                             *****^                  skip['r']=6
7                                   *****^            skip[' ']=6
8                                         *****^      skip[' ']=6
9                                               *****^ out of bounds, quit
text: Hath yoked a nation strong, trained up in arms.
```

The carat marks the "current pointer" in the text being searched, and the asterisks represent the other characters of the pattern for convenience. At line 1, we are about to begin the skip loop. The skip value for 'y' is consulted, and found to be 6; the skip is the length of the pattern for characters not in the pattern. The current pointer is moved 6 forward; since 'y' is not in the pattern, moving the pointer to anywhere where the 'y' would

have continued to line up with a character of the pattern is pointless.

Now, we look up the skip (shift) of 'a', four, and shift the pointer that much. This illustrates the "align to the next possible match" I spoke of earlier--by shifting the current pointer by four, the 'a' in the text and the (rightmost) 'a' in the pattern have been aligned, which is required if there is to be any hope of a match at all. Now back to skip table lookup, this time with an 't'. The pattern is shifted right three.

Now (line #4), the pointer is pointing to an 'n', which has skip value of zero. The value of zero is special, because in this right-to-left comparison thing we are doing, an 'n' indicates a possible pattern match. The algorithm falls out of the skip loop at this point and performs a character-by-character comparison of the pattern and the text. Sharp-eyed readers of the code will notice that the this comparison is left-to-right. Some research (and a little thought) indicates that there are often relationships between adjacent characters in patterns (consider the many common word suffixes like *ed*, *tion*, *es*, or any

word with 'q': *qu*, or common pairs of letters like *ck* and *th*) and that the "least relationship" is between the first and last character. So, since the character on the right has already matched, we look at the character on the other end of the pattern. Some experimentation I have done showed that typically 75-90% of the possible matches (where the algorithm has fallen out of the skip loop) mismatch on the first character test. To continue with the example, this is a match, which the algorithm finds, and then skips the length of the pattern forward (since the algorithm searches for non-overlapping matches).

Hmmm, another 'n'. The skip loop isn't getting much time in. We fall out of the skip loop again, but this time, we mismatch on the first comparison (with the space). At this point we know we matched with the rightmost character, so we shift the pointer so that the *second-to-the-rightmost* occurrence of the last character in the pattern is align with the character matched in the text. This is the same thing we did for the 'a' and 't' above (lines 2,3) (shift to next possible match) but with the second-to-the-rightmost occurrence, since the rightmost occurrence has had its chance, and failed. The value of *jump* has been

pre-computed in the pre-processing of the pattern.

The current pointer is now pointing to an 'r' in the text. Here we see why the skip loop is so wonderful. 'r' in the pattern? No, shift 6. ' ' in the pattern? No, shift 6. ' ' in the pattern? No, shift 6. Its like a pebble skipping across a lake; the algorithm only briefly "touches down on" the text before moving on. Once the pointer points beyond the end of the text, there aren't enough characters left to possibly match, so the algorithm quits.

If a reverse search is desired, the current pointer moves left, skipping looks at the leftmost character, and text-pattern comparison is from right-to-left; the mirror image of searching forward (which is the intuitive result, but requires some thinking to actually prove to onesefl that it can work). I've not said anything about case sensitive comparison in the algorithm, you may have noticed. The code seems to be comparing the skip values of the characters rather than the characters themselves. The reasoning behind this is left as an exercise to the reader. (Hint: write the code for the obvious comparison loop,

handling both search possibilities (case sensitive and case insensitive comparisons), pretending the value of *nocases* has been saved in the pattern structure by the pre-processing routine, then optimize. See near the end of this document for some discussion on this.)

While reading this example, you've probably thought once or twice, *What if...?* or *How about...?* or *Why not...?* Well, I invite you to experiment. If you come up with something interesting, please let me know. I've done quite a bit of it myself, and have decided that this particular coding of the algorithm is better or faster than other things I've tried. Here are two possible optimizations that are not-so-obvious that I haven't implemented; the reasons for that are the problems-at-the-end-of-the-chapter.

1. Ignore for a moment the first *if* statement in the skip loop and all those += assignments (the loop has been unrolled to reduce the overhead of the loop control; the overall speedup is 15-20% with this unrolling). The basic loop skips, then checks

for out-of-bounds and exists if so. If the text being searched is large, that bounds check is done an awful lot of times when there is no possibility of it being true (and in any case, of course, it is only true once). Can you think of a way to eliminate that *if* statement? There is a way: with sentinels. Suppose that the *patlen* characters after the end of the text were filled with the rightmost character of the pattern (which, recall, has a skip value of zero). When the skip loop runs off the end of the text-to-be-searched, it will run into this area of zero skips and fall out of the skip loop, where we can then put the bounds check. Since the skip loop iterates on average two to three times before exiting, and we've eliminated two comparisons per iteration, we can expect this to be a big win (another 15% $\pm$ 5% overall it turns out, with the loop unrolling that is also there). This is actually part of the algorithm presented in [Hum91]--sentinels are written after the end of the text before searching begins. Give at least two reasons why this is not a good idea. Are there ways of circumventing these problems?

2. The value of *jump*, the shift to move the pattern to the next alignment of the rightmost (in forward searching) pattern character is at least 1 and at most the length of the pattern. Another move-to-next-potential-match heuristic the algorithm could use after the text-pattern comparison has failed is to jump based on the skip value of the character *after* the current pointer--the character after the matched rightmost character (text[cp+1], if we pretend *cp* is an integer index into *text*). Rather than

cp += pattern->jump;

the statement could be

cp += pattern->skip[(cp+1)]+1;*

The values in the skip table are between 0 and the length of the pattern, inclusive, so this jump is also always at least 1 but may be 1 greater than the pattern length; potentially bigger jumps than the current jump heuristic. Try implementing this, and testing whether the algorithm is then faster or slower than the current one (you may wish to use *pattern->skip[*++cp];* on the righthand side of the assignment). If it is faster, why do you suppose that it wasn't used? If it is slower, why is it slower?

Would it be better to "get the best of both worlds" by using the maximum of the two values?

Discussion on case sensitivity in the text-pattern comparison loop: Did you split the loop into two loops, choosing one based on *if (pattern->nocases)*...? Note that this makes the same decision over and over again, a decision that is fixed for the duration of the algorithm. The "obvious" comparison loop can be coded as one loop, but the loop test involves then possibly two comparisons, two boolean operations, and a boolean variable (*pattern->nocases*). And again, there is a decision being made again and again that is constant for the entire algorithm. In a case insensitive search, the skip value for an uppercase letter is the same as its lower case equivalent. The code that being used uses the skip table as a lookup table much the same way that the *isupper()* and related macros use one, mapping characters into classes.

Discussion on skip loop sentinel optimization: Two possible reasons are that addresses "beyond" either end of the text may not be in the virtual memory space of the process, or that the memory may be read-only. There aren't any good ways around these problems; copying the text to other memory is just not practical unless the text length is quite small, and in this case the pattern pre-processing probably dominates the search time! In the theoretical literature, practical problems like this are often ignored, so Hume and Sunday can get away with this optimization.

Discussion on alternate jump heuristics: Actually, it is roughly a wash; neither heuristic is faster. The potentially greater jump of the proposed heuristic is negated by the extra memory access required. Taking the maximum of the two heuristics is definately worse: not only would both values have to be computed, but they then need to be compared to decide which is larger; a potentially greater jump of 1 cannot offset this. This is why the maximum of the two delta values of the original Boyer-Moore algorithm (for those of you familiar with it) is not used, nor is the second delta function; no speed advantage.

[Hum91] Hume, A., D.M. Sunday. *Fast String Searching*. Software--Practice and Experience. Vol. 21. No. 11. p. 1221-48. November 1991.

[2] Wm. Shakespeare. *Titus Andronicus*. 1.1.