Analysis of Fundamental Exact and Inexact Pattern Matching Algorithms

Jonathan Lee

ID# 05354211

BIOC 218

June 4, 2004

Introduction

The problem of pattern matching is encountered in numerous arenas, from internet searches to library catalogs. Pattern matching plays a special role in bioinformatics due to the wealth of information that can be gained from finding patterns in genomic sequences. However, the challenge to provide efficient and correct pattern matching continues to grow as the volume of genomic information swells. The following is an examination of a subset of the larger problem of pattern matching: exact and inexact pattern matching. Within these areas, we will concentrate on the more fundamental algorithms, those that give rise to other algorithms, and give a representative sample of the types of benefits offered and challenges encountered by current methods. In addition to descriptions of the algorithms, the methods will also be reviewed in the context of current technology and processor capabilities in an effort to identify possible means of improvement.

Exact Pattern Matching

Exact pattern matching involves finding all occurrences of a pattern *P* in a string *S*, where *S* is longer than *P*. The simplest form of pattern matching, exact pattern matching is still widely used in a variety of text searches from internet search engines to word processing. While higher processor speeds and other advances have reduced search response to negligible times, exact matching still remains a useful area of study and development for a number of reasons. First, as genomic information continues to grow, sequence searches will become increasingly taxing on search engines. Searching for patterns in sequences will invariably be more difficult than performing common internet searches because of fewer unique constraints (i.e. four nucleotides, or twenty amino acids), and the absence of spaces denoting words/patterns. Also, many of the searches performed on the internet are accelerated because many of the answers people seek are pre-computed, while those for genetic sequences are not. Secondly, the exact pattern match still remains an integral part of faster matching

algorithms, typically comprising the final part of a search. Lastly, an understanding of the classical methods of exact pattern matching lends itself to the development of new algorithms.

Naïve Method

The simplest method of exact matching is the naïve method. The premise is simple: the first letter of pattern *P* is lined up with the first letter of *S*, and the letters of the aligned region are compared until all of *P* is found to match the corresponding *S* region or a mismatched letter is found, in which case *P* is shifted one letter to the right and the process is repeated.

Example:
```
CATMOUSE
MOUSE
 MOUSE
  MOUSE
   MOUSE
```

While complete, the speed disadvantages to this method are clear. By moving one character at a time, the worst-case time required to compare the pattern to the entire sequence would be proportional to the length of *P* multiplied by the length of *S*.

By performing an initial preprocessing of the text, the speed of the main search can be increased appreciably (Gusfield, 1996). For example, to find the pattern "MOUSE" in the following string:

```
MOPMOUSEMOUNTAINMONKEY
```

one could first perform a search locating the M's in the string, and then only compare "MOUSE" to those locations.

Example:
```
MOPMOUSEMOUNTAINMONKEY
MOUSE
   MOUSE
      MOUSE
         MOUSE
```

Additionally, one could search for the prefix “MOU” during the preprocessing, and save additional time.

Example: MOPMOUSEMOUNTAINMONKEY

MOUSE

MOUSE

In a case of no matches found during preprocessing, time is saved because no further searching is needed, and thus the time saved is proportional to the relative sizes of the prefix and the original pattern. However, as might be likely in searches of DNA sequences, if the prefix is found often or if the useful size of the prefix is large compared to the pattern, the cumulative time of the preprocessing and main search may show only a negligible time savings.

Boyer-Moore

The Boyer-Moore algorithm (Boyer, Moore, 1977) provides marked improvement over the naïve method through the implementation of three key ideas. First, although the pattern *P* and string *S* are aligned on the left, the algorithm scans for matches from right to left. Secondly, when a mismatch is found between letter *p* in P and letter *s* in *S*, P is moved to the right so that the rightmost occurrence of *s* in P is aligned with *s*; if *s* does not occur in P, P is shifted one space to the right of *s* (“Bad Character Shift Rule”).

Example: CATDOGMOUSECAT

MOUSE	1. O-E mismatch
MOUSE	2. O in MOUSE shifted to O in DOG, O-E mismatch
MOUSE	3. O in MOUSE shifted to O in MOUSE

Lastly, the algorithm implements a third rule which is improved upon by Gusfield (1997). The modified rule (“Strong Good Suffix Rule”) states that if the suffix Q of pattern *P* matches a substring *R* in *S*, when a mismatch is found, P is shifted to the right so that Q’ is matched up with R, where Q’ is defined as the rightmost occurrence of Q in P with a different letter to the left. If *Q’* does not exist then the left end of *P* is shifted past the left end of *R* until another match is found, or until *P* is shifted past *R*.

Example: DOGMOUSECATBATBAT

CATBATB**AT** 1. Suffix AT matches, but C-B mismatch

CATBATBAT 2. Shift pattern *P* until suffix AT with different left letter is found (i.e. CAT)

The combination of the three characteristics of the Boyer-Moore algorithm makes the method simple yet powerful. Immediately, searching from right to left with the Bad Character Shift Rule, the potential to skip over a greater number of non-matching substrings is realized. However, without the Strong Good Suffix Rule, the algorithm would be vulnerable to short alphabets. For instance, in the previous example, the pattern would have only been shifted three spaces if the algorithm had not realized that positions 4-6 of P were the same as 7-9. Theoretically, Boyer-Moore would at worst run in linear time (i.e. proportional to the sum of the lengths of *P* and *S*), and would normally run sublinearly. However, the cumulative length of genomic information precludes the use of Boyer-Moore as the sole search method.

Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm seeks to improve the length of the pattern shift by utilizing information already gathered from searching a string (Knuth, Morris, Pratt, 1977). To accomplish this goal, the algorithm preprocesses the pattern and creates a finite state machine (or automaton). Typically, the information is portrayed in tabular form with “KNP Next” values assigned to each character in the pattern based on the number of spaces the machine moves the pattern if a mismatch is found. The algorithm then uses the finite state machine to process the string.

Example: KMP Next Table

```
G  C  A  G  A  G  A  G -
-1 0  0 -1  1 -1  1 -1 1 (KMP Next value)

GCATCGCAGAGAG
GCAGAGAG          1. Mismatch at position 4, shift by 3
                  (number of matches) minus -1(KMP Next
                  value of 'G' at position 4 of P) = 4
  GCAGAGAG        2. Mismatch at position 1, shift 0 - (-1) = 1
   GCAGAGAG       3. Pattern match, shift 8-1 = 7 to continue
```

(Courtesy: Lecroq, 2004)

Properly implemented, the Knuth-Morris-Pratt algorithm has the potential to look at each letter of the string only once. However, with the table in the previous example, one can see that some letters will be searched twice, namely those in the middle of mismatches, which have KMP Next values that shift the first letter of the pattern to the mismatched position.

Example: KMP Next Table

```
G  C  A  G  A  G  A  G -
-1 0  0 -1  1 -1  1 -1 1 (KMP Next value)

GTATACAGT
GCAGAGAG          1. Mismatch at position 2, shift by 1-0=1
 GCAGAGAG         2. Mismatch at position 1
```

As seen in the example above, after the mismatch at position 2, the pattern is only shifted over one character. As a result, the letter 'T' in position 2 of the string is run through the algorithm twice. Even though the character had already been looked at, the algorithm shifts the pattern into another mismatch at position 1. It may be possible to circumvent this issue by building another level into the KMP Next Table. The additional level would contain information pertaining to the character on the string which caused the mismatch.

Shift-And

The Shift-And (or Shift-Or) method is a semi-numerical algorithm developed by Yates and Gonnet (1992) based on the use of numerical comparisons rather than character comparison. For a given string and pattern, a two-dimensional array is created to store match/mismatch information for each character in the pattern compared to each character in the string. A match would then be scored as a 1, a mismatch, 0. For example if *P* = CAT and *S* = CART, an array *M* would have values of 1 in *M*(1,1), *M*(2,2), and *M*(3,4) and value of 0 in all others.

	C	**A**	**R**	**T**
C	1	0	0	0
A	0	1	0	0
T	0	0	0	1

The Shift-And method has an advantage over character comparison because bit comparisons are performed much more quickly by processors. However, because the size of the array is the length of *P* multiplied by the length of *S* +1, the method is only useful for relatively small strings. Highest efficiency would be achieved if the number of characters in the string is less than the number of bits in a computer word (Gusfield, 1997). Thus, if a typical computer word is 32 bits, a processor would be able to accommodate a string of 31 letters in a single operation, eliminating the need to repeatedly loop through the string. However, while 31 letters would be sufficient for an English word, even several times that capacity would only accommodate a small fraction of genomic sequences. Nevertheless, as a final processing step, Shift-And clearly has benefits in both exact and inexact matching.

It may also be useful to extend the idea of increasing the amount of string information stored in a computer word. A letter or punctuation mark is generally assigned 8 bits, which can accommodate 256 different combinations. For typing, these 256 combinations are necessary, given the 26 letters, capital letters,

numbers, punctuation marks, etc. However, for protein sequences, only 20 different combinations are needed, and as a result only 5 bits would need to be assigned to a single letter/amino acid in a protein sequence. For a DNA sequence, only 2 bits would need to be assigned to a single nucleotide. By not treating the sequence characters as keyboard letters, 1 ½- and 4- fold increases in information per computer word would be achieved for protein and DNA sequences respectively. However, in theory this acceleration would occur, but in practice, because processors tend to work with powers of 2, nucleotides would indeed be treated as 2 bits, but the 5 bits of the amino acid may end up being treated like an 8-bit character.

Karp-Rabin

The Karp-Rabin Random Fingerprint algorithm is a semi-numerical method which utilizes a hashing function to compare whether a substring of equal length to the pattern is a match (Karp, Rabin, 1987). The hashing function preprocesses the string by moving one letter at a time, assigning a value to substrings the length of the pattern. The function computes the hash value by performing an arithmetic calculation on the pre-assigned values of each letter in the substring. In the original Karp-Rabin algorithm, the values of each letter were randomly assigned prime numbers, which increased the probability that a match found was actually a match. Additionally, each step of the arithmetic operations performed ends in the modulus of a small prime integer. As a result, the values involved are small and easier to manage. After the hash function preprocessing, the algorithm performs the same hash function on the pattern, and then performs a left-to-right comparison of substring hash values. An example with arbitrary hash values follows.

Example: String = CABBATDAT, Pattern = BAT (Hash Value = 277)

```
CABATTDAT
CAB, Hash = 151
 ABA, Hash = 103
  BAT = 277 (Match Found)
   ATT, Hash = 129
```

A good hash function would only yield the same value if the compared strings were indeed a match. For the Karp-Rabin algorithm, with a pattern of 250 characters and a string of 4000 characters, the probability of producing a false positive is at most p=0.001 (Gusfield, 1997). An efficient function would also increase calculation speed by recognizing that with each move only one number is added and one number dropped.

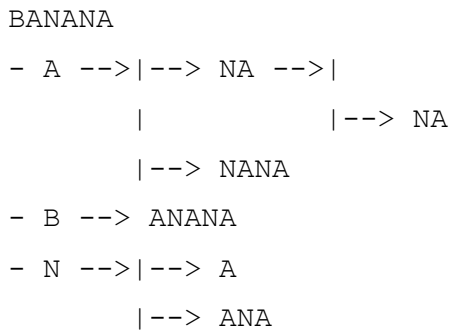
Because the Karp-Rabin looks at each character in the string twice (once as it enters the hash function and once as it exits), longer string searches are lengthy, especially during the preprocessing stage. Also, when a hash value match is found, the substring needs to confirmed, possibly adding appreciable time for long patterns which are repeated frequently in the string. Instead of randomly assigning prime numbers, because the alphabet size of proteins is fixed, it may be useful to assign numbers that are related to the frequency of the amino acid in nature. In vertebrates, for example, tryptophan accounts for only 1.3% of amino acids (Beals, 2004). Tryptophan could be assigned a relatively high value, then based on the values of the other amino acids, a large substring could be summed during the preprocessing step and it could be quickly determined whether tryptophan was present. In this way, depending on the amino acid composition of the pattern, the problem could be more quickly constrained as larger chunks of data could be dismissed, and areas with potential matching more quickly identified. The larger values assigned would not affect the hashing function because the modulus operation would return a small integer. However, the larger values may affect the probability of false positives.

Suffix Trees

The idea of the suffix tree was originally introduced by Weiner (1973), and was modified to create a more space-efficient algorithm by McCreight (1976). More recently, Ukkonen (1995) developed an “on-line” method, which requires much less memory than the other algorithms, but is equally fast. The main premise of the suffix tree is that a string can be broken down into as many suffixes as there

are characters, and this information can be stored in tree form. In addition, because the words are stacked alphabetically, suffixes with multiple locations and common prefixes can be readily located.

Example (Implicit suffix tree):



One of the primary benefits of a suffix tree is that after the tree is created, one can move between locations of a string quickly, regardless of actual distance in the string. However, this characteristic may also lead to slower processing in larger trees if the suffixes of interest are located in different areas of memory. In addition, longer trees require a much greater amount of space than the normal sequence. If every suffix is included in the tree, and there are no duplicate suffixes, for a string of length *M*, (*M*/2)*(*M*+1) characters are required. Suffix trees are also able to provide quick insight into repeated patterns and longest shared substrings. However, the construction of the tree does not benefit from dual processors because each letter must be addressed individually and in order. So suffix trees of appreciable length need to be stored and maintained to be useful, as building large trees on demand is inefficient.

Inexact Pattern Matching

While exact matching provides a fundamental basis for string searching in sequences, the nature of biological sequences requires that differences in sequences be accommodated and utilized. Similarities in the text of sequences often indicate similarities in function, structure, or homology. The challenge in inexact, or approximate, matching is allowing enough edits (insertions, deletions, and substitutions) to detect relevant patterns, but not so many as to make the

comparison irrelevant. With inexact matching, the concern shifts to alignment of sequences based on their similarity. The nature of the problems of sequence alignment and the numerous routes one could take to address them yield a variety of categories of alignment methods, ranging from pairs of sequences to large groups. However, of interest to this paper are the methods which are concerned with two-sequence comparisons. Many of the fundamental methods for this task can be found in the area of dynamic programming (Chan, 2004). In short, dynamic programming is the optimization of a solution through the solving of a series of problems within the larger problem, instead of stopping to recompute. Dynamic programming is well suited for use in biological sequences because it works only when the problem has an optimal substructure, i.e. the smaller problems can be solved in succession to solve the main problem. Due to the higher complexity of inexact algorithms, the exact implementation of these methods will not be given. Instead, more focus will be placed on the characteristics of each algorithm and the associated benefits and challenges.

Hirschberg

Later refined by Myers and Miller (1988), Hirschberg's (1975) algorithm utilizes a "divide and conquer" method to reduce the space needed for sequence alignments. In the algorithm, if sequences *A* and *B* are to be aligned, the method begins by cutting A into two substrings, *A1* and *A2*. Next, an optimal place for B to be separated is located to form *B1* and *B2*. The alignment is then derived recursively for the corresponding pairs of substrings. The weakness in this algorithm is the substantial increase in time from first analyzing the sequences to find the separation point and then solving the multiple sub-problems for the four substrings. However, because the solution is derived recursively, the amount of space needed is proportional to the length of the longer sequence, not a matrix between the two sequences. Depending on the length of the sequence, the trade-off may be worth it. If one has enough time/patience, the computer will eventually align even the longest of sequences using this method. Not the case for a computer which runs out of memory. In addition, time and space

calculations treat processor speed and memory independently. However, memory clearly has a noticeable effect on processor speed, and as such a savings in space may indeed translate into a decrease in time for larger sequences. Additionally, the "divide-and-conquer" aspect of the algorithm allows the programming of shipping subproblems to multiple processors to do the work in parallel.

k-difference

To increase speed of algorithms, upper limits can be placed on the number of allowable mismatches in the alignment. This method, generally referred to as k-difference or k-mismatch, reduces processing time by limiting the use of dynamic programming to a portion of the matrix proportional to the size of the larger sequence multiplied by the number of mismatches, *k* (Gusfield ,1997). The idea of k-difference can be applied to both global alignment as well as inexact matching, with inexact matching posing the added problem of gaps. Of particular concern are end gaps, which one would not want to factor into an edit distance calculation. The major benefit in the k-difference approach is the limiting of the distance the alignment can take away from the diagonal recursive path. Also, the user can decide whether to choose a smaller *k* and shorter processing time, or larger *k* for more iterations. A challenge with k-difference algorithms is identifying the proper size *k*. The efficiency of many of these algorithms are dependent on assumptions of certain-sized or bounded *k* (Chang, Lampe, 1993). The size or even presence of a limit on *k* may be dependent on the field of interest. Those studying more distantly related sequences would possibly benefit from a large *k* or no *k*. However, those interested in mutations and disease might be more inclined to keep *k* small in order to detect even the smallest differences.

Baeza-Yates and Perleberg

In an effort to further speed up k-difference calculations, Baeza-Yates and Perleberg (1992) developed a method of partitioning the pattern into equal length regions, such that there are k+1 regions. A keyword tree (common prefix) is built

with the regions and then an algorithm is run to find the locations in the string where patterns in those regions occur exactly. Finally, an approximate matching algorithm can be run to locate approximate substrings. As a result, because exact matches have previously been located and there is a limit on the number of edits which can be introduced, the sequence is aligned quickly. The idea has also been modified a number of times, including addition of the Shift-And method (Wu, Manber, 1992), and partitioning of the string, not the pattern (Chang, Lawler, 1994). The methods all result in an expected sublinear run time, proportional to the length of the string. However, the trade off is a higher occurrence of errors. Gusfield (1997) notes two more problems with these methods. First, the algorithms can often exclude regions which should be approximately matched. Secondly, in regions that do not end up matched, the algorithm still starts into dynamic programming or other resource intensive calculations. It follows that these algorithms could be more selective when handling the regions initially, and may benefit from intermediate calculations prior to longer calculations.

Four-Russians Speedup

Erroneously named for a paper by four authors (Arlazarov, et al, 1970), of which only one was Russian, the Four-Russians speedup has been used as an enhancement to many dynamic programming algorithms. The method uses "t-blocks" to describe a block of $t \times t$ area in a dynamic programming table (i.e., a table with the two sequences lined up on either axis). It then proceeds to compute by blocks, not individual cells, treating each block as one entity. By overlapping the blocks by one row/column and recognizing that neighboring cells can differ by at most one, the values for the overlapping areas can be calculated by the string and pattern data. The result is an increase in speed by a factor of t. For very large strings, the Four-Russians Speedup can make an appreciable difference. For smaller strings, it is typically not implemented in its fullest detail (Gusfield ,1997). The method also seems difficult to implement for blocks which are not of fixed length. So, one may need to choose an appropriate block length,

or determine a pattern to overlap the blocks. These steps may or may not be justified in smaller searches, as the parameters of a *t*-block are not immediately evident and may take time to discern.

Conclusion

With continuous advances in processor speed and growth of genomic information, the area of pattern matching remains a vital field. Newer methods will undoubtedly make use of technology for both the implementation and discovery of new algorithms. However, an understanding of fundamental pattern matching methods and challenges remains a necessity for those developing novel algorithms.

References

Arlazarov V.L., Dinic E.A., Kronrod M.A., and Faradzev I.A. On economic construction of the transitive closure of a directed graph. Dokl. Acad. Nauk SSSR, 194:487-88, 1970.

Baeza-Yates R. and Gonnet G. A new approach to text searching. Comm. ACM, 35:74-82, 1992.

Beals M. and Gross L. Amino Acid Frequency. The Institute for Environmental Modeling, June 2004 <http://www.tiem.utk.edu/~gross/bioed/webmodules/aminoacid.htm>.

Boyer R.S. and Moore J.S. A fast string searching algorithm. Comm. ACM, 20:762-72, 1977.

Chan S. Wikipedia, the free encyclopedia, June 2004 <http://en.wikipedia.org/wiki/Dynamic_programming>.

Chang W.I. and Lampe J. Theoretical and empirical comparisons of appromate string matching algorithms. Proc 3rd Symp. On Combinatorial Pattern Matching. Springer LNCS 644, p175-184, 1992

Chang W.I. and Lawler E.L. Sublinear expected time approximate string matching and biological applications. Algorithmica, 12:327-344, 1994.

Lecroq T. and Christian M. Exact String Matching Algorithms. Laboratoire d'Informatique de Rouen, June 2004 <http://www-igm.univ-mlv.fr/~lecroq/string/examples/exp8.html>.

Gusfield D. *Algorithms on strings, trees, and sequences: computer science and computational biology,* 1997.

Gusfield D. Simple uniform preprocessing for linear-time string matching. Technical Report CSE-96-5, UC Davis, Dept. Computer Science, 1996.

Hirschberg D. S. *A linear space algorithm for computing maximal common subsequences.* Comm. A.C.M. 18(6) p341-343, 1975.

Karp R.M. and Rabin M.O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31(2):249-260, 1987.

Knuth D.E., Morris (Jr) J.H., and Pratt V.R., Fast pattern matching in strings, *SIAM Journal on Computing* 6(1):323-350, 1977.

McCreight E. M. *A Space-Economical Suffix Tree Construction Algorithm.* Jrnl. of Algorithms, 23(2) pp262-272, 1976.

Myers E.W. and Miller W. Optimal alignments in linear space, CABIOS 4(1). 11-17, 1988.

Ukkonen E. *Constructing Suffix Trees On-Line in Linear Time.* In *Algorithms, Software, Architecture*, J.v.Leeuwen (ed), vol#1 of Information Processing 92, Proc. IFIP 12th World Computer Congress, Madrid, Spain, Elsevier Sci. Publ., pp484-492, 1992.

Weiner P. *Linear Pattern Matching Algorithms.* Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp1-11, 1973.

Wu S. and Manber U. Fast text searching allowing errors. Comm. ACM, 35:83-91, 1992.