Exact pattern matching: Adapting the Boyer-Moore algorithm for DNA searches

Exact pattern matching aims to locate all occurrences of a pattern in a text. Many algorithms have been proposed, but two algorithms, the Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM), are most widespread. It is the basis of some approximate string matching algorithms like BLAST, and in many cases it is desirable to locate an exact rather than approximate matches. Although several studies included measures with small alphabets, none of them specifically designed an algorithm to target nucleotide sequences. Since there are also no application programming interfaces available for pattern matching in nucleotide sequences, these two issues were aimed to be resolved. A portion of the *Chlamydomonas reinhardtii* genome (30 mega bases) was searched with queries ranging from 10 to 2000 nucleotides and an alternating number of matches between one and 25000. The results indicate that the use of two of the algorithms developed in this study is sufficient to efficiently cover the complete search space as presented in the experiment conducted here. Thus the aim of implementing an algorithm specifically targeting pattern matching in nucleotide sequences and making it available to the general public as an advanced programming interface was achieved. All algorithms are freely available at: http://bioinformatics.iyte.edu.tr/supplements/peerj/.
Exact pattern matching: Adapting the Boyer-Moore algorithm for DNA searches

Jens Allmer

Molecular Biology and Genetics, Izmir Institute of Technology, Urla, 35430, Izmir, Turkey
Abstract

Exact pattern matching aims to locate all occurrences of a pattern in a text. Many algorithms have been proposed, but two algorithms, the Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM), are most widespread. It is the basis of some approximate string matching algorithms like BLAST, and in many cases it is desirable to locate an exact rather than approximate matches. Although several studies included measures with small alphabets, none of them specifically designed an algorithm to target nucleotide sequences. Since there are also no application programming interfaces available for pattern matching in nucleotide sequences, these two issues were aimed to be resolved. A portion of the *Chlamydomonas reinhardtii* genome (30 mega bases) was searched with queries ranging from 10 to 2000 nucleotides and an alternating number of matches between one and 25000. The results indicate that the use of two of the algorithms developed in this study is sufficient to efficiently cover the complete search space as presented in the experiment conducted here. Thus the aim of implementing an algorithm specifically targeting pattern matching in nucleotide sequences and making it available to the general public as an advanced programming interface was achieved. All algorithms are freely available at: http://bioinformatics.iyte.edu.tr/supplements/peerj/.
Background

Exact pattern matching aims to find all occurrences of a pattern located in a text. As proposed by Woo-Cheol et al., exact pattern matching may be useful in finding sequences in DNA (Woo-Cheol, et al., 2006). Although approximate pattern matching is currently prevailing, exact pattern matching forms the basis for some approximate pattern matching algorithms such as BLAST (Altschul, et al., 1990) and GPF (Allmer, et al., 2004). In many cases, a gene needs to be retrieved from a collection, for instance a FASTA file (Pearson and Lipman, 1988), where part of the sequence is known; in these cases exact pattern matching can be helpful. Furthermore, it may be useful to perform exact pattern matching prior to approximate pattern matching to save computational time by removing queries which exactly match before employing algorithms with higher computational cost such as those of Smith and Waterman and Needleman and Wunsch {Needleman, 1970 #411; Smith, 1981 #412}.

Two algorithms can be clearly distinguished from the mass of proposed methods for exact pattern matching. The Boyer-Moore (Boyer and Moore, 1977) algorithm (BM), and in general algorithms that compare the pattern to the text on a right to left basis, are well known for their fast run-time in practice (Lecroq, 1995). The Knuth-Morris-Pratt (Knuth, et al., 1977) algorithm, based on matching from left to right, is considered to be better suited for searching in streams although using buffering along with BM style algorithms should be able to change this assumption.

Clearly, the best case complexity of BM alike algorithms of $O(n / m)$ (Hume and Sunday, 1991) cannot be improved upon. Variations of the BM algorithm, such as the Boyer-Moore-Horspool (Horspool, 1980) (BMH) implementation, make it faster on the average (Apostolico and Giancarlo, 1986, Crochmore, et al., 1992, Galil, 1979). Nucleotides represent a very small alphabet (here 6 characters A,C,G,N,T,U) which may contain highly repetitive subsequences. Therefore, BMH, which removes the good suffix shift, may not be well suited for searching nucleotide sequences.

Musser and Nishanov (Musser and Nishanov, 2002) claim that the skip loop of the fast BM algorithm performs poorly with small alphabets and long patterns, prevalent in searches in DNA. Their approach to solving the problem is straight forward and uses hashing. Clearly, transforming the alphabet to a different space, using hashing or so called q-grams (Kytojoki, et al., 2003) for example, can be useful in searching through DNA sequences. The drawback of using this approach is that several characters need to be evaluated in each step which usually involves an overhead of, for example, hashing. Two problems arise in strings from small alphabets: the time spent in the skip loop is reduced while the number of times that a match needs to be evaluated in detail is increased. This is even more pronounced when a large number of matches are expected in the text or if the suffix of the pattern is abundant in the text.

Raita created a variant of the BMH algorithm which introduced sentinels in order to speed up searches by first comparing the parts of the pattern with the weakest dependencies (Raita, 1992). He reported an improvement of approximately 25% over the BMH algorithm but it has been shown by Smith to be solely due to sentinel use, as opposed to character dependencies within the pattern, as Raita concluded (Smith, 1994). Small alphabets have been studied, but so far the Boyer-Moore algorithm has not been specifically adapted to work with small alphabets such as nucleotides {Hume, 1991 #395; Lecroq, 1995 #390}.

As opposed to preprocessing the pattern as done in the algorithms above, in fact all algorithms in this study, the text can also be processed. Indexing makes it possible to quickly locate a pattern. This usually restricts the patterns that can be retrieved as using the appropriate index. Ricardo et al. pointed out that indices take extra space and time to build and update. They propose that plain text search may be fast enough for even larger texts (Baeza-Yates and
Navarro, 2004). In natural languages, words can be used to index a text; but so far a genome cannot be split into words, which makes indexing problematic.

This study investigates the original fast Boyer-Moore algorithm, including skip loop, and compares it to versions including q-grams (infixes) and sentinels. Additionally, two changes to the Boyer-Moore algorithm are proposed which even allow for shifts when an initial match needs to be evaluated. The lookup table for the skip loop seems to be unnecessary and is therefore completely removed in most of the algorithms, slightly reducing the algorithm’s overhead. The semantics of the b1-shift table of the original BM algorithm is slightly changed so that it can be utilized for the skip loop as well as for all other evaluations. It also allows for shifts in case of partial matches which is a great benefit in case of small alphabets. It transpires that the shift can be calculated by using the maximum of the suffix shift and the shift at the position of the mismatch. In order to save computation, either the suffix shift or the mismatch shift is used in the algorithms developed in this study. In addition to the fast skip loop sentinels are employed in some of the algorithms, enabling more time to be spent in the skip loop, and less for extensive text character versus pattern character comparisons.

Seven algorithms were implemented in this study and their details are provided in the Algorithms Section below. The algorithms developed in this study are compared to straightforward implementations of BM, and the often overlooked fast BM (Hume and Sunday, 1991, Baeza-Yates and Navarro, 2004) and BMH incorporating a three fold unrolled skip loop, which reduces the overhead of the skip loop, as proposed by Hume and Sunday (Hume and Sunday, 1991). For the comparison the experimental settings outlined in the Materials and Methods Section are used. The results are presented and discussed in the Results and Discussion Section following the Algorithm Section finally Conclusion and Outlooks are provided. Due to the large search space represented by DNA sequences, KMP and brute force algorithms, as well as many other competing algorithms were not assessed since their predicted performance is too low to warrant evaluation. Although Davies and Bowsher propose to use KMP for small alphabets, this is ignored here since it seems to be due to the fact that only the first occurrence of the pattern in the text was desired and that only the bad character shift was employed as in BMH (Davies and Bowsher, 1986) which constantly performs worse than most other algorithms employed in this study.

Since the knowledge of fast algorithms is not commonplace and their implementation requires a vast amount of research (Hume and Sunday, 1991) and to the best of my knowledge, an advanced programming interface is not available, a library written in JAVA™ including all the algorithms presented in this study is provided and available for download at http://bioinformatics.iyte.edu.tr/supplements/peerj/

### Algorithms

The notations used in the following sections that describe the algorithms need some explanation which will be presented first. A text \((t)\) is a sequence of characters \((c)\) from an alphabet \((\Sigma)\) with the length of the text given as \(n\). The size of the alphabet can be given using the number of distinct symbols \((\alpha)\) of which it consists. In exact pattern matching, the aim is to locate a pattern \((p)\) in \(t\) where \(m\) is the length of the pattern and the alphabet of \(p\) is equal or a subset of the alphabet of \(t\). Often it is beneficial to use a window of several characters whose size is specified as \(w\) in the following.

The algorithms presented in this study can be thought of as being modular, consisting of several building blocks. Hume and Sunday as well as Cleophas et al. have previously broken-up search algorithms into components (Hume and Sunday, 1991, Cleophas, et al., 2004). The
components used in this study are shift tables, skip loops, sentinels, and the character comparison. The latter is not listed in the table since the same character by character comparison was used in all algorithms. As an overview the general composition of the algorithms employed in this study is presented in Table 1.

Table 1: shows the rough composition of the algorithms employed in this study. The table is a mere preview thus the abbreviations for the algorithms as well as skip loops and so forth need to be discussed in some more detail in the appropriate section below.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Shift tables</th>
<th>Skip loop</th>
<th>Sentinels</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmOrg</td>
<td>d1, d2</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>bmFast</td>
<td>d0, d1, d2</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>BMH</td>
<td>d0, d1</td>
<td>three fold unrolled</td>
<td>none</td>
</tr>
<tr>
<td>bm4DNAHM</td>
<td>similar d1</td>
<td>three fold unrolled</td>
<td>none</td>
</tr>
<tr>
<td>bm4DNAHMbs</td>
<td>similar d1, suffix shift</td>
<td>three fold unrolled</td>
<td>1</td>
</tr>
<tr>
<td>bm4DNA</td>
<td>increment</td>
<td>three fold unrolled</td>
<td>w</td>
</tr>
<tr>
<td>bm4DNAIS</td>
<td>d0, d1</td>
<td>three fold unrolled</td>
<td>w</td>
</tr>
<tr>
<td>bm4DNADS</td>
<td>d0, d1, suffix shift</td>
<td>three fold unrolled</td>
<td>w</td>
</tr>
</tbody>
</table>

The Boyer-Moore algorithm will now be presented in more detail, such that the changes introduced by the algorithms developed in this study can be highlighted within this context.

**Boyer-Moore String Matching Algorithm**

A fast exact pattern matching algorithm was published by Boyer and Moore (Boyer and Moore, 1977). In fact, they proposed two algorithms, one for presentation and another which performs best in practical applications (Hume and Sunday, 1991). Since then, some improvements or modifications have been added and parts of the algorithm have been proposed as non beneficial in certain situation (Hume and Sunday, 1991, Horspool, 1980, Raita, 1992). In general, the algorithm and its variants build on the idea that comparing a pattern from its rightmost end to the text in question allows for larger shifts which can be pre-computed from the pattern. Therefore, the best-case complexity is $O(n / m)$ since a pattern of length $m$ can be aligned $n / m$ times with a text of length $n$. Given that there is no match between $p$ and $t$, $n / m$ is the amount of comparisons necessary to establish the fact that $p$ is not in $t$.

In brief, the basic algorithm preprocesses the pattern by creating two look-up tables. One (d1) holds the shifts that are possible if a character mismatch between pattern and text is encountered. The size of this table is equal to $a$. The other look-up table contains shifts based on reoccurring infixes also called “good suffix shift”. This table was introduced to improve on the worst case complexity, $O(n * m)$ but was dropped by Horspool (Horspool, 1980). Cole showed that on the average the BM algorithm has a complexity of $O(n)$ in regards to the number of character comparisons (Cole, 1990). In addition to d1 and d2, the fast BM algorithm makes use of a third look-up table called d0.

After the look-up tables have been filled in the preprocessing phase, the pattern is aligned with the text and the right-most character of the pattern or q-gram is compared to the corresponding part of the text. If equal, the subsequent characters need to be compared. In the opposite case, the pattern may be shifted along the text by a shift whose step-size can be deduced as the maximum shift found in the two lookup tables, d1 and d2.
The Basic Boyer-Moore Algorithm

The following definition of the Boyer-Moore algorithm omits a skip loop because it is widely ignored in practice (Lecroq, 1995). This problem, as pointed out by Horspool (Horspool, 1980), is only present in this basic algorithm, all following algorithms contain a skip-loop. The basic algorithm, as shown below, will be referred to as \texttt{bmOrg} in the remainder of this paper. First the processing of the pattern will be detailed with the building of the look-up tables and then the algorithm for the searching phase will be given. The definition of the algorithm differs from the one given in the original paper but seems more appropriate in this context. The algorithms, developed in this study, are loosely built on the basic algorithm, and will be explained using references to locations in the basic algorithm (notice line numbers).

Preprocessing Phase
First the delta1 (d1) table, also called bad character shift, for looking up a shift value given a mismatch between two characters will be defined. Obviously, the table requires an extra space of \(O(\alpha)\) and is preprocessed in time \(O(m + \alpha)\).

```plaintext
10 foreach c in \(\Sigma\) do
20   let \(d1[c] = m\)
30 end foreach
40 for i from 0 to m do
50   let \(d1[p[i]] = m - i\)
60 end for
```
Now the delta2 (d2) table, also called good suffix shift, for looking up a shift value given a mismatch position as an index of p will be defined. An extra space of $O(m)$ is obviously needed to represent this table. The time complexity to construct this table is also $O(m)$.

```
10 i = m, j = m + 1
20 fbm[i] = j
30 do while i > 0
40   do while j <= m and p[i-1] != p[j-1]
50     if d2[j] = 0 then
60       d2[j] = j - i
70     end if
80     j = fbm[j]
90   end while
100 i--
110 j--
120 fbm[i] = j
130 end while
140 j = fbm[0];
150 for i:0 to m do
160   if d2[i] = 0 then
170     d2[i] = j
180   end if
190   if i = j then
200     j = fbm[j]
210   end if
220 end for
```

The two shift tables, d1 and d2, are needed in the search phase to determine the amount of characters the pattern can be progressed along t for the next attempt given a mismatch or a complete match in the previous attempt.

**Search Phase**

The algorithm used in the searching phase will be detailed next. The algorithm’s best and worst case complexities have been established previously (see above).

```
10 let tp = 0  //tp: text pointer
20 do while tp < n - m
30   let j = m
40   do while j>0 and t[tp+j] = p[j]
50     j--
60   end while
70   if j <= 0 then
80     reportMatch(tp+1)
90   let tp = tp + max(d1(tp+j),d2(j))
100 end if
110 let tp = tp + max(d1(tp+j),d2(j))
120 end while
```
Fast Boyer-Moore

This implementation of the BM algorithm, abbreviated as bmFast, incorporates the fast loop as described in the original paper (Boyer and Moore, 1977).

Preprocessing Phase

In addition to the d1 and d2 table of bmOrg, a table called d0 is needed which will be used during the skip loop. It needs an extra space of O(\(\alpha\)) and can be constructed in O(\(\alpha\)) time as a deep copy of the d1 table.

\[
\begin{align*}
10 & \quad \text{Let } d0 = d1 \\
20 & \quad \text{Let } d0[p[m]] = 2 \times n
\end{align*}
\]

Search Phase

The skip loop in the fast BM algorithm has been proposed since most time is spent in sliding the pattern along t due to immediate mismatches. The skip loop can be implemented as an addition to the basic algorithm. Note, that the line numbers can be inserted into the basic algorithm and that if line numbers are equal they supersede the original definition.

\[
\begin{align*}
21 & \quad \text{do while } tp < n \\
22 & \quad \text{let } tp = tp + d0[t[tp]] \\
23 & \quad \text{end while} \\
24 & \quad \text{if } tp < 2 \times n \text{ then} \\
25 & \quad \text{break} \\
26 & \quad \text{end if} \\
27 & \quad \text{let } tp = tp - 2 \times n - 1 \\
30 & \quad \text{let } j = m - 1
\end{align*}
\]

Boyer-Moore-Horspool

Preprocessing Phase

The Boyer-Moore algorithm, abbreviated as BMH, was altered by Horspool by simply dropping the good suffix shift (d2) and by reintroducing the skip loop. The good suffix shift may not be of practical value in larger alphabets where no extensive internal repeats may be expected. In DNA sequences composed of a very small alphabet, even in short query sequences internal repeats can be abundant therefore the good suffix shift may be beneficial. Nonetheless it has not been implemented in this algorithm.

\[
\begin{align*}
10 & \quad \text{Let } d0 = d1 \\
20 & \quad \text{Let } d0[p[m]] = 0
\end{align*}
\]

Search Phase

The implementation here, unlike the one given by Horspool, uses three fold unrolling of the skip loop as proposed by Hume and Sunday (Hume and Sunday, 1991). It depends on the suffix being set to zero in the d0 table (see Preprocessing Phase). The complexity is the same as established previously for bmOrg and bmFast.

\[
\begin{align*}
21 & \quad k = d0[t[tp]] \\
22 & \quad \text{do while } k = 0 \\
23 & \quad \text{let } tp = tp + (k = d0[t[tp]]) \\
24 & \quad \text{let } tp = tp + (k = d0[t[tp]]) \\
25 & \quad \text{end while}
\end{align*}
\]
The next algorithm that will be discussed has been designed in this study to specifically adapt the BM algorithm for best performance while searching nucleotide sequences.

**Boyer-Moore-HM**

This algorithm, abbreviated bm4DNAHM, uses a hash table (shift table) to store windows of size $w$ from the pattern. Hashing has been used in exact pattern matching from the beginning but the complete pattern was hashed which has led to a large overhead during searching (Karp and Rabin, 1987). Here, only subsets of the pattern are used and the size of the word $w$ is calculated from $m$ and $\alpha$ as described in Wu and Manber (Wu and Manber, 1994). From left to right, the sequences are extracted; their hash value and the potential shift along $t$ are calculated and stored in the shift table, excluding the suffix. This has two advantages, one being that occurrences of a pattern that has been encountered before will update the shift value in the shift table, and the other being that the shift table can almost directly be used and the actual shift values need not be repeatedly calculated in the search phase. Therefore, only a single pass over the pattern is necessary. Finally, the hash value of the suffix is calculated and stored in the shift table. If the hash is associated with another substring of $p$, then the shift value present in the table is stored as the suffix shift otherwise the suffix shift is set to $m - w$. In every case, the value in the shift table is set to zero. The complexity for the preprocessing phase is $O(m \times w)$ where $w \ll m$.

During the matching phase, the algorithm makes use of the shift table, the details of which are outlined below. Processing the text from left to right, $w$ sized portions of $t$ are extracted from the right most position of the alignment of $t$ and $p$, ending at the current position of the text pointer. The hash value for the $w$ sized portion of $t$ is calculated using the function as specified below and then the shift is determined using the shift function detailed below. It returns the largest possible shift for the hash value given or zero if the hash cannot be found in the shift table. In order to diminish the overhead introduced through the skip loop, the statement is repeated 3 times. Hume and Sunday determined three-fold unrolling as the best value in their assessment (Hume and Sunday, 1991). In case a potential match is encountered, it is first determined whether it is the suffix of $p$ or an infix. If it is indeed the suffix, the remainder of $p$ is compared character by character against $t$ until the end of $p$ or a mismatch is encountered. Any matches encountered are reported. Regardless of match or mismatch, $p$ is progressed along $t$ by the shift value stored for the suffix hash.

Two functions are needed for this algorithm which will be shown first. Afterwards the algorithms for preprocessing phase and search phase are defined.

**Shift function**

```
10   shift(key) { 
20     let res = dl[key] 
30     if res = notFound then 
40       return m - w 
50     end if 
60     return 0 
70   }
```

**Hash function**

```
10   hash(p) { 
20     let hashVal = 0 
30     for j:0 to w do 
40       let hashVal = (hashVal * 128 + p[j]); 
50     end for 
60     return hashVal 
```
Preprocessing phase
A single shift table is created in the preprocessing phase, which needs an extra space of $O(m)$ and can be created in $O(m)$ time. This table is similar to the bad character shift table in bmOrg except for the transformed alphabet and shall therefore be referred to as $d1$. Only the infixes of $p$ are represented in $d1$ which leads to the small extra space. Another extra space of $O(m)$ is needed since $p$ is appended to $t$.

```
10 append $p$ to $t$
20 foreach infix of size $w$ in $p$, excluding the suffix (from left to right)
30    let key = hash(infix)
40    let $s = shift(key)$
50    let $d1[key] = s$
60 endforeach
70 let key = hash(suffix)
80 if key not in $d1$ then
90    let suffixShift = $m-w$
100 end if
110 suffix = key
```

Search Phase
The search phase introduces an unspecified function getWindow(int,int), which in practice is implemented as an inline for loop, and returns the $w$ sized string portion of $t$ at the specified position in $t$. Clearly the best case complexity reduces to $O(n / (m - w))$ due to the $w$ sized suffix that needs to be checked. Here, the array operations for getting the $w$ sized portion are included. If only the number of comparisons is considered, the best case complexity is equal to bmOrg’s.

```
Start with bmORG
21 key = hash(getWindow(tp,w))
22 $k = shift(key)$
23 do while $k != 0$
24    let tp += ($k = d1[hash(getWindow(tp,w))])
25    let tp += ($k = d1[hash(getWindow(tp,w))])
26    let tp += ($k = d1[hash(getWindow(tp,w))])
27 End while
28 If key != suffix then
29    let tp += d1[key]
30    continue
31 end if
32 If tp > n then
33    break
34 End if
35 let $j = m$
Continue with bmOrg
110 let tp += d1[key]
Continue with bmOrg
```
**Boyer-Moore-HM best solution**

This algorithm, abbreviated as bm4DNAHMbs, is rather similar to the algorithm described above. A significant difference is that only the encounter of the suffix will halt the skip loop which means that this algorithm has the highly desirable potential to remain in the skip loop longer than the previous one. Another advantage is that the check for the hash being the suffix is removed, which requires a slight adaptation of the preprocessing phase is necessary because of this. The shift for the suffix is now zero and the real shift that would be possible needs to be stored in an extra variable. A sentinel chosen as $p[0]$ is installed and checked after a suffix match has been established. The value stored in suffixShift can be used to progress the pattern along the text since it would align with the first following internal repeat, if any. The complexity for both preprocessing and matching phase remains unchanged for the best case, but improvements on the average can be expected. The hash function is equal to the previous algorithm.

**Shift function**

The shift function is different from the one used in bm4DNAhm and always returns a positive shift unless the suffix is encountered which then returns 0 (see Preprocessing Phase).

```
10  shift(key) {
20      let res = d1[key]
30      if res = notFound
40         return m-w
50      return res
60  }
```

**Preprocessing phase**

In contrast to the bm4DNAhm a sentinel is used in this algorithm which is defined during preprocessing. Furthermore, the suffixKey member is removed and instead the suffix shift is stored.

```
5      Let sentinel = p[0]
Continue with bm4DNAhm
80      if key not in d1 then
90         let suffixShift = m - w
100     else
110        let suffixShift = d1[key]
120     end if
130    let d1[key] = 0
```

**Search Phase**

Although only a small number of changes have been made to bm4DNAhm but they may prove to be crucial for the practical behavior of the algorithm in the average case while the best case complexity remains unchanged. The suffix shift is used since it can potentially provide the largest shifts. Similar to the $d_2$ table which considers subsequences, the suffix shift aligns the text with the next occurrence of the suffix in $p$. It is to be expected that the algorithm would perform better, on the average, than bm4DNAHM. An improvement would be to shift the maximum of suffixShift and the shift determined from $d_1$ however this was beyond the scope of this study. The complexity of the algorithm remains unaffected by these changes.

Start with bm4DNAhm
Boyer-Moore-4DNA

Instead of using hashing and extracting $w$ sized portions from $t$ to speed up processing, the following algorithm uses sentinels, which are created by storing the first $w$ characters of $p$ from left to right in separate variables that can be quickly checked prior to comparing the remaining characters of $p$ to $t$. Sentinels are checked from left to right which while not being detrimental, may or may not be beneficial (Hume and Sunday, 1991). In case a sentinel determines a mismatch, $p$ is progressed along $t$. In case none of the sentinels report, the remainder of the pattern is checked against $t$ until a match or mismatch is determined. The complexity of the preprocessing phase is $O(2^\alpha + m)$ where $\alpha$ is 128 since the ASCII character set was used for simplicity. In this specific case, however, $\alpha$ should be much smaller than $m$ ($\alpha = 6; 9 < m < 2000$). In the best case the complexity is the same as determined for BMH while on the average runtime some improvements should be observed. In the following this algorithm will be referred to as bm4DNA.

Preprocessing Phase

In the preprocessing phase $w$ sentinels must be installed. This increases the extra space needed by $w$ and increases the extra time needed by $w$ as well.

```plaintext
1  Append p to t
2  sentinel0 = p[0]
3  sentinel1 = p[1]
//Until desired number of sentinels has been established
continue with BMH
```

Search phase

In the search phase each sentinel is checked prior to entering the check loop as in bmOrg which here is reduced by $w$ since the sentinels have already been checked (not shown below). The shifts in this algorithm are mere increments of one. Due this the best case complexity reduces to $O(n)$.

```plaintext
26  if t[tp - m] != sentinel0 then ,
27    let tp++
28  continue
29  end if
check for all sentinels
continue with BMH
110  tp++
continue with BMH
```

Boyer-Moore-IS

This variation of the 4DNA algorithm, termed bm4DNAIS, uses sentinels as well but differs in several other aspects. In the preprocessing phase, it is little different from BMH except for dropping the d0 table and instead using a negative value to represent the suffix’s membership to $p$. The preprocessing complexity is thus reduced to $O(\alpha + m)$. Dropping the d0 table...
However leads to the problem that the skip loop cannot be unrolled since the values returned may be negative rather than zero. Nonetheless, by merely multiplying the shift by -1 it can be used to progress $p$ along $t$ in case a mismatch, or when a complete match is encountered, which is better than increments by one as in bm4DNA. Using only a negative value for the suffix also guaranteed that in further processing at least the suffix of $p$ matches $t$ in the alignment.

In case a mismatch or match is encountered the shift can be determined by multiplying the shift value of the suffix by -1.

**Preprocessing phase**

1. Append $p$ to $t$
2. sentinel0 = $p[0]$
3. sentinel1 = $p[1]$
4. 
5. //Until desired number of sentinels has been established
6. continue with bmOrg
7. Let $d1[p[m]] = d1[p[m]] * -1$

**Search phase**

Instead of using simple increments for shifting as in bm4DNA, the $d1$ shift table is used and thus the best case complexity is the same as for bmOrg.

```plaintext
Start with bmFast
if $t[tp - m] != sentinel0$ then ,
let $tp += d1[t[tp]] * -1$
continue
end if
check for all sentinels
continue with BMH
let $tp += d1[t[tp]] * -1$
continue with BMH
```

**Boyer-Moore-DS**

Although reasonably similar to the algorithm above, some changes in the algorithm should lead to significant differences in runtime. Instead of using just one shift table, two tables are used, one representing the potential shifts with the suffix set to zero ($d0$) and the other representing the potential shifts with the suffix containing a valid shift value similar to the basic algorithm introduced at the beginning (bm4DNA). Some space is wasted in this case since two tables of size $\alpha$ need to be maintained. In addition, some time needs to be spent for cloning the array and adjusting the suffix to zero which leads to a slightly different complexity of $O(2 \alpha + m)$.

The gain is that during the search phase the skip loop can be unrolled again which should lead to improvements on the average, while leaving best case complexity of BMH unaffected. In contrast to the basic algorithm, previously introduced (4DNA) the increments for progressing $p$ along $t$ are taken from the shift table whereas they are increments by one in the basic version. This algorithm will be referred to as bm4DNADS.

**Preprocessing phase**

Initially, the sentinels and the $d1$ table are created as in bm4DNAIS. Afterwards the $d0$ table is created and the suffix is set to zero as in BMH.

Start with bm4DNAIS
continue with BMH
Search phase

Instead of using simple increments for shifting as in bm4DNA or shifting character dependent as in bm4DNAIS, the shift is always the value stored for the suffix. This is potentially the largest possible shift if there are no repetitions of the suffix in the pattern. The complexity is the same as for bmOrg.

Start with BMH

```plaintext
26 if t[tp - m] != sentinel0 then ,
27      let tp += d1[p[m]]
28      continue
29 end if
```

check for all sentinels

continue with BMH

```plaintext
110 let tp += d1[p[m]]
continue with BMH
```
Setup for Experimental Study

The variations of the Boyer-Moore algorithm were implemented in JAVA™. A library is provided on the projects webpage for ease of use. All algorithms were used to find exact patterns within 30,000,000 nucleotides which were read into memory and processed before timing the algorithms. A section of the genome of *Chlamydomonas reinhardtii* was used for the analysis (chlre2.fasta, JGI). All characters that were not nucleotides, such as whitespace, as well as all N’s (wildcard for any nucleotide) were removed. The length of the patterns ranged from 10 to 2000 and the number of expected matches within the nucleotide sequences varied between 10 and 25000, where the pattern is a random string from the nucleotide sequence. A pattern length of 2000 was chosen as the upper bound due to an analysis done by Woo-Cheol et al. (Woo-Cheol, Sanghyun, Jung-Im, Sang-Wook and Jee-Hee, 2006). The matches were introduced into the sequence by replacing parts of it with the expected sequence. The expected sequences were equally distributed over the range of the sequence. The runtime of the algorithms was measured using the JAVA™ system function nanoTime(). Each measurement was repeated 16 times, and the three lowest and three highest values were discarded and the remaining 10 values were averaged. Note that this runtime measurement corresponds neither to seconds nor to any other lesser unit of time. Measurements may therefore only be used to establish differences in runtime of various algorithms measured with highly similar environmental constraints (e.g. same computer).

The analysis was performed for ten different queries on a personal computer running a Microsoft Windows™ operating system. Another set of 10 different queries was analyzed on a UNIX™ system. On both systems the analysis was done with as few other processes competing for the processor time as possible.

For each algorithm which allows for the setting of a window, its size was calculated as proposed in Wu and Manber (Wu and Manber, 1994) where \( w = \log_2 m \) (see Algorithms section for explanation of the variables). Patterns were

- Number of hits: [10, 100, 1000, 10000, 25000]
- Pattern lengths: [20, 50, 100, 500, 1000, 2000]

In a second experiment fewer hits were introduced into the sequence. Two algorithms more (fast BM and bm4DNAHMbs) were evaluated but the pattern lengths as well as the hits were chosen from finer grained sets.

- Number of hits: [1, 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
- Pattern lengths: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 250, 500, 750, 1000, 2000]

Instead of 10 different patterns for the two environments, 34 (Microsoft Windows™) and 40 (UNIX™) distinct patterns were evaluated.
Results and discussion

The algorithms developed in this study and the algorithms implemented as references were evaluated in regards to their runtime usually given in milliseconds per megabyte. For each search space here, given by the length of the query, the length of the text, and the number of hits, the runtime of all algorithms was measured.

Table 2: For each input space composed of query length and number of hits, the algorithm with the highest speed is shown (ms/ MB), along with its average performance and how many times it performed best out of 74 experiments (experiment 2, see experimental setup). Next to the best algorithm, the algorithm which won most or at least an equal amount of times is listed. The overall fastest processing time is highlighted. Not all data are shown for simplicity, but is available in the online supplement.

<table>
<thead>
<tr>
<th>Query Length</th>
<th>Hits</th>
<th>w</th>
<th>Fastest Algorithm</th>
<th>Prevalent Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Algorithm</td>
<td>Highest Speed</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>bm4DNAIS</td>
<td>1.61</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
<td>bm4DNAIS</td>
<td>1.62</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>2</td>
<td>bm4DNAIS</td>
<td>1.62</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>2</td>
<td>bm4DNAIS</td>
<td>1.61</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>2</td>
<td>bm4DNAIS</td>
<td>1.62</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>3</td>
<td>bm4DNADS</td>
<td>1.11</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.32</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.31</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.33</td>
</tr>
<tr>
<td>50</td>
<td>1000</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.34</td>
</tr>
<tr>
<td>70</td>
<td>1</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.24</td>
</tr>
<tr>
<td>70</td>
<td>10</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.06</td>
</tr>
<tr>
<td>70</td>
<td>100</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.06</td>
</tr>
<tr>
<td>70</td>
<td>500</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.06</td>
</tr>
<tr>
<td>70</td>
<td>1000</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.07</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>3</td>
<td>bm4DNADS</td>
<td>1.15</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.08</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>3</td>
<td>bm4DNAIS</td>
<td>1.08</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.09</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>3</td>
<td>bm4DNA</td>
<td>1.09</td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>4</td>
<td>bm4DNAHMbs</td>
<td>0.71</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>4</td>
<td>bm4DNAHMbs</td>
<td>0.69</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>4</td>
<td>bm4DNAHMbs</td>
<td>0.70</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>4</td>
<td>bm4DNAHMbs</td>
<td>0.73</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>4</td>
<td>bm4DNAHMbs</td>
<td>0.78</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>5</td>
<td>bm4DNAHMbs</td>
<td>0.36</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>5</td>
<td>bm4DNAHMbs</td>
<td>0.37</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>5</td>
<td>bm4DNAHMbs</td>
<td>0.38</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>5</td>
<td>bm4DNAHMbs</td>
<td>0.45</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>5</td>
<td>bm4DNAHMbs</td>
<td>0.53</td>
</tr>
</tbody>
</table>

For each of the 74 runs (34 on Microsoft Windows™ and 40 on UNIX™) that were evaluated, the best algorithm was determined for the various search spaces. Table 2 presents...
the fastest algorithm and how often it performed best, as well as showing, which was most frequently the best. In the smaller search spaces (up to a pattern length of 100) algorithms including sentinels, perform better than the algorithm employing hashing. At pattern lengths of 100, these algorithms, namely bm4DNA, bm4DNAIS, and bm4DNADS present the fastest algorithms, but they do not consistently perform well. Here, in most cases, bm4DNAHMbs outperforms all other algorithms. The search speed for this algorithm increases with longer patterns since the word size $w$ is adaptive and shifts are therefore greater with increasing pattern length. Runtime increases slightly with increasing number of hits. Since all measured data points would overload Table 2 too much, only a small selection of results is shown but the complete set of results can be found in the online supplement.

The runtime increases with the number of hits as can be seen in Table 2 but this is more pronounced in experiment 1 where greater numbers of hits were studied. This effect can be seen for all algorithms most dramatically at 25000 hits. Note that 25000 hits multiplied by a pattern length of 1000 means that the text almost exclusively consists of concatenated patterns, leading to a large number of character comparisons and thus an extended runtime. In the case of 25000 matches for a pattern length of 2000 the experiment could not be performed since the text was too small to accommodate this amount of matches. Interestingly, the fast Boyer-Moore algorithm performed well with a pattern length of 50. Although not being the fastest algorithm overall for this search space it outperforms all others in the majority of experiments. This is not as pronounced as the performance of bmDNAHMbs at larger query lengths yet approximately 40% of the time bmFast is the prevalent algorithm at a query length of 50. This factor can be explained by the preprocessing, which because of the b2 table is more involved for bmFast as compared to the algorithms which employ sentinels such as bm4DNA. Once the gain in processing speed, due to using the b2 table, eats up the extra time needed to build this table, bmFast becomes faster than the algorithms employing sentinels. The same is true for the algorithms using hashing, where this overhead takes slightly longer to be compensated. At a query length of 70 bmDNAHMbs is already the prevalent, but not the fastest algorithm, yet. Therefore, the space occupied by bmFast as the prevalent algorithm is restricted to a small area of input space.

In an assessment of the first experiment the relative performance of the algorithms were compared for a number of different pattern lengths. Since for smaller pattern lengths the best algorithm was not as decisive as with longer patterns where the algorithm involving hashing performed best in almost all cases, the majority of pattern lengths compared in Figure 1 is rather short.
Interestingly, the performance of the algorithms was relatively comparable on the two computers employed in this study, for both experiments 1 and 2, hence the results for all 20 queries of experiment 1 and all 74 queries in experiment 2 were averaged. Figure 1 shows the averaged results giving the runtime for the algorithm for different numbers of hits. For a pattern length of 20, bm4DNAIS and bm4DNADS perform best approximately 23% better than bmOrg and almost 30% better than the optimized version of BMH. At this pattern length, bm4DNAHM takes approximately 200% as long for the pattern matching as all the competing algorithms. With a slight increase in pattern length to 50 nucleotides, bm4DNAHM is no longer the slowest algorithm. At this pattern length, BMH performs least effectively, which may be due to the absence of the d2 shift table. Many larger shifts may be missed and smaller shifts may be abundant because of the small alphabet investigated here. As compared to a pattern length of 20, the overall performance decreased by almost 20% and the differences between bm4DNADS and bmOrg decreased to about 6% whereas the difference to BMH remained almost constant at 30%. The largest difference to the performance with a pattern length of 20 is apparent for bm4DNAHM where the difference in speed falls from 200% to less than 25% slower than bm4DNADS and even more than 5% faster than BMH. This trend continues with longer pattern lengths; at a length of 100 nucleotides, bm4DNAHM is the fastest algorithm. The overall speed increases by 20% as compared to the initial pattern length of 20 and bm4DNAHM is approximately 20% faster than bmOrg. The latter algorithm in turn performs slightly better than bm4DNADS and bm4DNAIS, which were more efficient with shorter pattern lengths. With a further increase in pattern length to 1000, the overall speed increases again so that the necessary runtime is only 20% of that needed to solve a problem of pattern length 20. The relative performance difference of bmOrg and bm4DNAHM also increases to about 76%. It would appear that bm4DNADS and bm4DNAIS, which employ sentinels, perform very well with short pattern lengths whereas bm4DNAHM performs rather less sound in those
search spaces. Once the overhead incurred by hashing is amortized, as it is with pattern lengths somewhere in-between 50 and 100 (for bm4DNAHM), the performance is unequalled. The overhead may also be overcome by increasing the text length instead of the pattern length, which will be a topic for future investigation.

For the second experiment, the average of the standard deviation in percent across all the measurements was calculated for about 4000 sets of 16 measurements for each algorithm. Again, the three highest and the three lowest measurements were discarded, while the remaining measurements were averaged and the standard deviation was calculated. Although the averages across different search spaces may not be computable into an overall average, the standard deviation can be used in this manner. Thus, across all queries and all search spaces the average of the standard deviation in percent of the average was calculated and can be seen in Table 3.

Table 3: Shows the average of the standard deviations for the individual measurements which were repeated 16 times for each search space for both Microsoft Windows™ and UNIX™. The last column shows the average of the standard deviation, given in percent of the average across different patterns.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Microsoft Windows™ Across measurements</th>
<th>UNIX™ Across measurements</th>
<th>All Experiments Across patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>bm4DNA</td>
<td>2.26</td>
<td>0.22</td>
<td>36.55</td>
</tr>
<tr>
<td>bmFast</td>
<td>2.11</td>
<td>0.28</td>
<td>23.26</td>
</tr>
<tr>
<td>bm4DNADS</td>
<td>2.34</td>
<td>0.27</td>
<td>30.85</td>
</tr>
<tr>
<td>bm4DNAHM</td>
<td>3.32</td>
<td>0.30</td>
<td>15.70</td>
</tr>
<tr>
<td>bm4DNAHMbs</td>
<td>3.38</td>
<td>0.33</td>
<td>14.85</td>
</tr>
<tr>
<td>bm4DNAIS</td>
<td>2.41</td>
<td>0.30</td>
<td>33.09</td>
</tr>
<tr>
<td>bMH</td>
<td>2.58</td>
<td>0.30</td>
<td>35.69</td>
</tr>
<tr>
<td>bmOrg</td>
<td>2.69</td>
<td>0.28</td>
<td>26.12</td>
</tr>
</tbody>
</table>

Table 3 clearly shows that the measurements made in the UNIX™ environment are significantly more homogeneous. Although the standard deviation and here the average of the standard deviations in percent of the average across the experiment within Microsoft Windows™ was almost an order of magnitude higher than within UNIX™, it does not invalidate the analysis of either since this error is almost an order of magnitude smaller than the spread across measuring different patterns. Across different patterns an additional error was introduced due to combining results from the two systems employed in the experiment. This additional error was however small compared to the difference across measured patterns. Clearly, patterns must have features which influence their search speed, one could be the occurrence of the suffix within the text being searched or more severely, within the pattern itself. Each time the suffix is encountered in the text a large number of computations need to be performed, potentially increasing the runtime. Depending on the distribution of the suffix in the text this may largely influence the runtime. This can be seen by the fact that standard deviation in algorithms using hashing is a fully 50% less than for the remaining algorithms. Across different search spaces and different sizes of $w$ it can be seen that with smaller $w$ the deviation increases for all algorithms that use infixes. The remaining deviation cannot be explained with the data gathered at this point and will be subject to further investigation. For a $w$ of 5 one would expect to find about 30000 occurrences in a text with $n$ equal to 30000000, purely by chance. This number obviously increases as $w$ decreases. The chance of the suffix also being an infix, which leads to smaller shifts, also increases accordingly. For a query
length of 1000 and a w of 5, one may expect to find the suffix in the query approximately one time in addition to the suffix itself. Both these events are detrimental to searching and will increase the overall runtime of the algorithms.

The results of Experiment 2 were plotted showing the time (milliseconds) necessary to process one megabyte of text against different number of expected hits (Figure 2). Here the number of forced hits is rather low compared to the first experiment, which makes the results appear as almost horizontal lines. The standard deviation is not shown since it is stable across the different points measured and can thus be deducted from Table 3.

![Figure 2: Processing speed in milliseconds per megabyte (y-axis) of text for a selected number of query lengths (10 upper left, 50 upper right, 100 lower left, 1000 lower right) in regards to a the number of hits (x-axis); compare for Figure 1. Note, that a larger number of algorithms are compared here and that less matches have been forced, with respect to Figure 1.](image)

While being slowest at a query length of 10, algorithms employing partial hashing rapidly become competitive in the region of pattern length 50, outperforming all other algorithms with increasing pattern length. The opposite is true for algorithms only employing sentinels. These are competitive for small pattern lengths, but are quickly outperformed by most other algorithms when greater pattern lengths are present.

In order to better assess the relative performance of the algorithms the fast BM algorithm was taken as a standard and the relative performance was plotted (Figure 3). It can be seen that bm4DNADS performs best for small query lengths and is not significantly slower than fast BM up to a query length of around 60. At that point the bm4DNAHMbs algorithm outperforms all others across the remaining experimental space. The bm4DNAHM algorithm performs this transition slightly later around a query length of 100 and across the whole experiment performs less effectively than the bm4DNAHMbs algorithm. This was expected since using the suffix shift and an additional sentinel were envisioned to speed up bm4DNAHMbs. Although for a small pattern lengths bm4DNA is slightly faster than bmOrg it consistently performs less well than bmFast. This is also true for bmOrg and BMH, whose design was clearly not optimal for searches in small alphabets. This can be explained by the...
missing b2 shift table for BMH, and for bmOrg by the absence of a skip loop, which would appear to accelerate the search over the whole search space. Since bmOrg seems to be converging against bmFast with increasing pattern length, the advantage of incorporating the skip loop seems to be diminishing on the same terms. Finally, bm4DNA performs worse than bmFast although it uses even three times unrolling of the skip loop since the skips that are possible are mere increments by one. For all other algorithms the skips are potentially larger. If compared to BMH the gain in performance due to using of sentinels can be seen.

![Figure 3: Relative performance of the algorithms in regard to the fast version of the Boyer-Moore algorithm. Vertical axis shows times faster (positive) and times slower than the fast BM algorithm. Horizontal axis is logarithmic and shows the pattern length.](https://doi.org/10.7287/peerj.preprints.1758v1)

Overall, it can be deduced from Figure 3 that with increasing pattern length the algorithms that perform significantly better are those employing transformation of the alphabet by hashing windows of the text. It is also clear that the overhead needs to be amortized first but general searches in nucleotide sequences involve long rather than short patterns.

**Conclusions**

A number of experiments were conducted which define the input spaces in which the algorithms work efficiently. These were compared and it was shown that using two algorithms from the set created in this study suffices to cover most of this space. Practical considerations lead to the conclusion that even though there are outliers for some input spaces these can be safely ignored. Thus choosing in-between bm4DNAHMbs (query length >= 50) and bm4DNADS (query length < 50) depending on the input will greatly accelerate exact pattern matching in nucleotide sequences. The algorithms are provided as a JAVA™ library and can thus be directly used on for example web servers providing DNA search facilities.
References


Joint Genome Institute.