The Basic Boyer-Moore Algorithm

The following definition of the Boyer-Moore algorithm omits a skip loop because it is widely ignored in practice. The basic algorithm, as shown below, will be referred to as **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.

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(α) and is preprocessed in time O(*m* + α).

```
foreach c in ∑ do
      let d1[c] = m
end foreach
for i from 0 to m do
      let d1[p[i]] = m - i
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*).

```
i = m, j = m + 1
fbm[i] = j
do while i > 0
      do while j <= m and p[i-1] != p[j-1]
            if d2[j] = 0 then
                  d2[j] = j - i
            end if
            j=fbm[j]
      end while
      i--
      j--
      fbm[i] = j
end while
j = fbm[0];
for i:0 to m do
      if d2[i] = 0 then
            d2[i] = j
      end if
      if i = j then
            j = fbm[j]
      end if
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.

```
let tp = 0   //tp: text pointer
do while tp < n - m
      let j = m
      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 +  d2(0)
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.

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(α) and can be constructed in O(α) time as a deep copy of the d1 table.

```
10     Let d0 = d1
20     Let d0[p[m]] = 2 * n
```

Search Phase

The skip loop in the fast BM algorithm has been proposed since most time is spend in sliding the pattern along *t* due to immediate mismatches. The skip loop can be implemented as an addition to the basic algorithm.

```
 10    let tp = 0    //tp: text pointer
 21    do while tp < n
 22          let tp = tp + d0[t[tp]]
 23    end while
 24    if tp < 2 * n then
 25          break
 26    end if
 27    let tp = tp - 2 * n - 1
 28    do while tp < n - m
 30          let j =  m - 1
 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 +  d2(0)
100          end if
110          let tp = tp + max(d1(tp+j),d2(j))
120    end while
```

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.

```
 10    foreach c in ∑ 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
 70    Let d0 = d1
 80   Let d0[p[m]] = 0
 90   for i from 0 to m do
100         let d1[p[i]] = m - i
110   end for
```

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. 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.

```
 10   let tp = 0   //tp: text pointer
 20   do while tp < n - m
 21         k = d0[t[tp]]
 22         do while k != 0
 22               let tp = tp + (k = d0[t[tp]])
 23               let tp = tp + (k = d0[t[tp]])
 24               let tp = tp + (k = d0[t[tp]])
 25         end while
 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 +  d2(0)
100         end if
110         let tp = tp + max(d1(tp+j),d2(j))
120   end while
```

Boyer-Moore-HM

This algorithm, abbreviated **bm4DNAHM**, uses a hash table (shift table) to store windows of size *w* from the pattern. Here, only subsets of the pattern are used and the size of the word *w* is calculated from *m* and α as described by Wu and Manber. 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* * *w*) where *w* << *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. 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

```
shift(key) {
        let res = d1[key]
        if res = notFound then
                return m - w
        end if
        return 0
}
```

Hash function

```
hash(p) {
        let hashVal = 0
        for j:0 to w do
                let hashVal = (hashVal * 128 + p[j]);
        end for
        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*.

```
append p to t
roreach infix of size w in p, excluding the suffix (from left to right)
        let key = hash(infix)
        let s = shift(key)
        let d1[key] = s
end foreach
let key = hash(suffix)
if key not in d1 then
        let suffixShift = m-w
end if
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.

```
let tp = 0   //tp: text pointer
do while tp < n - m
        key = hash(getWindow(tp,w))
        k = shift(key)
        do while k != 0
                let tp += (k = d1[hash(getWindow(tp,w)))]
                let tp += (k = d1[hash(getWindow(tp,w)))]
                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
 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 +  d2(0)
100         end if
110         let tp += d1[key]
120   end while
```

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]
 10   append p to t
 20   roreach 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   end foreach
```

```
let key = hash(suffix)
if key not in d1 then
      let suffixShift = m - w
else
      let suffixShift = d1[key]
end if
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 d2 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 d1 however this was beyond the scope of this study. The complexity of the algorithm remains unaffected by these changes.

```
let tp = 0   //tp: text pointer
do while tp < n - m
      key = hash(getWindow(tp,w))
      k = shift(key)
      do while k != 0
            let tp += (k = d1[hash(getWindow(tp,w)))]
            let tp += (k = d1[hash(getWindow(tp,w)))]
            let tp += (k = d1[hash(getWindow(tp,w)))]
      End while
      If(t[p - m) != sentinel)
            let tp += suffixShift
            continue
      end if
      If tp > n then
            break
      End if
      let j = m
      do while j>0 and t[tp+j] = p[j]
            j--
      end while
      if j <= 0 then
            reportMatch(tp+1)
            let tp = tp +  d2(0)
      end if
      let tp += suffixShift
end while
```

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. 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 α is 128 since the ASCII character set was used for simplicity. In this specific case, however, α 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

```
  1    Append p to t
  2    sentinel0 = p[0]
  3    sentinel1 = p[1]
//Until desired number of sentinels has been established
 10    foreach c in ∑ 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
 70    Let d0 = d1
 80    Let d0[p[m]] = 0
 90    for i from 0 to m do
100          let d1[p[i]] = m - i
110    end for
```

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).

```
 10    let tp = 0   //tp: text pointer
 20    do while tp < n – m
 21          k = d0[t[tp]]
 22          do while k != 0
 22                let tp = tp + (k = d0[t[tp]])
 23                let tp = tp + (k = d0[t[tp]])
 24                let tp = tp + (k = d0[t[tp]])
 25          end while
 26          if t[tp - m] != sentinel0 then ,
 27                let tp++
 28                contiune
 29          end if
 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 +  d2(0)
100          end if
110          tp++
120    end while
```

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(α + *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

```
   Append p to t
   sentinel0 = p[0]
   sentinel1 = p[1]
//Until desired number of sentinels has been established
   foreach c in ∑ do
          let d1[c] = m
   end foreach
   for i from 0 to m do
          let d1[p[i]] = m - i
   end for
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.

```
let tp = 0   //tp: text pointer
do while tp < n
       let tp = tp + d0[t[tp]]
end while
if tp < 2 * n then
       break
end if
if t[tp - m] != sentinel0 then ,
       let tp += d1[t[tp]] * -1
       contiune
end if
do while tp < n - m
       let j = m
       do while j>0 and t[tp+j] = p[j]
              j--
       end while
       if j <= 0 then
              reportMatch(tp+1)
              let tp = tp +  d2(0)
       end if
       let tp += d1[t[tp]] * -1
end while
```

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 α 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 α + *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 uneffected.
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.

```
  1    Append p to t
  2    sentinel0 = p[0]
  3    sentinel1 = p[1]
//Until desired number of sentinels has been established
 10    foreach c in ∑ 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
 70 Let d1[p[m]] = d1[p[m]] * -1
```

Afterwards the d0 table is created and the suffix is set to zero as in BMH.

```
 10    foreach c in ∑ 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
 70    Let d0 = d1
 80    Let d0[p[m]] = 0
 90    for i from 0 to m do
100          let d1[p[i]] = m - i
110    end for
```

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.

```
10     let tp = 0   //tp: text pointer
 20    do while tp < n - m
 21          k = d0[t[tp]]
 22          do while k != 0
 22                let tp = tp + (k = d0[t[tp]])
 23                let tp = tp + (k = d0[t[tp]])
 24                let tp = tp + (k = d0[t[tp]])
 25          end while
 26          if t[tp - m] != sentinel0 then ,
 27                let tp += d1[p[m]]
 28                contiune
 29          end if

 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 +  d2(0)
100          end if
110          let tp += d1[p[m]]
120    end while
```

Since all measured data points would overload Online Supplement Table 1, only a representative selection of results is shown.

Table 1: 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.

Input Space			**Fastest Algorithm**				**Prevalent Algorithm**			
Query Length	**Hits**	**w**	**Algorithm**	**Highest Speed**	**Average Speed**	**# of Wins**	**Algorithm**	**Highest Speed**	**Average Speed**	**# of Wins**
10	1	2	4DNAIS	1.61	2.78	19	4DNADS	2.68	3.77	27
10	10	2	4DNAIS	1.62	2.84	21	4DNADS	2.73	3.84	26
10	100	2	4DNAIS	1.62	2.81	19	4DNADS	2.72	3.71	27
10	500	2	4DNAIS	1.61	2.76	19	4DNADS	2.73	3.81	27
10	1000	2	4DNAIS	1.62	2.90	18	4DNADS	2.73	3.72	26
50	1	3	4DNADS	1.11	2.40	8	Fast	1.94	2.51	31
50	10	3	4DNA	1.32	2.00	8	Fast	1.90	2.36	29
50	100	3	4DNA	1.31	1.98	10	Fast	1.92	2.37	28
50	500	3	4DNA	1.33	1.95	11	Fast	1.94	2.37	30
50	1000	3	4DNA	1.34	1.88	10	Fast	1.96	2.38	31
70	1	3	4DNAIS	1.24	1.82	12	Fast	1.93	2.34	41
70	1	3	4DNAIS	1.06	1.47	9	4DNAHMbs	1.52	1.87	55
70	10	3	4DNAIS	1.06	1.55	8	4DNAHMbs	1.52	1.87	54
70	100	3	4DNAIS	1.06	1.51	8	4DNAHMbs	1.52	1.89	53
70	500	3	4DNAIS	1.07	1.51	8	4DNAHMbs	1.53	1.89	53
70	1000	3	4DNAIS	1.08	1.45	8	4DNAHMbs	1.55	1.90	53
100	1	3	4DNADS	1.15	1.15	1	4DNAHMbs	1.49	1.73	62
100	10	3	4DNAIS	1.08	1.37	2	4DNAHMbs	1.46	1.65	66
100	100	3	4DNAIS	1.08	1.26	2	4DNAHMbs	1.47	1.66	68
100	500	3	4DNA	1.09	1.32	2	4DNAHMbs	1.48	1.66	68
100	1000	3	4DNAIS	1.09	1.42	3	4DNAHMbs	1.48	1.67	68
500	1	4	4DNAHMbs	0.71	1.14	73	4DNAHMbs	0.71	1.14	73
500	10	4	4DNAHMbs	0.69	0.82	73	4DNAHMbs	0.69	0.82	73
500	100	4	4DNAHMbs	0.70	0.83	73	4DNAHMbs	0.70	0.83	73
500	500	4	4DNAHMbs	0.73	0.87	73	4DNAHMbs	0.73	0.87	73
500	1000	4	4DNAHMbs	0.78	0.91	73	4DNAHMbs	0.78	0.91	73
1000	1	5	4DNAHMbs	0.36	0.43	73	4DNAHMbs	0.36	0.43	73
1000	10	5	4DNAHMbs	0.37	0.40	73	4DNAHMbs	0.37	0.40	73
1000	100	5	4DNAHMbs	0.38	0.42	73	4DNAHMbs	0.38	0.42	73
1000	500	5	4DNAHMbs	0.45	0.49	73	4DNAHMbs	0.45	0.49	73
1000	1000	5	4DNAHMbs	0.53	0.56	73	4DNAHMbs	0.53	0.56	73

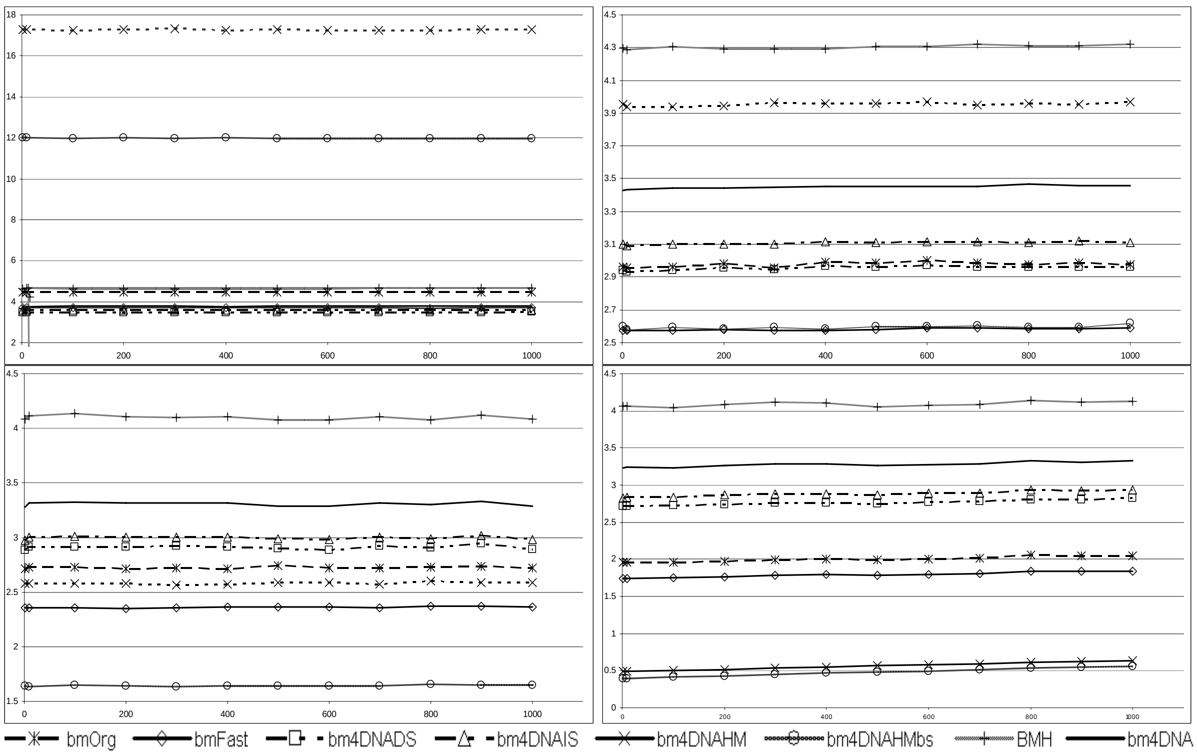

Figure 1: Processing speed in milliseconds per megabyte (y-axis) of text for a selected number of query lengths (10 upper left, 50 upper right, 100 lower left, 1000 lower right) in regards to a the number of hits (x-axis); compare for Figure 1. Note, that a larger number of algorithms are compared here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