Boyer–Moore string-search algorithm

Anders Søberg <andersob@ifi.uio.no>
Martin Mihle Nygaard <martimn@ifi.uio.no>

IN4120 2022 Science Fair

Boyer–Moore string-search [1]

- Proposed by Boyer and Moore in 1977.
- Standard benchmark for string-search
- E.g used by several `grep` implementations

Programming Techniques

G. Manacher, S.L. Graham Editors

A Fast String Searching Algorithm

Robert S. Boyer
Stanford Research Institute

J Strother Moore
Xerox Palo Alto Research Center

An algorithm is presented that searches for the location, "*i*," of the first occurrence of a character string, "*pat*," in another string, "*string*." During the search operation, the characters of *pat* are matched starting with the last character of *pat*. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched. Thus the algorithm has the unusual property that, in most cases, not all of the first *i* characters of *string* are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of *pat*. For a random English pattern of length 5, the algorithm will typically inspect *i*/4 characters of *string* before finding a match at *i*. Furthermore, the algorithm has been implemented so that (on the average) fewer than *i* + *patlen* machine instructions are executed. These conclusions are supported with empirical evidence and a theoretical analysis of the average behavior of the algorithm. The worst case behavior of the algorithm is linear in *i* + *patlen*, assuming the availability of array space for tables linear in *patlen* plus the size of the alphabet.

Key Words and Phrases: bibliographic search, computational complexity, information retrieval, linear time bound, pattern matching, text editing

CR Categories: 3.74, 4.40, 5.25

Authors' present addresses: R.S. Boyer, Computer Science Laboratory, Stanford Research Institute, Menlo Park, CA 94025. This work was partially supported by ONR Contract N00014-75-C-0816; JS. Moore was in the Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA 94304, when this work was done. His current address is Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

1. Introduction

Suppose that *pat* is a string of length *patlen* and we wish to find the position *i* of the leftmost character in the first occurrence of *pat* in some string *string*:

```
pat:          AT-THAT
string:   ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
i:                                  ↑
```

The obvious search algorithm considers each character position of *string* and determines whether the successive *patlen* characters of *string* starting at that position match the successive *patlen* characters of *pat*. Knuth, Morris, and Pratt [4] have observed that this algorithm is quadratic. That is, in the worst case, the number of comparisons is on the order of $i * patlen$.[1]

Knuth, Morris, and Pratt have described a linear search algorithm which preprocesses *pat* in time linear in *patlen* and then searches *string* in time linear in $i + patlen$. In particular, their algorithm inspects each of the first $i + patlen - 1$ characters of *string* precisely once.

We now present a search algorithm which is usually "sublinear": It may not inspect each of the first $i + patlen - 1$ characters of *string*. By "usually sublinear" we mean that the expected value of the number of inspected characters in *string* is $c * (i + patlen)$, where $c < 1$ and gets smaller as *patlen* increases. There are patterns and strings for which worse behavior is exhibited. However, Knuth, in [5], has shown that the algorithm is linear even in the worst case.

The actual number of characters inspected depends on statistical properties of the characters in *pat* and *string*. However, since the number of characters inspected on the average decreases as *patlen* increases, our algorithm actually speeds up on longer patterns.

Furthermore, the algorithm is sublinear in another sense: It has been implemented so that on the average it requires the execution of fewer than $i + patlen$ machine instructions per search.

The organization of this paper is as follows: In the next two sections we give an informal description of the algorithm and show an example of how it works. We then define the algorithm precisely and discuss its efficient implementation. After this discussion we present the results of a thorough test of a particular machine code implementation of our algorithm. We compare these results to similar results for the Knuth, Morris, and Pratt algorithm and the simple search algorithm. Following this empirical evidence is a theoretical analysis which accurately predicts the performance measured. Next we describe some situations in which it may not be advantageous to use our algorithm. We conclude with a discussion of the history of our algorithm.

[1] The quadratic nature of this algorithm appears when initial substrings of *pat* occur often in *string*. Because this is a relatively rare phenomenon in string searches over English text, this simple algorithm is *practically* linear in $i + patlen$ and therefore acceptable for most applications.

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

`E K S E M P E L`

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
```

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
- E M P - - - -
```

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
- E M P - - - -
- - E M P - - -
```

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
- E M P - - - -
- - E M P - - -
- - - E M P - -
```

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
- E M P - - - -
- - E M P - - -
- - - E M P - -
- - - - E M P -
```

Proposition: Naïve string-search

Brute-force all alignments of a pattern P in text T.

Example

$T =$ "EKSEMPEL" and $P =$ "EMP" gives $n - m + 1 = 6$ alignments to try (where n and m is the length of T and P)

```
E K S E M P E L
E M P - - - - -
- E M P - - - -
- - E M P - - -
- - - E M P - -
- - - - E M P -
- - - - - E M P
```

Improvement

Skip as many alignments as possible.

Improvement

Skip as many alignments as possible.

Idea

- Compare the last aligned characters in P to T, and shift along T based on result.

Improvement

Skip as many alignments as possible.

Idea

- Compare the last aligned characters in P to T, and shift along T based on result.
- Length of shift based on characters and suffixes of P.

Improvement

Skip as many alignments as possible.

Idea

- Compare the last aligned characters in P to T, and shift along T based on result.
- Length of shift based on characters and suffixes of P.
- Do some preprocessing of P.

Shift rules

The bad character rule ($delta_1$)

Shift rules

The bad character rule ($delta_1$)

If aligned character $\in P$:

- Shift P so that next occurrence of the character is aligned.

```
E K S E M P E L
S E M P - - - -
- - S E M P - -
```

Shift rules

The bad character rule (*delta*$_1$)

If aligned character $\in P$:

- Shift P so that next occurrence of the character is aligned.

```
E K S E M P E L
S E M P - - - -
- - S E M P - -
```

Else, aligned character $\notin P$:

- Shift P by it's own length, m.

```
E K S E M P E L
E M P - - - - -
- - - E M P - -
```

Shift rules

The good suffix rule (*delta*$_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)

Shift rules

The good suffix rule ($delta_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)

Example

```
            ... FLÅKLYPA-GRAND-PRIX-SUFFIX-SUFFIX ...
Case 1:                   FIX-SUFFIX
```

Shift rules

The good suffix rule ($delta_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)

Example

```
        ... FLÅKLYPA-GRAND-PRIX-SUFFIX-SUFFIX ...
Case 1:                FIX-SUFFIX
                            FIX-SUFFIX
```

Shift rules

The good suffix rule (*delta*$_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)
- Else if, a prefix of P matches a suffix of t, shift to it. (Case 2)

Example

```
         ... FLÅKLYPA-GRAND-PRIX-SUFFIX-SUFFIX ...
Case 1:                FIX-SUFFIX
Case 2:                           FIX-SUFFIX
```

Shift rules

The good suffix rule ($delta_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)
- Else if, a prefix of P matches a suffix of t, shift to it. (Case 2)

Example

```
        ... FLÅKLYPA-GRAND-PRIX-SUFFIX-SUFFIX ...
Case 1:                FIX-SUFFIX
Case 2:                       FIX-SUFFIX
                                     FIX-SUFFIX
```

Shift rules

The good suffix rule ($delta_2$)

Suppose for an alignment, S and P shares a suffix t.

- If a copy of t exists in P, shift to it. (Case 1)
- Else if, a prefix of P matches a suffix of t, shift to it. (Case 2)
- Else, shift P past t.

Example

```
         ... FLÅKLYPA-GRAND-PRIX-SUFFIX-SUFFIX ...
Case 1:                FIX-SUFFIX
Case 2:                         FIX-SUFFIX
                                         FIX-SUFFIX
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "`KEBABABA`" and $P =$ "`BABA`", without and with Galil rule.

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

```
      ↓
K E B A B A B A
B A B A - - - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓
K E B A B A B A
B A B A - - - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "`KEBABABA`" and $P =$ "`BABA`", without and with Galil rule.

```
  ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
  ↓ ↓ ↓   ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

```
  ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

```
        ↓
  ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

		↓	↓				
	↓	↓	↓	↓	↓		
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

```
    ↓ ↓
  ↓ ↓ ↓ ↓ ↓   ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

```
    ↓ ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓   ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
      ↓ ↓ ↓ ↓
    ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
```

11 comps.

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

		↓	↓	↓	↓		
	↓	↓	↓	↓	↓	↓	↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

11 comps.

			↓				
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓ ↓ ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
```

11 comps.

```
    ↓ ↓
K E B A B A B A
B A B A - - - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

$$
\begin{array}{cccccccc}
 & & \downarrow & \downarrow & \downarrow & \downarrow & & \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\mathtt{K} & \mathtt{E} & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} \\
\mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} & - & - & - & - \\
- & - & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} & - & - \\
- & - & - & - & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A}
\end{array}
$$

11 comps.

$$
\begin{array}{cccccccc}
 & \downarrow & \downarrow & \downarrow & & & & \\
\mathtt{K} & \mathtt{E} & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} \\
\mathtt{B} & \mathtt{A} & \mathtt{B} & \mathtt{A} & - & - & - & -
\end{array}
$$

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓ ↓ ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
   11 comps.
```

```
  ↓ ↓ ↓   ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓ ↓ ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
   11 comps.
```

```
  ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "`KEBABABA`" and $P =$ "`BABA`", without and with Galil rule.

		↓	↓	↓	↓		
	↓	↓	↓	↓	↓	↓	↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

11 comps.

			↓				
	↓	↓	↓	↓	↓		
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

T = "KEBABABA" and P = "BABA", without and with Galil rule.

```
    ↓ ↓ ↓ ↓
  ↓ ↓ ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
- - - - B A B A
   11 comps.
```

```
    ↓ ↓
  ↓ ↓ ↓ ↓ ↓
K E B A B A B A
B A B A - - - -
- - B A B A - -
```

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

		↓	↓	↓	↓		
	↓	↓	↓	↓	↓	↓	↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

11 comps.

		↓	↓				
	↓	↓	↓	↓	↓		↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

Shift rules

The Galil rule [2]

- Proposed by Galil in 1979.
- Improvement by making the comparisons faster.
- Speeds up multi-matching, with *periodic* patterns.
- Improves worst cases significantly.

Example

$T =$ "KEBABABA" and $P =$ "BABA", without and with Galil rule.

		↓	↓	↓	↓		
	↓	↓	↓	↓	↓	↓	↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

11 comps.

		↓	↓				
	↓	↓	↓	↓	↓	↓	↓
K	E	B	A	B	A	B	A
B	A	B	A	-	-	-	-
-	-	B	A	B	A	-	-
-	-	-	-	B	A	B	A

9 comps.

Performance

- Best case performance is $O(n/m)$

Performance

- Best case performance is $O(n/m)$
  - When you can skip m elements at every step

Performance

- Best case performance is $O(n/m)$
  - When you can skip m elements at every step
- Worst case – two cases

Performance

- Best case performance is $O(n/m)$
  - When you can skip m elements at every step
- Worst case – two cases
  - If the pattern does not appear in the text, the worst case is $O(m + n)$

Performance

- Best case performance is $O(n/m)$
  - When you can skip m elements at every step
- Worst case – two cases
  - If the pattern does not appear in the text, the worst case is $O(m + n)$
  - Proven in "Fast Pattern Matching in Strings"

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
- - A A A - - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
- - A A A - - -
- - - A A A - -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
- - A A A - - -
- - - A A A - -
- - - - A A A -
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
- - A A A - - -
- - - A A A - -
- - - - A A A -
- - - - - A A A
```

Performance

- Worst case for the original algorithm when the pattern does appear in the text is $O(mn)$.
- This can be seen by considering the case where the pattern and text both contain the same letter repeatedly:
- With Galil rule, linear in general

```
A A A A A A A A
A A A - - - - -
- A A A - - - -
- - A A A - - -
- - - A A A - -
- - - - A A A -
- - - - - A A A
```

Performance

- Preprocessing:
  - Varies depending on implementation.
  - Usually $O(m)$ or $O(km)$, where k is the size of the alphabet
  - Space complexity $O(k)$

EXTRA: Boyer–Moore–Horspool [3, 5]

A practical simplification: just use the 'bad character rule'.

- Worst case is $O(nm)$, but average case is $O(n)$
- Trivial to implement.

EXTRA: Boyer–Moore–Horspool [3, 5]

A practical simplification: just use the 'bad character rule'.

- Worst case is $O(nm)$, but average case is $O(n)$
- Trivial to implement.
  - See implementation

References

[1] Robert S. Boyer and J. Strother Moore. "A Fast String Searching Algorithm". In: *Commun. ACM* 20.10 (Oct. 1977), pp. 762–772. ISSN: 0001-0782. DOI: 10.1145/359842.359859. URL: https://doi.org/10.1145/359842.359859.

[2] Zvi Galil. "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm". In: *Commun. ACM* 22.9 (Sept. 1979), pp. 505–508. ISSN: 0001-0782. DOI: 10.1145/359146.359148. URL: https://doi.org/10.1145/359146.359148.

[3] R. Nigel Horspool. "Practical fast searching in strings". In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506. DOI: 10.1002/spe.4380100608. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380100608.

[4] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. "Fast Pattern Matching in Strings". In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. URL: https://doi.org/10.1137/0206024.

[5] Wikipedia. *Boyer–Moore–Horspool algorithm — Wikipedia, The Free Encyclopedia*. Nov. 7, 2022. URL: https://en.wikipedia.org/w/index.php?title=Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm.