Boyer-Moore matching

Goldsmiths Computing

November 1, 2018

Motivation

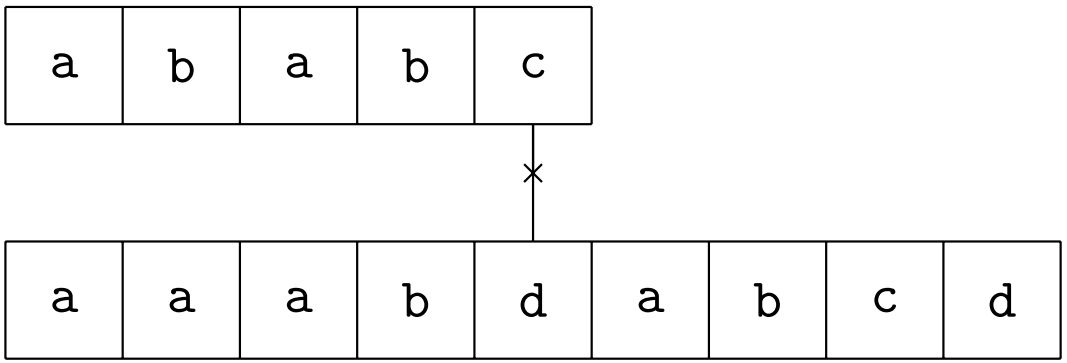
- deterministically $\Theta(m + n)$ string matching
- can achieve $\Theta(n/m)$ for matching phase in the best case

The bad character heuristic

- previously: use the *fact* that a mismatch has occurred to save work;
- now: use the specific character in the *text* that doesn't match (the "bad character") to save work.
  - check characters backwards from the end of the pattern for maximum effect

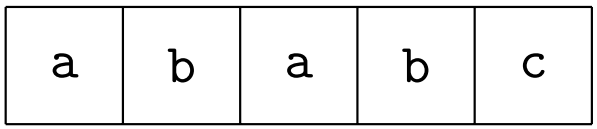
Diagram

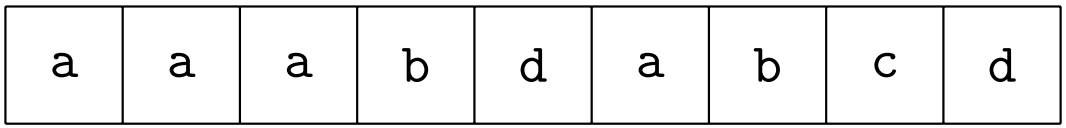
- bad character not in pattern:



Diagram

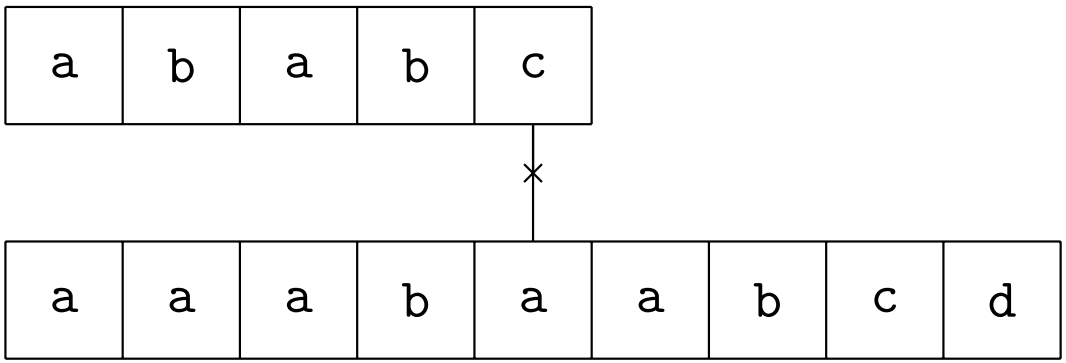
- bad character not in pattern:





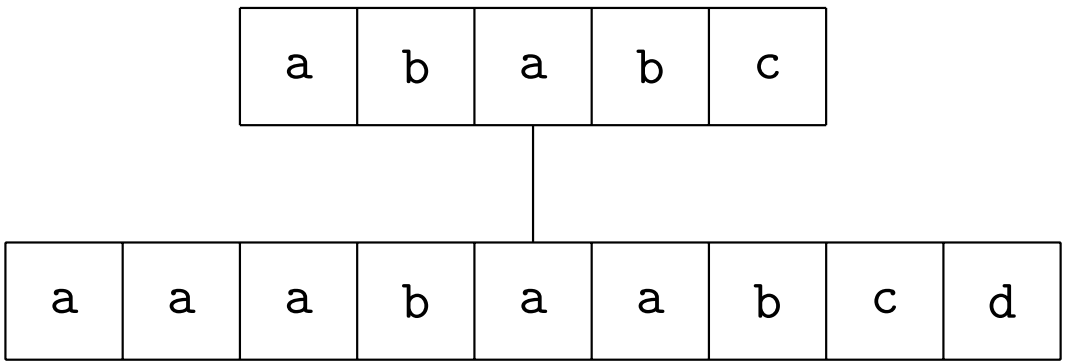
Diagram

- bad character in pattern:



Diagram

- bad character in pattern:



Boyer-Moore-Horspool

```
function BMHmatch(T,P)
    n ← length(T); m ← length(P)
    λ ← computeBadCharacter(P)
    s ← 0
    while s ≤ n - m do
        j ← m - 1
        while j ≥ 0 ∧ P[j] = T[s+j] do
            j ← j - 1
        end while
        if j = -1 then
            return s
        else
            s ← s + max(1,j-λ[T[s+j]])
        end if
    end while
    return false
end function
```

Boyer-Moore-Horspool: compute bad character

```
function computeBadCharacter(P)
    m ← length(P)
    λ ← new Table(-1)
    for 0 ≤ j < m do
        λ[P[j]] ← j
    end for
    return λ
end function
```

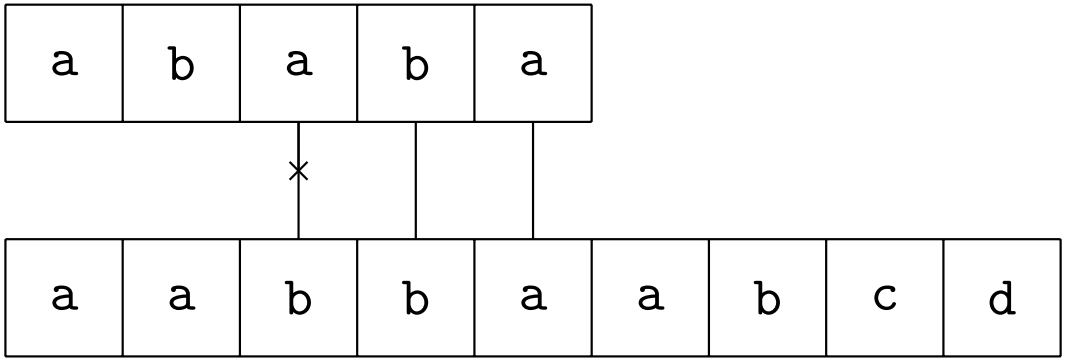
Work

1. Reading
    - Drozdek, section 13.1.3 “The Boyer-Moore Algorithm”

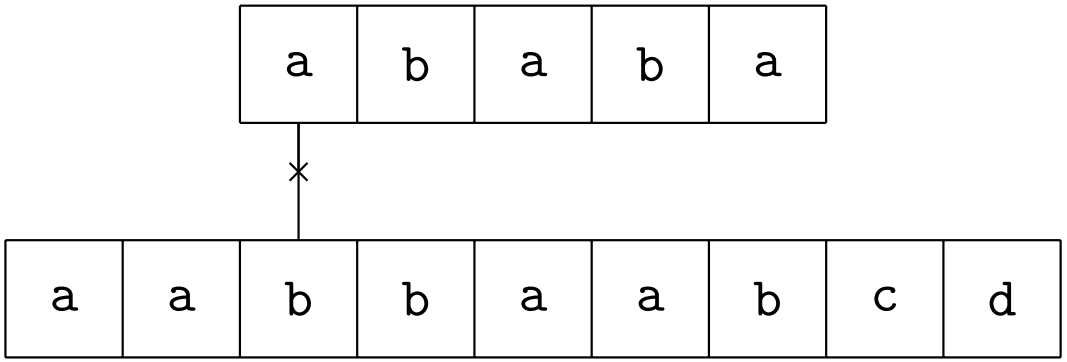
The good suffix heuristic

- bad character heuristic can recommend zero (or negative) shift
- not using information about any partial match
- good suffix: use knowledge that the suffix of the pattern matched must match any shifted pattern
  - find rightmost instance of good suffix...
  - ... not at the end of the pattern ...
  - (... preceded by a different character)

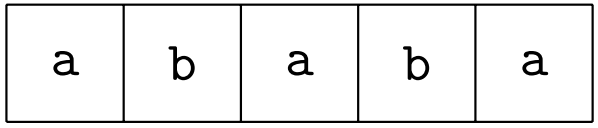
Diagram

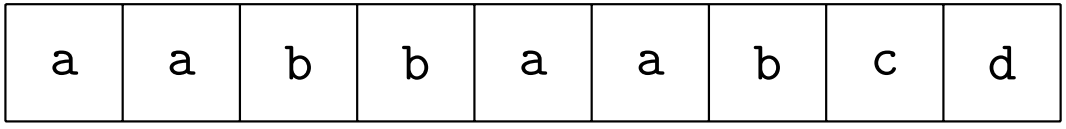


Diagram



Diagram





Boyer-Moore

```
function BMmatch(T,P)
    n ← length(T); m ← length(P)
    λ ← computeBadCharacter(P)
    γ ← computeGoodSuffix(P)
    s ← 0
    while s ≤ n - m do
        j ← m - 1
        while j ≥ 0 ∧ P[j] = T[s+j] do
            j ← j - 1
        end while
        if j = -1 then
            return s
        else
            s ← s + max(γ[j],j-λ[T[s+j]])
        end if
    end while
    return false
end function
```

Boyer-Moore: compute good suffix

```
function computeGoodSuffix(P)
    m ← length(P); π ← computePrefix(P)
    P' ← reverse(P); π' ← computePrefix(P')
    γ ← new Array(m)
    for 0 ≤ j < m do
        γ[j] ← m - π[m-1]
    end for
    for 0 ≤ l < m do
        j ← m - π'[l] - 1
        if γ[j] > l + 1 - π'[l] then
            γ[j] ← l + 1 - π'[l]
        end if
    end for
    return γ
end function
```

Galil Rule

If pattern is shifted to start at a text position after positions already checked:

- no need to recheck known-good matches

Complexity Analysis

space

γ, λ each $\Theta(m)$

- λ is $\Theta(\Sigma)$ if implemented using an array

time

Boyer-Moore-Horspool and Boyer-Moore

- preprocessing: $\Theta(m)$
- match:
  - worst case $\Theta(mn)$
  - best case $\Theta(n/m)$

With Galil Rule:

- worst case $\Theta(m + n)$
- best case $\Theta(n/m)$