Penerapan Algoritma KMP dan BM dalam Penyelesaian Permainan Cari Kata

Shafiq Irvansyah - 13522003
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): shafiq.irvansyah@gmail.com

***Abstract*—Permainan cari kata adalah jenis permainan teka-teki di mana pemain mencari kata-kata yang tersembunyi dalam sebuah grid huruf. Efisiensi dalam pencarian kata-kata ini merupakan faktor penting untuk meningkatkan pengalaman pemain. Algoritma pencocokan string seperti Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) dikenal dengan kemampuan mereka dalam mencari kecocokan string secara efisien. Makalah ini menjelaskan penerapan algoritma KMP dan BM dalam penyelesaian permainan cari kata. Dengan membandingkan kinerja kedua algoritma ini, kami berusaha menemukan metode yang lebih efektif dan efisien dalam memproses grid kata sehingga dapat membantu pemain menemukan kata-kata dengan lebih cepat. Studi ini menawarkan wawasan tentang penerapan algoritma pencocokan string yang tidak hanya teoretis tetapi juga praktis dalam pengembangan permainan berbasis teka-teki.*t.**

Keywords—KMP (Knuth-Morris-Pratt); Algoritma BM (Boyer-Moore); String Matching; Permainan Cari Kata;

I. Introduction (*Heading 1*)

Permainan cari kata adalah teka-teki yang menarik dan menantang di mana pemain harus menemukan satu atau lebih kata dalam kumpulan huruf yang disusun dalam grid. Kata-kata tersebut bisa muncul dalam berbagai arah: horizontal, vertikal, atau diagonal. Pemain bisa menikmati permainan ini secara individu atau bersama teman dan keluarga, yang menjadikannya sumber hiburan yang populer di berbagai kalangan dan usia. Dalam konteks kompetitif, pemenang ditentukan berdasarkan jumlah kata yang berhasil ditemukan. Ukuran *grid* karakter serta tingkat kesulitan diatur oleh setiap pemain.

Gambar 1. Permainan Cari Kata (play.google.com)

Dalam permainan seperti ini, efisiensi dalam mencari dan mengidentifikasi kata-kata tersembunyi sangat penting. Penggunaan algoritma yang tepat dapat mempercepat proses pencarian dan meningkatkan kualitas pengalaman bermain. Algoritma Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) adalah dua metode yang terkenal dalam bidang pencocokan string yang telah diterapkan dalam berbagai aplikasi pemrosesan teks. Algoritma KMP memungkinkan pencarian pola tanpa kembali ke belakang, sementara BM mengoptimalkan pergeseran pola, membuat keduanya menjadi kandidat yang ideal untuk digunakan dalam permainan cari kata. Makalah ini akan menjelajahi dan membandingkan efektivitas dari kedua algoritma ini dalam mengatasi tantangan yang ditawarkan oleh permainan cari kata, dengan tujuan untuk menentukan pendekatan yang paling efisien dalam konteks yang ditentukan.

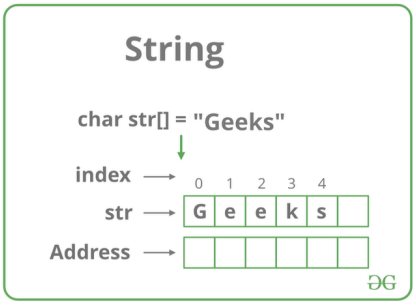
II. Landasan Teori

A. Permainan Cari Kata

Permainan cari kata melibatkan grid huruf di mana pemain harus menemukan kata-kata yang tersembunyi. Kata-kata ini dapat muncul dalam berbagai orientasi: horizontal, vertikal, atau diagonal. Kompleksitas permainan ini sering kali diatur oleh ukuran grid dan kepadatan kata.

B. String

String adalah tipe data yang digunakan dalam bahasa pemrograman untuk menyimpan dan mengubah kalimat dalam teks. Sebuah string bisa mengandung huruf, angka, dan simbol. Dalam konteks pemrograman, string biasanya diperlakukan sebagai array karakter. Sebagai contoh, kata "Geeks" bisa diinisialisasi dalam bahasa pemrograman sebagai string.



Gambar 2. Contoh String (www.geeksforgeeks.org)

Salah satu konsep dalam string adalah prefix dan suffix. Misalkan ada sebuah string *S* dengan panjang *i*. Prefix dari *S* adalah substring dari *S* yaitu *S*[0..*j*], di mana *j* adalah indeks yang lebih kecil atau sama dengan *i*. Sementara itu, suffix dari *S* adalah substring dari *S* yaitu *S*[k..*i* - 1], di mana *k* adalah nilai yang dapat berkisar dari 0 hingga *i* - 1. Nilai *j* dan *k* bisa berbeda, dan penting untuk dicatat bahwa k tidak akan melebihi i - 1.

Prefiks:

G	e	e

Sufiks:

e	k	s

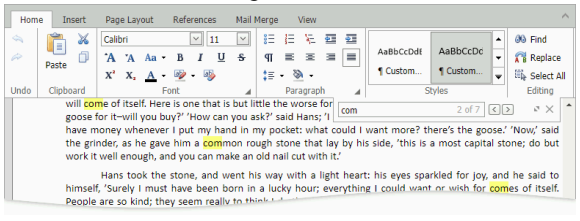
C. *String Matching*

String matching adalah proses untuk memeriksa apakah suatu pola (*pattern*) terdapat dalam suatu teks. Proses *string matching* ini memiliki beberapa algoritma yang umum dipakai,yaitu Brute Force, KMP (Knuth-Morris-Prat), dan BM (Boyer-Moore). Secara umumnya, algoritma *string matching* memiliki komponen berikut:

- Teks (text):
  Sebuah string yang panjangnya n karakter
- Pola (*pattern*):
  Sebuah string dengan panjang m karakter (m < n) yang akan dicari di dalam teks

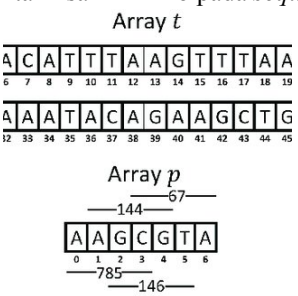
Aplikasi dari teknik *string matching* sangat beragam, mencakup berbagai sektor termasuk teknologi dan kesehatan. Berikut adalah beberapa contoh aplikasi dari *string matching* yang umum digunakan:

1. Pencarian kata/kalimat pada *text editor*



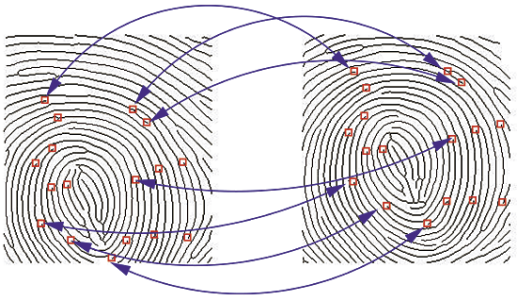
Gambar 3. Search pada Word (devexpress.github.io)

2. Pencarian Rantai Asam Amino pada *sequence* DNA



Gambar 4. DNA Sequence Matching (www.semanticscholar.org)

3. Pencocokan Citra Sidik Jari

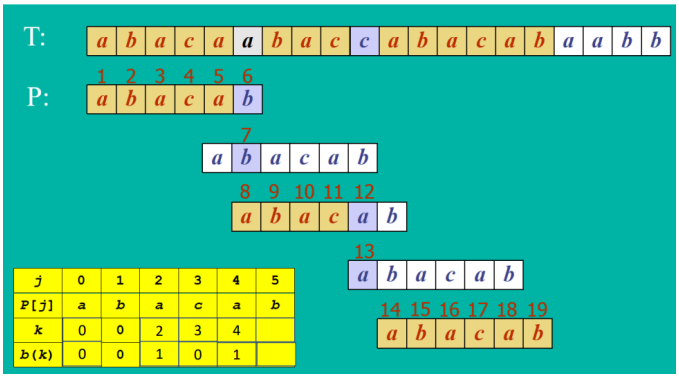


Gambar 5. Pencocokan Sidik Jari (www.researchgate.net)

D. *Algoritma KMP (Knuth-Morris-Pratt)*

Algoritma pencocokan KMP menggunakan sifat degradasi (pola memiliki sub-pola yang sama yang muncul lebih dari sekali dalam pola tersebut) dari pola dan meningkatkan kompleksitas kasus terburuk menjadi O(*n*+*m*) , di mana *n* adalah panjang teks dan *m* adalah panjang pola.

Konsep utama dari algoritma KMP adalah ketika terjadi ketidakcocokan setelah beberapa karakter cocok, sebenarnya sudah sikenali beberapa karakter di bagian teks berikutnya. Informasi ini dimanfaatkan untuk menghindari pencocokan ulang karakter-karakter yang sudah dipastikan akan cocok. Untuk lebih jelasnya dapat dilihat pada contoh berikut.



Gambar 6. Contoh Algoritma KMP (informatika.stei.itb.ac.id)

Sebelum proses *string matching* dimulai, akan dibuat *Border Function*-nya terlebih dahulu pada pola yang akan dicari. *Border Function* ini digunakan sebagai referensi ketika terjadi mismatch pada pola, sehingga proses pencocokan tidak perlu dimulai dari awal pola lagi.

Untuk menentukan *border function* pada sebuah pola, perlu diperiksa setiap indeks dari 0 hingga panjang pola dikurangi satu, misalnya saat memeriksa pada indeks *i*. Pemeriksaan dilakukan dengan mengambil jumlah huruf maksimal yang sama antara prefix dan suffix. Prefix diambil dari pola[0..*i*] dan suffix dari pola[1..i]. Sebagai contoh, akan digunakan pola "abacab". Misalnya, diperiksa pada indeks ke-3. Di sini nilai *k* adalah *j*-1. Akan dicek apakah ada prefix dari [0..*k*] yang sama dengan suffix dari [1..k]. Pertama, ditemukan prefix "a" dan suffix "a", yang merupakan kecocokan, sehingga disimpan nilai *k* sementara sebagai 1 (*j* - 1 = 2). Selanjutnya, dibandingkan prefix [0..1] dan suffix [*k*-1..*k*] yaitu "ab" dan "ba". Karena tidak cocok, maka hasil dari dua huruf ini tidak disimpan. Karena suffix sudah mencapai angka 1, maka pencocokan dihentikan, sehingga untuk indeks 3 dan k = 2, *border function* yang dihasilkan adalah 1.

Tabel 1. Pseudocode Pembuat Border Function

```
function computeBorder(pattern: string) -> array of int
    m = length of pattern
    b = new array of int with size m
    b[0] = 0
    j = 0
    i = 1

    while i < m do
        if pattern[j] == pattern[i] then
            j = j + 1
            b[i] = j
            i = i + 1
        else if j > 0 then
            j = b[j - 1]
        else
            b[i] = 0
            i = i + 1
        end if
    end while

    return b
end function
```

Setelah *border function* sudah terbentuk, pencocokan string diawali dari ujung kiri teks. Saat terjadi mismatch pada indeks tertentu j dalam pola, algoritma ini tidak memulai pencocokan dari awal lagi, melainkan menggeser indeks j menjadi nilai dari *BorderFunction*[j-1]. Jika mismatch terjadi di indeks awal pola (j = 0), maka pencocokan dilanjutkan dengan menggeser pola satu huruf ke kanan terhadap teks. Algoritma KMP memanfaatkan fungsi border ini untuk meningkatkan efisiensi dengan menghindari pencocokan ulang pada bagian pola yang sudah terbukti tidak cocok.

Tabel 2. Pseudocode KMP Keseluruhan

```
function kmpMatch(text: string, pattern: string) -> int
    n = length of text
    m = length of pattern
    b = computeBorder(pattern)
    i = 0
    j = 0

    while i < n do
        if pattern[j] == text[i] then
            if j == m - 1 then
                return i - m + 1 // match found
            end if
            i = i + 1
            j = j + 1
        else if j > 0 then
            j = b[j - 1]
        else
            i = i + 1
        end if
    end while

    return -1 // no match found
end function
```

E. Algoritma BM (Boyer-Moore)

Berbeda dengan algoritma KMP, algoritma Boyer-Moore memulai pencocokan dari akhir pola daripada awalnya. Hal ini memungkinkan beberapa optimasi yang meningkatkan efisiensi dengan meminimalisir jumlah perbandingan yang perlu dilakukan terhadap teks. Algoritma ini memiliki kompleksitas waktu rata-rata O(*n/m*), di mana *n* adalah panjang teks dan *m* adalah panjang pola.

Dalam implementasinya, algoritma ini menggunakan 2 teknik.

1. Teknik *Looking Glass*

   Pencocokan dimulai dari akhir pola, bukan awal. Ini memungkinkan algoritma untuk mengevaluasi potensi kecocokan dengan lebih cepat dan melompati bagian yang tidak perlu diperiksa lebih lanjut.

2. Teknik Character-Jump

   Ketika terjadi mismatch, algoritma akan menentukan seberapa jauh pola harus digeser berdasarkan informasi terakhir dari karakter yang tidak cocok. Ini dilakukan dengan mengevaluasi posisi terakhir dari karakter tersebut dalam pola.

Algoritma ini memproses pola terlebih dahulu untuk menentukan posisi terakhir setiap karakter dalam pola. Hal ini memungkinkan algoritma untuk menghindari perbandingan yang tidak perlu dengan melompat lebih jauh dalam teks ketika mismatch terjadi. Misalkan, diberikan pola "ASCII". Maka akan dibentuk *last occurrence table* sebagai berikut

Tabel 3 Contoh Last Occurrence

A	C	I	S
0	2	4	1

Tabel 4. Pseudocode Pembuat Last Occurrence Table

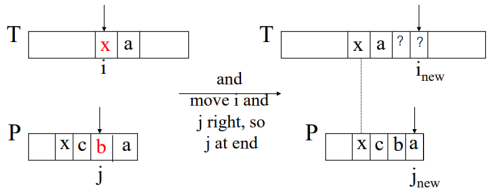
```
function buildLast(pattern: string) -> array of int
   // Initialize array for ASCII characters
   last = new array of int with size 128
   for i = 0 to 127 do
      last[i] = -1 // initialize all entries with -1
   end for

   // Process the pattern to update last occurrence of each
character
   for i = 0 to length of pattern - 1 do
      charIndex = ASCII value of pattern[i]
      last[charIndex] = i
   end for

   return last
end function
```

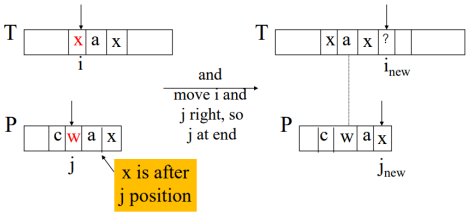
Dalam proses pencocokan, terdapat tiga kasus pencocokan yang mungkin terjadi saat terjadi mismatch antara teks dan pola.

Pertama, jika karakter yang tidak cocok terdapat di suatu tempat dalam pola, algoritma akan mencoba menggeser pola ke kanan untuk menyelaraskan kejadian terakhir karakter tersebut dengan posisi saat ini dalam teks.



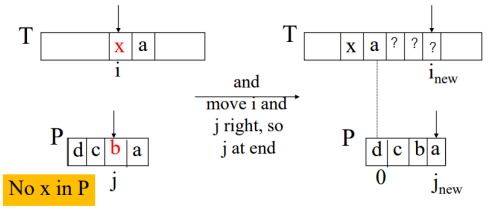
Gambar 7. BM Missmatch Case 1 (informatika.stei.itb.ac.id)

Kedua, jika penggeseran seperti di kasus pertama tidak memungkinkan, maka pola digeser ke kanan sebanyak satu karakter.



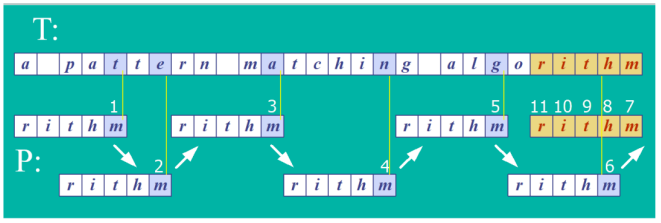
Gambar 8. BM Missmatch Case 2 (informatika.stei.itb.ac.id)

Ketiga, jika karakter yang tidak cocok tersebut tidak terdapat dalam pola, maka pola akan digeser ke posisi tepat setelah karakter yang tidak cocok tersebut dalam teks.



Gambar 9. BM Missmatch Case 3 (informatika.stei.itb.ac.id)

Untuk lebih jelas proses algoritma BM secara keseluruhan, dapat dilihat dari contoh sebagai berikut.



Gambar 10. Contoh String Matching BM (informatika.stei.itb.ac.id)

Pada contoh diatas proses pencocokan string dimulai dengan menyelaraskan pola 'rithm' dengan bagian akhir dari teks yang dicari. Pencocokan karakter diawali dari ujung kanan pola, bergerak ke kiri. Dalam contoh ini, setiap kali terjadi mismatch, algoritma menentukan berapa jauh pola harus digeser ke kanan menggunakan dua teknik: last occurrence dan good suffix.

Pada langkah pertama, ketika 'rithm' diuji cocokkan di posisi awal pada teks, terjadi mismatch pada karakter 't' dalam teks yang tidak cocok dengan 'm' dalam pola. Berdasarkan last occurrence table, algoritma mengidentifikasi bahwa 't' teridentifikasi pada pola, sehingga pola digeser supaya *last occurrence* 't' pada pola sejajar dengan 't' yang missmatch. Lalu terjadi lagi missmatch pada karakter 'a' di teks, namun karena 'a' tidak terdefinisi pada *last occurrence table* maka patterna akan digeser keseluruhannya hingga melwati indeks karakter yang missmatch tersebut. Proses ini diulangi hingga ditemukan sub-string yang match secara keseluruhan dengan karakter. Dari contoh tersebut, dapat dilihat algoritma BM melewati banyak tahapan pencocokan yang tidak diperlukan.

Tabel 5. Pseudocode BM Keseluruhan

```
function bmMatch(text: string, pattern: string) -> int
   last = buildLast(pattern)
   n = length of text
   m = length of pattern
   i = m - 1

   if i > n - 1 then
      return -1 // no match if pattern is longer than text

   j = m - 1
   while i <= n - 1 do
      if pattern[j] == text[i] then
         if j == 0 then
            return i // match found
```

```
        else
            i = i - 1
            j = j - 1
        end if
    else
        lo = last[ASCII value of text[i]] // last occurrence
        i = i + m - min(j, 1 + lo)
        j = m - 1
    end if
end while

return -1 // no match found
end function
```

III. IMPLEMENTASI

Implementasi penyelesaian permainan cari kata berikut menggunakan bahasa pemrograman python. Program ini dapat memproses solusi dari permainan cari kata dalam bahasa inggris dengan menggunakan KMP atau BM. Solusi yang akan diberikan adalah katanya beserta dengan indeks kata tersebut pada grid dan arah teksnya pada grid (diagonal/horizontal/vertikal).

Program ini memulai operasinya dengan meminta pengguna memasukkan nama file yang mengandung matriks berisi grid karakter. Kata-kata yang akan dicari oleh program ini berasal dari sebuah kamus yang telah disiapkan sebelumnya dalam file yang bernama "words.txt". Pengguna memiliki opsi untuk memodifikasi kamus ini jika dirasa kamus tersebut belum memadai.

Setelah memasukkan nama file, pengguna selanjutnya akan diminta untuk memilih metode pencarian yang diinginkan. Program menyediakan dua pilihan metode pencarian: algoritma Knuth-Morris-Pratt (KMP) dan algoritma Boyer-Moore (BM). Setelah metode dipilih, program akan memproses pencarian solusi, menampilkan hasilnya, dan mencatat durasi waktu yang diperlukan untuk proses tersebut. Setelah satu sesi pencarian selesai, program akan kembali ke kondisi awal dan siap untuk menerima input file matriks baru dari pengguna

F. String Matching BM ()

1. Last Occurrence Table

```python
def LastOccurrence(pattern):
    last = [-1] * 128  # Assuming ASCII character set
    for index in range(len(pattern)):
        last[ord(pattern[index])] = index
    return last
```

Gambar 11. Kode Pencari Last Occurrence
(arsip pribadi)

2. BM Search

```python
def bmSearch(pattern, text):
    last = LastOccurrence(pattern)
    n = len(text)
    m = len(pattern)
    i = m - 1  # Start at the end of the pattern,
               # aligned with corresponding position in text

    while i < n:
        j = m - 1  # Start comparison at the end of the pattern
        while j >= 0 and pattern[j] == text[i]:
            j -= 1
            i -= 1
        if j < 0:  # If all characters matched
            return i + 1  # Match found, return starting index
        else:
            i += m - min(j, 1 + last[ord(text[i])])  # Shift the pattern

    return -1  # No match found
```

Gambar 12. Kode BM Search
(arsip pribadi)

G. String Matching KMP ()

1) Border Function Table

```python
def compute_border(pattern):
    m = len(pattern)
    border = [0] * m
    j = 0

    # i starts from 1 since border[0] is always 0
    for i in range(1, m):
        while (j > 0 and pattern[i] != pattern[j]):
            j = border[j - 1]

        if pattern[i] == pattern[j]:
            j += 1
            border[i] = j
        else:
            border[i] = 0

    return border
```

Gambar 13. Kode Pencari Border Function
(arsip pribadi)

3) KMP Search

```python
def kmpSearch(pattern, text):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0  # Immediate match if pattern is empty

    border = compute_border(pattern)
    j = 0  # index for pattern

    # i is the index for text
    for i in range(n):
        while (j > 0 and text[i] != pattern[j]):
            j = border[j - 1]

        if text[i] == pattern[j]:
            j += 1

        if j == m:
            return i - m + 1  # Match found at index i - m + 1

        # If we've reached the end of text and no match was found
        if i == n - 1 and j != m:
            return -1

    return -1  # Pattern not found
```

Gambar 14. Kode KMP Search
(arsip pribadi)

H. Main Program

```python
while(True):
    words = readWordFile("words.txt")
    row_answers = []
    col_answers = []
    diag_answers = []

    board_file = input("Masukkan nama file papan kata yang akan di cari jawabannya : ")
    board = readMatrixFile(board_file)
    while board == []:
        board_file = input("Masukkan nama file papan kata yang akan di cari jawabannya : ")
        board = readMatrixFile(board_file)

    mode = input("Masukkan mode pencarian (KMP/BM) : ")
    while mode not in ["KMP", "BM","kmp","bm"]:
        mode = input("Masukkan mode pencarian (KMP/BM) : ")

    #  Set timer
    start = time.time()

    # Searching horizontal
    rows = extract_rows(board)
    ...
    # Searching vertical
    cols = extract_columns(board)
    for i in range(len(cols)):
    ...

    print(GREEN  + "\nJawaban Pada sisi COLUMNS :" + RESET)
    for col in col_answers:
        print(col)

    # Searching diagonal
    diag = extract_diagonals(board)
    for i in range(len(diag)):
    ...

    print(GREEN + "\nJawaban Pada sisi DIAGONAL :" + RESET)
    for dia in diag_answers:
        print(dia)

    #  Set timer
    end = time.time()
    print(YELLOW + f"\nWaktu eksekusi program : {end - start} detik" +RESET)

```

Gambar 15. Kode Main Program
(arsip pribadi)

Dalam implementasi *main program*, proses dimulai dengan parsing kamus yang telah disiapkan pada file "words.txt". Selanjutnya, program melakukan parsing terhadap grid karakter dari permainan cari kata yang diberikan oleh pengguna. Program ini akan mengekstraksi semua string potensial dari grid tersebut, yang meliputi ekstraksi dari arah horizontal, vertikal, dan diagonal. Setelah ekstraksi, pencarian kata akan dilakukan menggunakan salah satu dari dua metode yang dipilih oleh pengguna: algoritma Knuth-Morris-Pratt (KMP) atau algoritma Boyer-Moore (BM).

Hasil pencarian akan ditampilkan secara berurutan, dimulai dari pencarian pada sisi horizontal, diikuti oleh vertikal, dan terakhir diagonal. Program ini juga menyertakan penampilan waktu yang diperlukan untuk seluruh proses pencarian, memberikan gambaran efisiensi metode yang dipilih.

IV. PENGUJIAN

Test pengujian teks dilakukan dengan menggunakan beberapa parameter yang berbeda, yaitu dari ukuran grid yang relatif kecil dan grid yang relatif besar, serta grid yang kepadatan katanya tinggi dan grid yang kepadatan katanya rendah.

Kualitas solusi yang dihasilkan sangat dipengaruhi oleh tingkat filterisasi dari kamus yang digunakan. Dalam implementasi ini, penulis menggunakan kamus yang filterisasinya masih belum optimal, yang berdampak pada solusi yang relatif banyak dengan kata-kata aneh.

I. Test Case Keseluruhan

Tabel 6. Test Case Keseluruhan

TC 1	TC2
S T A R S T A R T S A R E A S R T E A R S A R T S	N O T E B O O K Z Q X W E R T U M I C R O S C O P E I J K L M D F G H I J K L M N O P Q S A E L M N O P Q R S T U V W X T C T U V W X Y Z A B C D E F R H Y P H E N Y Z Q W E R T Y I I J K L M N O P Q R S T U V C A L C U L A T O R X Y Z A B A B C D E F G H I J K L M N O L C D M N O P Q R S T U V W X E E F G H I J K L M N O P Q R N D T U V W X Y Z A B C D E F D A Z Q W E R T Y U I O P A S A R K L M N O P Q R S T U V W R E F R I G E R A T O R B C D
TC 3	**TC 4**
S T A R S T A R T S A E E A S R R A A R S A T T S	S T A R S T A R S T A R S T T A R T S T A R T S T A R T A R E A S A R E A S A R E A R T E A R R T E A R R T E A S A R T S S A R T S S A R T T S T A R S T A R S T A R S A R T S T A R T S T A R T S R E A S A R E A S A R E A S T E A R R T E A R R T E A R S R T S S A R T S S A R T S T A R S T A R S T A R S T A S T A R T S T A R T S T A R A R E A S A R E A S A R E A R T E A R R T E A R R T E A S A R T S S A R T S S A R T

J. Ukuran Kecil dan Kepadatan Tinggi

1) Jawaban

Tabel 7. Solusi Test Case 1

Jawaban Pada sisi ROWS (21) : ['stars', [0, 0]] ['tarts', [1, 0]] ['areas', [2, 0]] ['star', [0, 0]] ['tars', [0, 1]] ['arts', [1, 1]] ['tart', [1, 0]] ['area', [2, 0]] ['tear', [3, 1]] ['arts', [4, 1]] ['ars', [0, 2]] ['tar', [0, 1]] ['art', [1, 1]] ['tar', [1, 0]] ['are', [2, 0]] ['eas', [2, 2]] ['rea', [2, 1]] ['ear', [3, 2]] ['tea', [3, 1]] ['art', [4, 1]] ['sar', [4, 0]]	Jawaban Pada sisi COLUMNS (10) : ['stars', [0, 0]] ['star', [0, 0]] ['tars', [1, 0]] ['tart', [0, 1]] ['ars', [2, 0]] ['tar', [1, 0]] ['art', [1, 1]] ['tar', [0, 1]] ['are', [0, 2]] ['ree', [1, 2]]	Jawaban Pada sisi DIAGONAL (6) ['tret', (1, 0)] ['ret', (2, 1)] ['eas', (2, 2)] ['sae', (0, 0)] ['tra', (0, 1)] ['ats', (0, 2)]

2) KMP

```
Waktu eksekusi program : 0.5794358253479004 detik
```

Gambar 16. Waktu KMP pada Test Case 1
(arsip pribadi)

3) BM

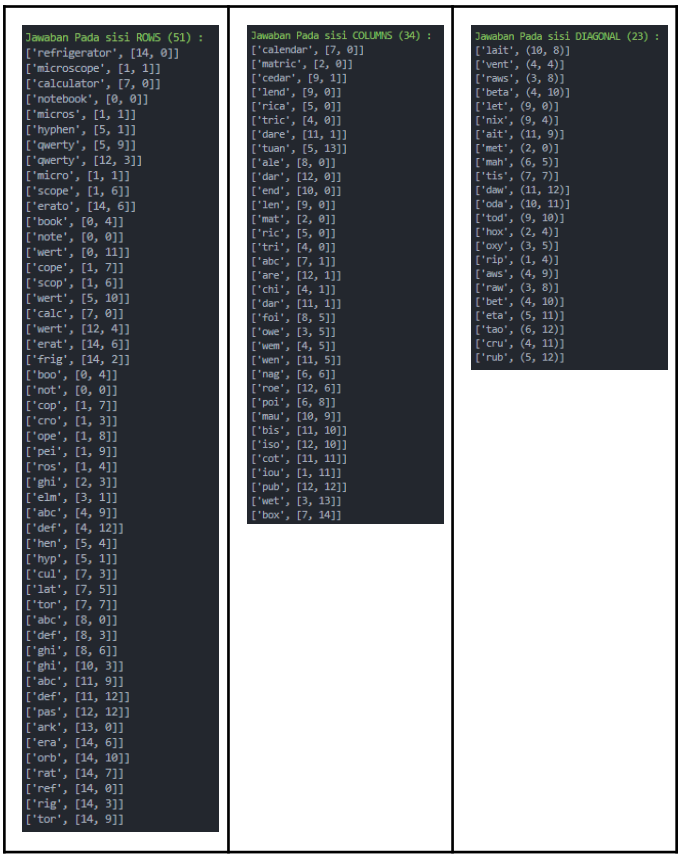
```
Waktu eksekusi program : 0.47487711906433105 detik
```

Gambar 17. Waktu BM pada Test Case 2
(arsip pribadi)

K. Ukuran Besar dan Kepadatan Rendah

1) Jawaban

Tabel 8. Solusi Test Case 2



2) KMP

```
Waktu eksekusi program : 23.3904070854187 detik
```

Gambar 18. Waktu KMP pada Test Case 2
(arsip pribadi)

3) BM

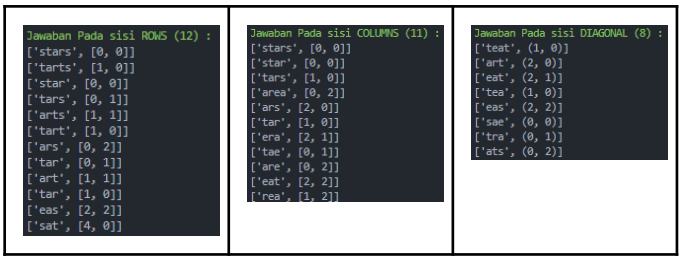
```
Waktu eksekusi program : 14.18241572380066 detik
```

Gambar 19. Waktu BM pada Test Case 2
(arsip pribadi)

L. Ukuran Kecil dan Kepadatan Rendah

1) Jawaban

Tabel 9. Solusi Test Case 3



2) KMP

```
Waktu eksekusi program : 0.6099469661712646 detik
```

Gambar 20. Waktu KMP pada Test Case 3
(arsip pribadi)

3) BM

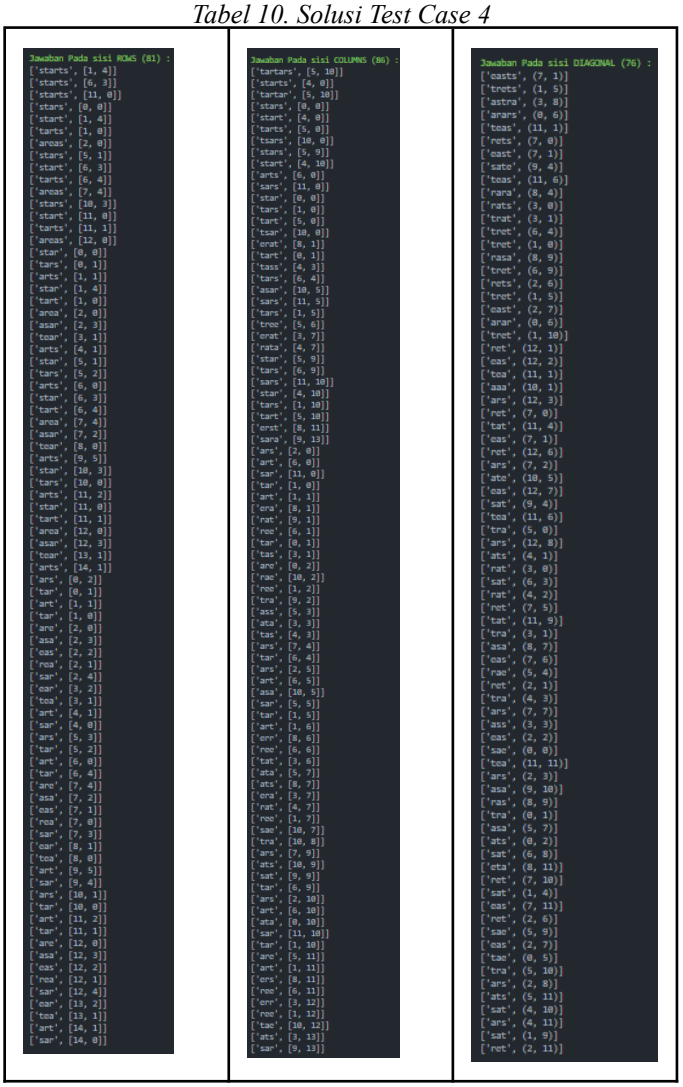
```
Waktu eksekusi program : 0.5190794467926025 detik
```

Gambar 21. Waktu BM pada Test Case 3
(arsip pribadi)

M. Ukuran Besar dan Kepadatan Tinggi

1) Jawaban

Tabel 10. Solusi Test Case 4



2) KMP

```
Waktu eksekusi program : 22.031745195388794 detik
```

Gambar 22. Waktu KMP pada Test Case 4
(arsip pribadi)

3) BM

```
Waktu eksekusi program : 14.10554027557373 detik
```

Gambar 23. Waktu BM pada Test Case 4
(arsip pribadi)

V. Kesimpulan

Secara keseluruhan, algoritma Boyer-Moore (BM) menunjukkan keunggulan dibandingkan dengan algoritma Knuth-Morris-Pratt (KMP) dalam mencari solusi permainan cari kata. Keunggulan ini dapat disebabkan oleh beberapa faktor yang mempengaruhi efisiensi dan kecepatan dalam pencarian kata.

Pertama, algoritma BM memanfaatkan heuristik *last occurrence* yang memungkinkan pencarian untuk melompati bagian teks yang besar tanpa melakukan pencocokan, apabila karakter yang sedang dianalisis tidak sesuai. Ini sangat efektif terutama ketika pola yang dicari tidak sering muncul dalam teks. Dibandingkan dengan KMP yang memproses setiap karakter secara berurutan, BM dengan cepat mengecualikan area non-potensial yang mempercepat proses pencarian secara signifikan.

Kedua, BM mengimplementasikan teknik *character jump*, yang tidak hanya mempertimbangkan karakter yang salah saat terjadi mismatch, tetapi juga segmen yang telah cocok sebelumnya. Hal ini memungkinkan algoritma untuk membuat lompatan yang lebih informatif, sering kali melewatkan area teks yang lebih luas daripada yang mungkin dilakukan oleh KMP.

Ketiga, pendekatan pencocokan dari akhir pola (teknik *looking glass*) yang digunakan BM menyebabkan mismatch terdeteksi lebih awal jika pola tidak cocok di akhir. Ini berlawanan dengan KMP yang memulai dari awal pola, sehingga BM sering kali mengurangi jumlah perbandingan yang perlu dilakukan dalam proses pencocokan.

Keempat, dalam konteks permainan cari kata di mana grid bisa sangat luas dan pola bisa relatif pendek, kecepatan ini menjadi sangat penting. Dalam kasus di mana teks mengandung banyak kata atau karakter yang sama, BM dapat lebih cepat menentukan tidak adanya kecocokan, yang membuatnya lebih efisien.

Kesimpulannya, algoritma Boyer-Moore, dengan teknik pencocokan string yang komplek, menyediakan solusi yang lebih optimal untuk pencarian cepat dalam permainan cari kata dibandingkan dengan KMP. Ini menjadikannya pilihan yang umum digunakan dalam pengembangan aplikasi yang menggunakan *pattern matching*.

Ucapan Terima Kasih

Penulis mengucapkan syukur kepada Allah SWT atas kemudahan dan keberhasilan dalam menuntaskan makalah ini tepat waktu. Ucapan terima kasih juga ditujukan kepada para dosen program Strategi Algoritma IF2211, khususnya kepada Dr. Ir. Rinaldi Munir, M.T., yang telah mengajar di kelas K1 dan memberikan pengetahuan yang memungkinkan penulis mengerjakan tugas ini. Penulis juga berterima kasih kepada keluarga dan teman-teman yang memberikan motivasi dan dukungan selama proses penyusunan makalah. Meski menyadari adanya kekurangan dalam makalah ini, penulis berharap bahwa karya ini dapat bermanfaat bagi masyarakat luas.

Link Youtube

https://youtu.be/7zmWedWB__4

Referensi

[1] R. Munir, 'IF2211 Strategi Algoritma - Semester II Tahun 2023/2024', https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf
Diakses pada 10 Juni 2024.

[2] geeksforgeeks.org, 'What is String – Definition & Meaning', https://www.geeksforgeeks.org/what-is-string/
Diakses pada 10 Juni 2024.

[3] geeksforgeeks.org, 'Boyer Moore Algorithm for Pattern Searching', https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
Diakses pada 11 Juni 2024.

[4] geeksforgeeks.org, 'KMP Algorithm for Pattern Searching', https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
Diakses pada 11 Juni 2024.

[5] gwicks.net, 'Dictionaries and Word Lists', http://www.gwicks.net/dictionaries.htm
Diakses pada 11 Juni 2024.

[6] geeksforgeeks.org, 'Applications of string matching algorithms', https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/
Diakses pada 11 Juni 2024.

[7] Repository penulis, 'word-search-solver', shafiqIrv/word-search-solver (github.com)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Shafiq Irvansyah, 13522003