

PERBANDINGAN ALGORITMA *BRUTE-FORCE* DAN *KNUTH-MORRIS-PRATT* PADA PERMASALAHAN STRING MATCHING

Fakhri Akbar Pratama
Ida Bagus Dwi Satria K.

1103130153
1301140297

GroupID : 4

e-mail: ahriabar@gmail.com , dwisatriakusuma4@gmail.com

ABSTRAK

String matching adalah salah satu kasus yang sering dibahas dalam *computer science*. Permasalahan ini akan semakin kompleks seiring dengan bertambahnya ukuran data dan semakin kompleksnya solusi yang diinginkan. Dalam makalah ini akan dibandingkan penyelesaian menggunakan *brute-force* dan KMP, dan yang mana yang lebih baik.

Kata kunci: *String matching, brute-force, KMP*

1. PENDAHULUAN

Dalam *computer science*, *string matching problem* adalah suatu permasalahan untuk mencari apakah sebuah atau beberapa *string* (yang juga disebut *patterns* (pola)) dapat ditemukan di dalam sebuah *string* yang lebih besar atau di dalam sebuah teks^[1]. Misalkan Σ adalah alfabet (merupakan set finite). Contoh paling sederhana dari permasalahan *string matching* adalah *pattern* x dicari dan ditentukan pada bagian mana saja *pattern* tersebut muncul.

Dalam makalah ini, akan dibeirkan dua masalah yang harus diselesaikan dengan dua algoritma yang berbeda, dan penyelesaian kedua masalah tersebut akan dibahas.

Masalah yang pertama, diberikan sebuah *source string* $S[0..n-1]$ dengan panjang n , yang terdiri dari simbol a dan b . Misal, diberikan *pattern string* $P[0..m-1]$ dengan panjang $m \ll n$, yang terdiri dari simbol a , b , dan $*$, yang menggambarkan *pattern* yang ditemukan di dalam *string* S . Simbol $*$ adalah simbol "wild card", yang cocok dengan setiap simbol, a maupun b . Selain simbol ini, simbol yang lain harus cocok dengan tepat. *Output* dari masalah ini adalah *list* M yang berisi posisi yang cocok, di mana posisi j di dalam S sedemikian sehingga *pattern* P cocok dengan *substring* $S[j..j+|P|-1]$. Sebagai contoh, jika $S = a b a b a b$ dan $P = a b *$, maka *output* M adalah $[0, 2]$. Tuliskan algoritma yang bersifat *naive* (*straightforward*) untuk menyelesaikan masalah. Algoritma tersebut harus jalan dalam waktu $O(nm)$.

Masalah yang kedua, kurang lebih sama seperti masalah yang pertama, namun diberikan simbol alfabet yang lebih besar. Secara spesifik, misal diberikan representasi untaian DNA sebagai *string* $D[0..n-1]$ dengan panjang n , terdiri dari simbol A , C , G , dan T ; dan diberikan *pattern string* $P[0..m-1]$ dengan panjang $m \ll n$, terdiri dari simbol A , C , G , T , dan $*$. Masalahnya sama, dan *output* yang diharapkan adalah *sorted list* M yang berisi posisi yang cocok, di mana posisi j di dalam D sedemikian sehingga *pattern* P cocok dengan *substring* $D[j..j+|P|-1]$. Sebagai contoh, jika $D = ACGACCAT$ dan $P = AC * A$, maka *output* M adalah $[0, 3]$. Tuliskan algoritma yang efisien untuk kasus kedua, dan algoritma tersebut harus berjalan dalam waktu $[O(n+m), O(n \lg m)]$.

2. METODE

Metode yang akan digunakan dalam makalah ini adalah algoritma *brute force* untuk kasus pertama, dan algoritma KMP untuk kasus kedua.

2.1 Algoritma *Brute-Force* untuk Kasus *String Matching*

Sebelum melangkah ke algoritma *brute-force*, ada baiknya kita memahami dulu bagaimana ide solusi dengan menggunakan algoritma *brute-force* untuk *string matching*.^{[2][3]}

Ide dari algoritma *brute-force* untuk kasus ini adalah meng-*generate* setiap kemungkinan solusi. Setiap kemungkinan solusi dalam kasus ini mencocokkan *pattern* *P* terhadap setiap *substring* berukuran *m* yang dibangun pada setiap iterasi.

```
BRUTEFORCE_SM(S,P)
M = []
for s = 0 to n - m
    valid = TRUE
    for j = 0 to m - 1
        if P[j] ≠ * or P[j] ≠ S[s+j]
            valid = FALSE
    if valid
        M.append(s) //masukkan s ke
        dalam list solusi M

return M
```

Algoritma 1. *Brute-force* untuk kasus *String Matching*

Dengan menggunakan algoritma tersebut, maka untuk contoh kasus pertama, jawabannya adalah sebagai berikut

S = a b a b b a b
P = a b *

a b a b b a b
↑ ↑ ↑
a b * BENAR

a b a b b a b
↑ ↑ ↑
a b * SALAH

a b a b b a b
↑ ↑ ↑
a b * BENAR

a b a b b a b
↑ ↑ ↑
a b * SALAH

a b a b b a b
↑ ↑ ↑
a b * SALAH

Dari hasil perhitungan algoritma tersebut, kita mendapatkan solusi adalah [0, 2]. Algoritma ini memiliki *running time* $O(nm)$. Sekilas dapat dilihat, *running time* ini didapatkan dari *for* di dalam *for*, di mana dilakukan perulangan terhadap ukuran *string* *S* dan di dalamnya terdapat perulangan terhadap ukuran *pattern* *P*.

2.2 Algoritma *Knuth-Morris-Pratt* untuk Kasus *String Matching*

Untuk kasus kedua, algoritma yang digunakan adalah algoritma *Knuth-Morris-Pratt* (KMP). Ide dari algoritma ini adalah melabeli beberapa indeks, sehingga pencarian akan menjadi lebih efisien. Secara sistematis, langkah-langkah yang dilakukan oleh algoritma KMP dalam *string matching* adalah ^[4] :

1. Mencocokkan *pattern* pada awal teks.
2. Dari kiri ke kanan, algoritma akan mencocokkan karakter per karakter *pattern* dengan karakter di teks yang bersesuaian sampai salah satu kondisi berikut dipenuhi :
 - a. Karakter di *pattern* dan di teks yang dibandingkan tidak cocok.
 - b. Semua karakter di *pattern* cocok. Kemudian algoritma akan memberitahukan penemuan di posisi ini.
3. Algoritma kemudian menggeser *pattern* berdasarkan tabel *next*, lalu mengulangi langkah 2 sampai *pattern* berada di ujung teks.

Berikut adalah *pseudo code* dari KMP :

```
procedure preKMP (
    input P : array[0..n-1] of char,
    input n : integer,
    input/output kmpNext : array[0..n]
    of integer)
```

Deklarasi :
i, j = integer

```

Algoritma :
  i := 0;
  j := kmpNext[0] := -1;

  while (i < n) {
    while (j > -1 and not (P[i]=P[j]))
or P[i]=P[j]='*')
      j:=kmpNext[j];
    i:= i+1;
    j:= j+1;
    if (P[i] = P[j])
      kmpNext[i] := kmpNext[j];
    else
      kmpNext[i] := j;
    endif
  endwhile

```

Algoritma 2. preKMP untuk kasus *pre-processing* menggunakan KMP

```

procedure KMPSearch(
  input m, n : integer,
  input P : array[0..n-1] of char,
  input T : array[0..m-1] of char,
  output ketemu : array[0..m-1] of
boolean
)

Deklarasi:
i, j,next: integer
kmpNext : array[0..n] of interger

Algoritme:
preKMP(n, P, kmpNext)
i:=0
while (i<= m-n) do
  j:=0
  while (j < n and T[i+j] = P[j])
do
    j:=j+1
  endwhile
  if(j >= n) then
    ketemu[i]:=true;
  endif
  next:= j - kmpNext[j]
  i:= i+next
endwhile

```

Algoritma 3. KMP untuk kasus *String Matching*

Kompleksitas dari algoritma adalah $O(n+m)$.

IV. KESIMPULAN

Jadi, berdasarkan hasil perhitungan menggunakan algoritma *brute-force* dan KMP, kami menyimpulkan bahwa untuk kasus *string matching* lebih baik menggunakan algoritma KMP dibanding *brute-force* dilihat dari kompleksitas atau *running time*-nya. *Running time* KMP adalah $O(n+m)$ dan untuk *brute-force* adalah $O(nm)$.

REFERENSI

- [1] https://en.wikipedia.org/wiki/String_searching_algorithm
- [2] MIT6_046JS15 – Problem Set 2.
- [3] Diktat Rinaldi – IF2251 Strategi Algoritmik
- [4] https://id.wikipedia.org/wiki/Algoritma_Knuth-Morris-Pratt