Implementasi Algoritma FJS dalam Pencarian Sekuens Gen Spesifik pada DNA Genom

Johanes Boas Badia 13517009
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
boas.marbun@yahoo.com, 13517009@std.stei.itb.ac.id

***Abstract*—FJS adalah salah satu algoritma pattern matching yang merupakan gabungan dari Boyer-Moore dan Knuth-Morris-Pratt. Penggunaan FJS yang dibahas kali ini adalah implementasinya dalam sekuens gen spesifik pada DNA Genom.**

***Keywords*—FJS, Boyer-Moore, Knuth-Morris-Pratt, String matching, sekuens gen, DNA Genom**

I. PENDAHULUAN

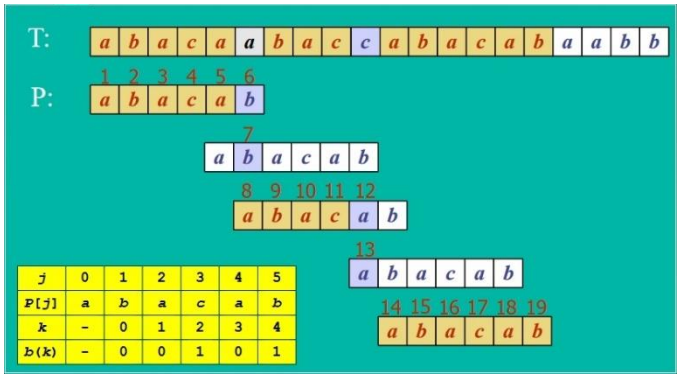
Pada Algoritma String Matching, ada dua algoritma yang sangat sering diperkenalkan dan digunakan di dunia *computer science*, yaitu KMP(Knuth-Morris-Pratt) algorithm, dan BM(Boyer-Moore) algorithm. Keduanya merupakan algoritma pencarian string, dan biasa digunakan sampai saat ini.Kedua algoritma melakukan pencarian *pattern*, namun dengan pendekatan yang berbeda. Singkatnya, Algoritma KMP melakukan pencarian suatu *pattern* dari depan, sedangkan BM dari belakang. Tentu masing-masing algoritma punya kekurangan dan kelebihannya tersendiri. Namun dalam makalah ini, akan dibahas lebih dalam tentang FJS, yaitu algoritma yang "menggabungkan" BM dan KMP, sehingga kekurangan dan kelebihan masing-masing algoritma dapat tertutupi.

Untuk implementasi *pattern matching*, sangat banyak implementasinya di dunia nyata. Salah satunya yang akan dibahas pada makalah ini, yaitu pencarian sekuens protein pada DNA genom. Pencarian ini mempunyai banyak tujuan, yaitu salah satunya untuk mengetes apakah ada kelainan pada rantai DNA, mencari suatu protein dalam DNA, dll. Pada makalah ini akan dibahas penggunaan algoritma FJS pada pencarian sekuens protein dalam DNA Genom.

II. ALGORITMA KMP(KNUTH MORRIS PRATT)

Algoritma KMP adalah algoritma yang mencari sebuah *pattern* dalam sebuah string dari kiri ke kanan[1]. Diciptakan oleh Donald E. Knuth. Pencarian ini mirip seperti pencarian menggunakan *brute force* daru kiri ke kanan, namum ada perbedaan di bagian pergeseran patternnya.

Pada *brute force*, penggeseran karakter dilakukan setiap ditemukan karakter salah, dan akan dimajukan satu huruf, dan akan dilakukan sampai string sudah mencapai akhir. Namun pada algoritma KMP, tidak dilakukan penggeeseran secara satu tahap, melainkan menggunakan boundary function yang berfungsi untuk mengetahui harus berapa karakter. Namun pada makalah ini, tidak akan dibahas secara detail tentang algoritma KMP.



Contoh algoritma KMP[1]

Tabel kunging merupakan boundary function. Seperti yang sudah dijelaskan sebelumnya, fungsi ini dapat mengetahui berapa huruf yang sama antara suffix dan prefix. Tabel ini dapat dicari dengan menggunakan fungsi tertentu.

Dilihat dari contoh di atas, kita memulai dengan *pattern* P dan string T. dari indeks 1-5, kecocokan ditemukan antara pattern dan string.Namun pada karakter ke-6, ditemukan perbedaan. Karena ditemukan perbedaan pada indeks ke-6, yang merupakan indeks 5 (pada tabel, j) pada string, dapat dilihat dari tabel, untuk j = 5, b(k) atau boundary functionnya bernilai 1. Artinya, ada prefix dan suffix sama sejauh 1 karakter, sehingga karakter pertama bisa dilewatkanuntuk pencarian selanjutnya. Setelah tau boundary function bernilai 1, maka pattern akan mulai dicocokkan mulai dari tempat ditemukannya perbedaan(karakter ke-6) – 1(boundary function). Sehingga pencarian dimulai kembali dari posisi 5, dengan pattern indeks-0. Dan pada indeks ini tidak perlu lagi dilakukan pengecekan apakah karakter sama, karena sudah pasti sama, ditentukan oleh boundary function.

Lalu iterasi dilanjutkan, dan mismatch langsung ketemu, sehingga pattern diulang kembali menjadi indeks 0. Hal yang sama dilakukan, sampai pattern ketemu di string atau pattern tidak ditemukan pada string.

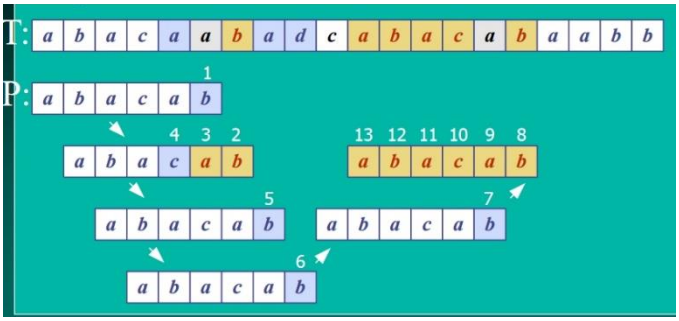
```
#include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix
suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {


            if (len != 0) {
                len = lps[len - 1];

            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Implementasi program KMP dalam bahasa C++ [3]

III. ALGORITMA BM(BOYER-MOORE)

Algoritma BM adalah algoritma yang mencari sebuat pattern dari sebuah string, namun berbeda dengan KMP, pencarian Boyer-Moore bukan dari depan ke belakang, melainkan berbalikan dengan cara biasanya, yaitu dari belakang ke depan.[1]

Pada algoritma BM, terdapat tiga kasus yang menetukan kasus saat pencarian dari belakang ke depan. Jika pada KMP ada boundary function, BM punya Last Occurence Function, yang berguna untuk menentukan langkah selanjutnya. Biasanya, Last Occurence Function ini disimpan dalam sebuah array, agar mempermudah pengaksesan.



x	a	b	c	d
L(x)	4	5	3	–1

Contoh Algoritma BM[1]

Pada contoh diatas, dapat dilihat terjadi missmatch pada karakter dengan indeks 5 pada pattern. Mismatch terjadi saat string ada pada indeks 5, dan pada indeks tersebut terdapat karakter a. Karena itu, dilihat dari tabel L(x), huruf a memiliki nilai 4. Maka, akan dilakukan pergeseran sehingga karakter yang mengalami missmatch akan dipasangkan dengan a yang ada pada indeks ke 5 dari pattern.

Setelah itu, dilakukan kembali pencarian dari kanan ke kiri. Ditemukan missmatch kembali pada c(pada string yang karakter yang ditunjuk adalah a). Setelah itu, karena pada tabel, huruf a berindeks 4, maka huruf indeks huruf a sudah melewati indkes huruf c. Maka dari itu, diberlakukan kasus kedua, yaitu memajukan pattern sebanyak satu langkah.

Langkah langkah selanjutnya, mirip dengan langkah-langkah sebelumnya.Sampa pada pattern yang berangka 6 pada gambar. Jika searching sampai ke tahap ini, maka dua langkah diatas sudah tidak bisa dilakukan lagi. Satu langkah yang tersisa, yaitu melewati pencocokan, dan menyamakan posisi indeks 0 pattern pada indeks +1 setelah mismatch. Langkah kembali dilakukan sampai pattern ditemukan atau string sudah tidak bisa diperiksa lagi.

```
#include <bits/stdc++.h>
using namespace std;
# define NO_OF_CHARS 256

// The preprocessing function for Boyer Moore's
// bad character heuristic
void badCharHeuristic( string str, int size,
                       int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}

/* A pattern searching function that uses Bad
Character Heuristic of Boyer Moore Algorithm */
void search( string txt, string pat)
{
    int m = pat.size();
    int n = txt.size();

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling
    the preprocessing function badCharHeuristic()
    for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with
                // respect to text
    while(s <= (n - m))
    {
        int j = m - 1;

        /* Keep reducing index j of pattern while
        characters of pattern and text are
        matching at this shift s */
        while(j >= 0 && pat[j] == txt[s + j])
            j--;

        /* If the pattern is present at current
        shift, then index j will become -1 after
        the above loop */
        if (j < 0)
        {
            cout << "pattern occurs at shift = " <<  s <<
endl;
              s += (s + m < n)? m-badchar[txt[s + m]] : 1;

        }

        else
            s += max(1, j - badchar[txt[s + j]]);
    }
}
```

Algoritma BM[4]

IV. ALGORITMA FJS

Algoritma FJS adalah algoritma pattern matching buatan Frantisek Franek, Christopher G. Jennings and W. F. Smyth.[2] Walaupun tidak dikatakan secara eksplisit, namun menurut saya FJS adalah singkatan dari Franek, Jennings, dan Smyth, diambil dari masing-masing pembuat algoritma ini. FJS merupakan gabungan dari dua algoritma yang telah dibahas diatas.

Kira-kira algoritmanya adalah, Pertama, lakukan pencarian pada string, mulai dari kanan ke kiri, seperti pada BMS. Pelompatan yang dilakukan akan mengikuti aturan BM. Apabila karakter indeks pattern terakhir yang dicocokkan ditemukan kecocokan, maka pencarain akan dilakukan secara KMP. Apabila ada string yang ditemukan, maka akan dilakukan maju secara KMP. Apabila string tidak ditemukan, huruf pertama pada pattern akan dicocokan dengan indeks string + 1. Hal ini akan terus diulang sampai ditemukannya pattern atau string sudah mencapai indeks akhir, sehingga tidak dapat dibaca lagi.

```
    #include <stdio.h>
    #include <string.h>

    typedef unsigned char CTYPE;   // type of alphabet letters

    // For large alphabets, such as Unicode, see the Web page
above
    // for techniques to improve performance

    #define ALPHA      (256)       // alphabet size
    #define MAX_PATLEN (100)       // maximum pattern length

    int betap[ MAX_PATLEN+1 ];
    int Delta[ ALPHA ];

    void output( int pos ) {
        static int matches = 0;
        printf( "match %d found at position %d\n", ++matches,
pos );
    }

    void makebetap( const CTYPE* p, int m ) {
        int i = 0, j = betap[0] = -1;

        while( i < m ) {
            while( (j > -1) && (p[i] != p[j]) ) {
                j = betap[j];
            }
            if( p[++i] == p[++j] ) {
                betap[i] = betap[j];
            } else {
                betap[i] = j;
            }
        }
    }

    void makeDelta( const CTYPE* p, int m ) {
        int i;

        for( i = 0; i < ALPHA; ++i ) {
            Delta[i] = m + 1;
        }
        for( i = 0; i < m; ++i ) {
            Delta[ p[i] ] = m - i;
        }
    }

    void FJS( const CTYPE* p, int m, const CTYPE* x, int n )
{
        if( m < 1 ) return;
        makebetap( p, m );
        makeDelta( p, m );

        int i = 0, j = 0, mp = m-1, ip = mp;
        while( ip < n ) {
            if( j <= 0 ) {
                while( p[ mp ] != x[ ip ] ) {
                    ip += Delta[ x[ ip+1 ] ];
                    if( ip >= n ) return;
                }
                j = 0;
                i = ip - mp;
                while( (j < mp) && (x[i] == p[j]) ) {
                    ++i; ++j;
                }
```

```
            if( j == mp ) {
                output( i-mp );
                ++i; ++j;
            }
            if( j <= 0 ) {
                ++i;
            } else {
                j = betap[j];
            }
        } else {
            while( (j < m) && (x[i] == p[j]) ) {
                ++i; ++j;
            }
            if( j == m ) {
                output( i-m );
            }
            j = betap[j];
        }
        ip = i + mp - j;
    }
}
```

Algoritma FJS[5]

V. ALGORITMA FJS FJS DALAM PENCARIAN SEKUENS GEN SPESIFIK PADA DNA GENOM

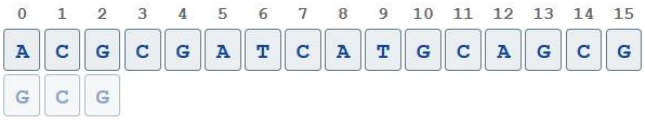
Sekarang, kita akan masuk ke inti pembahasan makalah ini. Sebelumnya, saya ingin menjelaskan sedikit tentang sekuens gen. Sekuens gen adalah bagian dari rantai DNA panjang yang ada di dalam tubuh manusia. Dari rantai DNA yang panjang ini, banyak informasi yang bisa kita dapat, seperti ada protein apa saja yang ada di dalam DNA, sampai mendeteksi suatu penyakit tertentu, dengan cara mendeteksi kelainan DNA.

Dengan hal-hal yang telah saya sebutkan diatas, saya harap program ini bisa berguna, terutama di perkembangan bidang bioinformatika tentunya. Pada bab ini, akan dibahas per setp bagaimana jalannya algoritma FJS dalam pencarian string.

Untuk kasus dibawah ini, anggap kita punya:

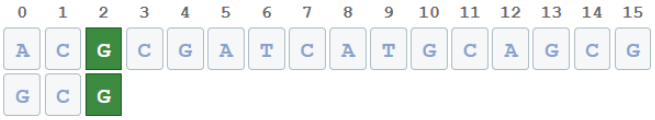
- string : “ACGCGATCATGCAGCG”
- pattern : “GCG

Dari pengamatan mata kosong, GCG ditemukan pada indeks 1-3 dan indeks 13-15 pada string.



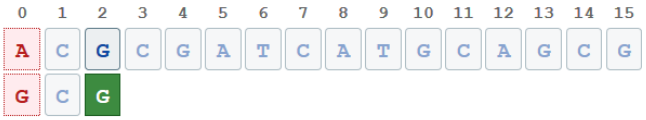
Proses JFS[6]

Ini adalah kondisi awal yang sudah dijelaskan tadi. Sekarang, kita akan melakukan pencarian, Mulai dari indeks ke-2. Huruf G pada pattern dicocokan dengan huruf G pada string, sehingga timbul match.



Proses JFS[6]

Karena Match Ditemukan, maka akan dilakukan algoritma KMP, yaitu mencocokan string dari kiri ke kanan.

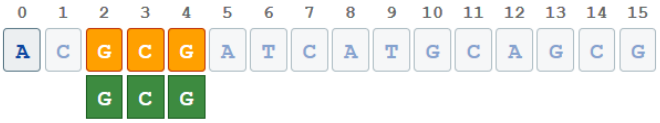


Proses JFS[6]

Setelah dicek dengan KMP, string tidak ditemukan. Karena tidak ditemukan, pencarian deilanjutkan dengan menggeser karakter pertama pattern ke indeks 1(indeks yang digunakan adalah angka yang terletak diatas string).

Proses JFS[6]

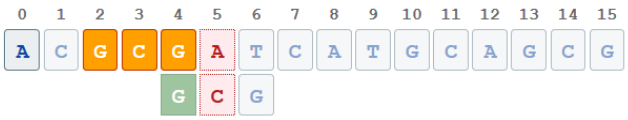
Setelah pattern digeser, dicek kembali karakter terakhir dengan prinsip BM, dan tidak ada match. Setelah ini akan dilakukan penggeseran sesuai dengan metode BM. Karena di pattern ada karakter C, karakter yang missmatch dengan string, maka indeks huruf c pada pattern disesuaikan dengan indeks c pada string, yaitu 3, sehingga huruf G awal pattern berada di indeks 2.



Proses JFS[6]

Setelah pattern digeser, dilakukan kembali pengecekan BM. Karena karakter terakhir pattern sesuai dengan karakter di

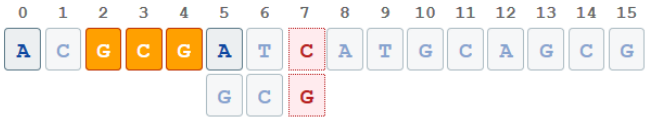
string berindeks 4, maka dilakukan KMP. G pada pattern sesuai dengan string pada index 2, dan huruf C pada indeks ke 3 sesuai pula dengan pattern. Karena ini, ditemukan match pattern pada string. Sehingga, sekuens gen “GCG” ditemukan pada indeks 2.



Proses JFS[6]

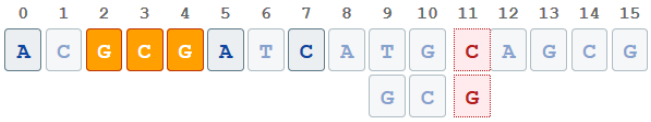
Karena pattern ditemukan, maka akan dicari match lain pada string tersebut. Karena pencarian berhasil, maka pattern akan maju sesuai dengan aturan KMP, yaitu dengan melihat boundary function. Karena sekarang pattern berada di indeks 4, maka akan dilihat b(4), yang dalam kasus ini adalah 1. Atau dengan kata lain, ada 1 karakter suffix yang sama dengan karakter prefix.

Maka, setelah maju, algoritma KMP dilanjutkan. Sekarang, huruf C dari pattern dibandingkan dengan karakter dengan indeks 5, yaitu A. Karena terjadi missmatch, maka pattern akan dimajukan satu indeks, dan pattern dimulai pada indeks 5



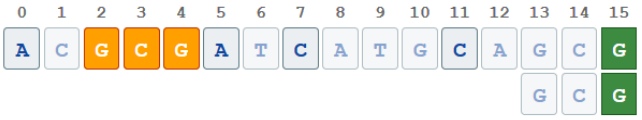
Proses JFS[6]

Karena sebelumnya terdapat missmatch, maka dilakukan lagi algoritma BM. Indeks ke-7 pada string tidak cocok dengan karakter terakhir pada pattern, sehingga akan dilakukan peloncatan dengan algoritma BM. Sehingga huruf terakhir pattern berada di indeks 11.



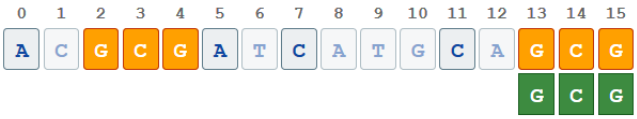
Proses JFS[6]

Karena sekali lagi tidak ditemukan kesamaan, maka akan dilakukan pelompatan BM, yang menghasilkan karakter terakhir pattern berada di indeks 15.



Proses JFS[6]

Karena karakter terakhir pattern match dengan karakter string berindeks 15, maka akan dilakukan KMP matching pada karakter awal pattern dengan karakter string berindeks 13.



Proses JFS[6]

Pada akhirnya, karena semua terjadi matching pada semua pattern, maka string berindeks 13 juga menjadi solusi. Jadi, untuk permasalahan ini, solusinya adalah indeks ke-2 dan ke 13.

VI. Pengembangan Penggunaan

Menurut saya, masih ada banyak pengembangan yang dilakukan. Mulai dari algoritmanya tersendiri. Mungkin masih banyak cara-cara yang dapat membuat pencarian String lebih efektif. Algoritma-algoritma baru ini harus dicari, karena pencarian string bukanlah hal yang sepele, melainkan selalu dipakai di dunia informatika.

Selanjutnya, algoritma juga bisa digunakan di banyak hal, seperti plagiarism checker, mencari typo, dll. Selain fungsi-fungsi itu pun, pasti masih banyak penggunaan searching string pada program-program pada umumnya, karena hampir tidak ada program yang tidak memakai file eksternal yang memanfaatkan pattern matching.

VII. Kesimpulan dan Saran

Ada banyak algoritma untuk pencocokan String, bahkan selain KM, BMP, dan FJS. Namun, dari semua algoritma yang telah dibahas, FJS lah yang paling cepat, karena menggabungkan algoritma BM dan KMP sehingga menjadi algoritma yang sangat efektif untuk pencarian String.

Saran saya adalah untuk mengembangkan apa yang sudah saya teliti di makalah ini, agar algoritma bisa lebih efektif dan bisa digunakan untuk kebaikan bersama dan kemajuan bangsa.

Terima Kasih.

REFERENCES

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Greedy-(2018).pdfdiakses pada 24 April 2019

[2] https://cgjennings.ca/articles/fjs.html diakses pada 24 April 2019

[3] https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/ diakses pada 24 April 2019

[4] https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ diakses pada 24 April 2019

[5] https://github.com/CGJennings/fjs-string-matching/blob/master/c/fjs.c diakses pada 24 April 2019

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, 26 April 2019

Johanes Boas Badia
13517009