PERANCANGAN ALGORITMA PENCOCOKAN STRING MENGGUNAKAN FUNGSI *HASH* UNTUK PENDETEKSIAN PLAGIARISME

Dian Perdhana Putra - 13507096

Mahasiswa Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jalan Ganesha 10 Bandung 40132
e-mail : if17096@students.if.itb.ac.id

ABSTRAK

Plagiarisme berasal dari bahasa latin *plagiarus*, yang artinya mencuri. Plagiarisme adalah pengambilan karya orang lain, berupa tulisan, makalah, atau pendapat dan sebagainya, untuk diakui sebagai hasil karya si pelaku. Plagiarisme adalah permasalahan yang telah berusia ratusan tahun, sering terjadi dalam dunia akademis dan jurnalistik. Isu tentang plagiarisme kembali menghangat, terutama akhir-akhir ini.

Pendeteksian plagiarisme adalah dengan cara memeriksa kesamaan suatu teks sumber dengan teks yang dicurigai mengandung plagiarisme. Cara yang paling mudah adalah dengan menggunakan *brute force*. Cara yang lebih cepat adalah dengan menggunakan algoritma pencarian string, misalnya Knuth-Morris Pratt, Booyer Moore, dsb. Namun, untuk teks yang sangat panjang, pencarian menggunakan algoritma pencarian string juga membutuhkan waktu lama disebabkan karena banyaknya operasi perbandingan. Untuk mengatasi permasalahan ini, salah satu solusinya adalah dengan menggunakan fungsi hash. Contoh penerapan dari solusi ini adalah pada algoritma Rabin-Karp.

Makalah ini akan membahas tentang perbandingan algoritma-algoritma pencocokan string, memperlihatkan penerapan fungsi hash di dalam algoritma pencocokan string, dan memberikan rancangan proses pemeriksaan plagiarisme.

Kata kunci: plagiarisme, algoritma pencocokan string, fungsi *hash*.

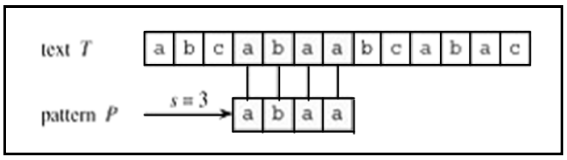
1. PENDAHULUAN

1.1 Algoritma Pencocokan *String*

Pencocokan *string* merupakan subjek yang amat penting dalam bidang *text processing*. Pencocokan string adalah usaha untuk menemukan keberadaan sebuah *string* (biasanya disebut *pattern*) dalam sebuah string yang lain (disebut *text*).

Masalah pencocokan *string* dapat didefinisikan secara lebih formal. Asumsikan bahwa *text* adalah sebuah *array of character* $T[1 \ldots n]$ dan *pattern* adalah *array of character* $P[1 \ldots m]$ dengan $m \leq n$. Karakter dalam *text T* dan *pattern P* adalah karakter dalam *finite alphabet* Σ. Sebagai contoh, Σ dapat berupa $\{a, b, \ldots, z\}$ atau $\{0,1\}$.

Kita dapat mengatakan bahwa *pattern P* ditemukan dengan *shift s* (*occurs with shift s*) dalam *text T* (atau dengan kata lain, *pattern P* ditemukan pada posisi *s*+1 dalam *text T*) jika $T[s+1 \ldots s+m] = P[1 \ldots m]$ dan $0 \leq s \leq n-m$. Jika *pattern P* ditemukan dalam *shift s* dalam *text T*, maka kita dapat menyebut *s* adalah *valid shift*, sebaliknya, kita menyebut *s* adalah *invalid shift* [1].



Gambar 1 : Pencocokan String. Pada gambar ini, *shift s* = 3 adalah *valid shift* [1].

Algoritma *brute force* adalah algoritma melacak keberadaan *pattern P* dalam *text T* dengan membandingkan satu-persatu karakter *pattern P* dan *text T*. Algoritma ini merupakan algoritma yang paling umum digunakan dan memiliki kompleksitas *O(mn)*. Terdapat banyak perbaikan dari algoritma *brute force* ini. Pencocokan *string* dengan operasi perbandingan dari kiri ke kanan, contohnya *Rabin-Karp algorithm*, *Knuth-Morris-Pratt algorithm*, *Forward Dawg Matching algorithm*, dan *Apostolico-Crochemore algorithm*.

Pencocokan *string* dengan operasi perbandingan dari kanan ke kiri, contohnya *Boyer-Moore algorithm*.

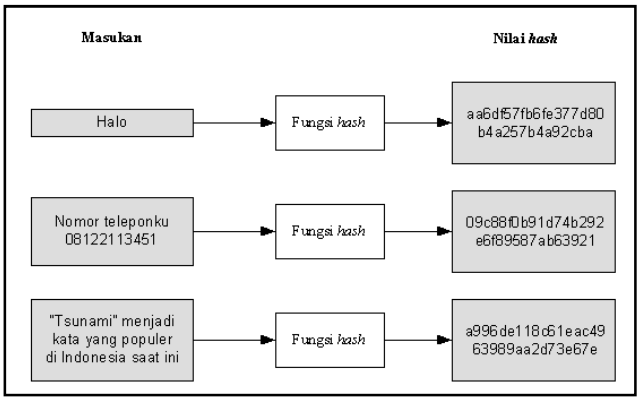
1.2 Fungsi *Hash*

Fungsi *hash* (*hash function*) adalah fungsi transformasi yang mengambil parameter data berukuran besar dan mengembalikan *ouput* berupa bilangan bulat atau *string* dengan ukuran yang tetap (*fixed-size string*).

Fungsi *hash H* menerima masukan string *M* dengan ukuran sembarang dan melakukan kompresi untuk menghasilkan nilai *hash* (*hash value*) yaitu *h*. Persamaan dari fungsi *hash* adalah sebagai berikut :

$$h = H(M) \qquad (1)$$

Nilai *hash* sering juga disebut dengan pesan ringkas (*message digest*). Gambar di bawah ini memperlihatkan fungsi *hash* akan selalu menghasilkan nilai *hash* yang panjangnya tetap walaupun ukuran pesan berbeda-beda.



Gambar 2 : Contoh penggunaan fungsi *hash* [1]

Fungsi *hash* memegang peranan yang sangat penting dalam aplikasi keamanan informasi dan kriptografi, misalnya *digital signature*, *message authentication codes* (*MACs*), dan bentuk-bentuk autentikasi lain.

Fungsi *hash* yang bersifat satu arah , disebut fungsi *hash* satu arah (one way *hash* function) adalah fungsi *hash* yang bekerja dalam satu arah, artinya pesan yang sudah diubah menjadi *message digest* tidak dapat dikembalikan lagi menjadi pesan semula. Oleh karena itu, dalam kriptografi, fungsi *hash* satu arah yang lazim digunakan.

Agar bisa digunakan dalam untuk mengamankan informasi, ada beberapa syarat yang harus dimiliki oleh fungsi *hash*[3] :

1. Fungsi *hash* dapat diterapkan pada blok berukuran berapa pun.
2. Fungsi *hash* menghasilkan nilai *hash* dengan panjang tetap.
3. Fungsi *hash H*(*x*) mudah dihitung untuk semua nilai *x* yang diberikan.
4. Untuk setiap nilai *hash h* yang diberikan, tidak mungkin mencari *x* sedemikian hingga *H*(*x*) = *h*.
5. Untuk setiap nilai *x* yang diberikan tidak mungkin mencari *y*, dimana $y \neq x$ sedemikian hingga *H*(*y*) = *H*(*x*).
6. Tidak mungkin untuk mencari pasangan *x* dan *y* di sedemikian hingga *H*(*y*) = *H*(*x*).

2. ANALISIS DAN PEMBAHASAN

2.1 Perbandingan Beberapa Algoritma Pencocokan String

Algoritma yang akan dibandingkan adalah algoritma pencocokan *string* yang sering digunakan, yaitu algoritma BruteForce, algoritma Knuth-Morris-Pratt, algoritma Boyer-Moore, dan algoritma Rabin-Karp yang menggunakan fungsi *hash.*

Berikut *pseudocode* dari masing-masing algoritma :

Algoritma Brute Force:

```
//Fungsi untuk menghitung jumlah perbandingan
//yang dilakukan dalam algoritma brute force
function hitungBruteForce
   (text : array [0..n-1] of char,
    pattern : array [0..m-1] of char)→ int{
       type j : integer
       type i : integer
       type count : integer

       count ← 0

       for i←0 to (n-m)do{
           j ← 0
           count ← count+1

           while ((j<m)and
                  (text[i+j]=pattern[j])){
              j ← j+1
              count ← count+1
           }
           if(j=m){
              →(count-1)
           }
       }
       → (count-1)
}
```

Algoritma Knuth-Morris-Pratt :

```
//Fungsi menghitung jumlah perbandingan yang
//gagal
//untuk menentukan fungsi pinggiran
function hitungGagal
    (pola : array [0..n-1]of char)
       →array [0..n-1] of integer{

       type fail : array [0..n-1] of integer

       fail[0] ← 0
```

```
        type m : integer
        type j : integer
        type i : integer
        m ← n
        j ← 0
        i ← 1

        while (i<m){
          if(pola[j]= pola[i]){
             fail[i] ← j+1
             i ← i+1
             j ← j+1
          }else if(j>0)
                j ← fail[j-1]
             else {
                fail[i] ← 0
                i ← i+1
             }
        }
        → fail
}

//Fungsi untuk menghitung jumlah perbandingan
//menggunakan Algoritma Knuth Morris Pratt
function hitungKMP(text : array [0..n-1] of char,
pattern : array [0..m-1] of char)→ int
{
        type j : integer
        type i : integer
        type count : integer

        count ← 0
        i←0
        j←0

        fail ← hitungGagal(pattern)

        while (i<n){
          count ← count+1
          if(pattern[j]=text[i]){
             if(j=m-1)
                → count
             i ← i+1
             j ← j+1
          } else {
             if(j>0)
                j ← fail[j-1]
             else
                i ← i+1
          }
        }
        → count
}
```

Algoritma Boyer-Moore :

```
//Fungsi yang mengembalikan larik integer yang
//berisi indeks
//dari kemunculan terakhir karakter ASCII dalam
//string yang dimasukkan
function buildLast(pola : array [0..m-1] of
char)→ array [0..127] of integer {
        type i : integer
        type last ← array [0..127] of integer

        for i←0 to 127 do
            last[i] ← -1
        for i←0 to m do
            last[pola[i]] ← i
        → last
}

//Fungsi untuk menghitung jumlah perbandingan
//menggunakan algoritma Boyer Moore
function hitungBM()
   (text : array [1..n] of char,
    pattern : array [1..m] of char)→ int {
       type last : array [1..m] of char
       type i : integer
       type lastOcc : integer
       last ← buildLast(pattern)

       i ← m-1
       count ← 0

       if(i > n-1)
          → count
       j ← m-1
          do{
             count ← count+1
             if(pattern[j]=text[i]){
                if(j=0)
                   →count
                else {
                   i ← i-1
                    j ← j-1
                }
             else{
               lastOcc ← last[text[i]]
               i ←(i+m-min(j, 1+lastOcc))
               j ← m-1
          }
   }while(i<=n-1)
   → -1;
}
```

Algoritma Rabin-Karp :

```
//Fungsi untuk menghitung jumlah perbandingan
//yang dilakukan dalam algoritma Rabin-Karp
function hitungRabinKarp
   (text : array [0..n-1] of char,
    pattern : array [0..m-1] of char)→ int{
       type j : integer
       type i : integer
       type count : integer

       count ← 0
       hPattern ← hash(pattern[0..m-1])
       hText ← hash(text[0..m-1])

       for i←0 to (n-m)do{
           if(hPattern=hText){
              k ← 0
              for j←i to i+m-1 do {
                  if text[j] = pattern[k]{
                      count ← count+1
                      k ← k+1
                   }
              }
           }
           hText ← hash(text[i+1..i+m])
       }
       → (count-1)
}
```

Hasil pengujian :

1. Pengujian pertama
   Text : `abababababababababababababaaaaa`
   Pattern : `aaaaa`

   Algoritma Brute Force : 47 kali operasi perbandingan
   Algoritma Knuth-Morrs-Pratt : 33 kali operasi perbandingan
   Algoritma Boyer-Moore : 19 kali operasi perbandingan
   Algoritma Rabin-Karp : 5 kali operasi perbandingan

2. Pengujian kedua
   Text : `abababababababaaaaaababababababab`
   Pattern : `aaaaa`

   Algoritma Brute Force : 58 kali operasi perbandingan
   Algoritma Knuth-Morrs-Pratt : 29 kali operasi perbandingan
   Algoritma Boyer-Moore : 25 kali operasi perbandingan
   Algoritma Rabin-Karp : 10 kali operasi perbandingan

3. Pengujian ketiga (kasus terbaik)
   Text : `aaaaaaaaaaaaaaaaaaaaaaaaabbbbb`
   Pattern : `bbbbb`

   Algoritma Brute Force : 29 kali operasi perbandingan
   Algoritma Knuth-Morrs-Pratt : 29 kali operasi perbandingan
   Algoritma Boyer-Moore : 11 kali operasi perbandingan
   Algoritma Rabin-Karp : 5 kali operasi perbandingan

4. Pengujian keempat (kasus terburuk)
   Text : `aaaaaaaaaaaaaaaaaaaaaaaaaaaab`
   Pattern : `aaaab`

   Algoritma Brute Force : 125 kali operasi perbandingan
   Algoritma Knuth-Morrs-Pratt : 53 kali operasi perbandingan
   Algoritma Boyer-Moore : 29 kali operasi perbandingan
   Algoritma Rabin-Karp : 5 kali operasi perbandingan

5. Pengujian kelima (*pattern* tidak ditemukan)
   Text : `aaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
   Pattern : `aaaab`

   Algoritma Brute Force : 125 kali operasi perbandingan
   Algoritma Knuth-Morrs-Pratt : 53 kali operasi perbandingan
   Algoritma Boyer-Moore : 25 kali operasi perbandingan
   Algoritma Rabin-Karp : 0 kali operasi perbandingan (tidak ada operasi perbandingan)

Analisis :

Algoritma *brute force* melakukan operasi perbandingan dengan cara memeriksa karakter dalam *pattern* dan *teks* satu per-satu, jika ditemukan karakter yang berlainan, maka pencarian akan digeser tepat satu langkah ke kanan. Hal ini akan menyebabkan banyaknya operasi perbandingan yang terjadi pada algoritma *brute force*. Untuk kasus terbaik (karakter pertama *pattern* tidak pernah sama dengan karakter pada *text* kecuali pada pencocokan yang terakhir) jumlah perbandingan yang dilakukan adalah *n* kali (pada contoh di atas perbandingan dilakukan sebanyak 29 kali untuk test dengan panjang 29 karakter). Sedangkan, untuk kasus terburuk, jumlah yang perbandingan yang dilakukan adalah sebanyak $m(n-m+1)$, untuk contoh di atas perbandingan dilakukan sebanyak 125 kali untuk *text* dengan panjang 29 karakter dan *pattern* dengan panjang 5 karakter. Oleh karena itu, kompleksitas waktu dari algoritma *brute force* adalah $O(mn)$.

Algoritma Knuth-Morris-Pratt membutuhkan waktu $O(m)$ untuk menghitung fungsi pinggiran, sedangkan proses pencarian *string* membutuhkan waktu $O(n)$. oleh karena itu kompleksitas waktu dari algoritma Knuth-Morris-Pratt adalah $O(m+n)$. Kompleksitas waktu yang dibutuhkan KMP cukup kecil, sehingga pencocokan *string* dengan algoritma KMP dapat dilakukan dengan cukup cepat. Selain itu, algoritma KMP tidak perlu melakukan *backward* dalam *text*, sehingga KMP efektif untuk memeriksa *text* yang berukuran besar.

Algoritma Boyer-Moore memindai karakter dari pattern dari kanan ke kiri mulai dari karakter yang paling kanan. Jika ditemukan mismatch (atau complete match untuk seluruh pattern) maka ada dua fungsi yang di gunakan untuk *shift* dari window (berisi *pattern* dan bagian dari *text* yang dibandingkan). Kedua fungsi itu adalah *good-suffix shift* (*matching shift*) dan *bad-character shift* (atau disebut *occurrence shift*). Algoritma ini sangat efektif dan disebut sebagai algoritma paling efektif untuk string matching dimana kompleksitas algoritmanya adalah $O(n)$.

Algoritma Rabin_Karp menggunakan fungsi hash untuk menghindari operasi perbandingan karakter yang tidak perlu. Alih-alih melakukan operasi perbandingan karakter pada *text* yang terlihat mirip dengan *pattern*, algoritma ini mengambil *substring* dari *text* sepanjang *pattern*, mulai dari kiri ke kanan, dan membandingkan apakah nilai hash dari substring yang diambil tersebut

sama dengan nilai hash dari *pattern*. Tahap *preprocessing* dari algoritma Rabin-Karp ini, yaitu menghitung *hash* (*pattern*) memiliki kompleksitas waktu $O(m)$. Proses pencarian dilakukan dengan membandingkan *hash* (*pattern*) dengan *hash* (*text*[*j..j+m*-1]) untuk $0 \leq j \leq n\text{-}m$. Jika nilai keduanya, maka proses selanjutnya adalah membandingkan kesamaan dari *pattern* dan *text*[*j..j+m*-1] karakter per karakter. Kompleksitas waktu dari tahap pencarian dari algoritma Rabin-Karp sendiri adalah $O(mn)$. Jika tidak pernah ditemukan kesamaan pada *hash* (*pattern*) dan *hash* (*text*[*j..j+m*-1]), maka algoritma Rabin-Karp tidak melakukan operasi perbandingan karakter sama sekali.

2.2 Pemilihan Fungsi *Hash*

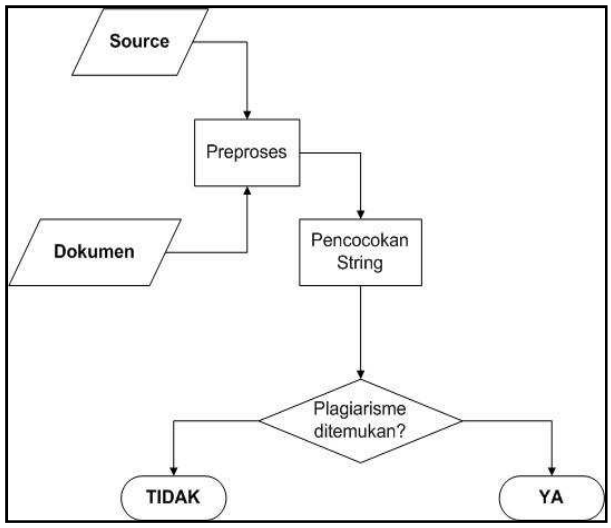
Agar dapat digunakan di dalam pencocokan string, fungsi *hash* yang dipilih harus memiliki karakteristik sebagai berikut :

1. Mudah dikomputasi untuk masukan string apapun.
2. Tidak mungkin menemukan pesan dari hash value yang diberikan.
3. Tidak mungkin memodifikasi pesan tanpa mengubah nilai *hash*nya.
4. Tidak mungkin menemukan dua buah pesan dengan nilai *hash* yang sama

Ada beberapa fungsi hash yang memenuhi semua kriteria di atas, yaitu RIPEMD-128/256, RIPEMD-160/320, SHA-1, SHA-256/224, SHA-512/384, Tiger(2)-192/160/128, dan WHIRLPOOL.

Fungsi *hash* yang dapat digunakan dalam pencocokan string dapat memilih dari salah satu fungsi di atas.

2.3 Metode Pemeriksaan Plagiarisme dengan Algoritma Pencocokan String menggunakan Fungsi Hash



Gambar 3 : Skema Pemeriksaan Plagiarisme dalam Dokumen

Penulis mengusulkan skema di atas yang menunjukkan proses yang harus dilakukan untuk memeriksa plagiarisme dalam sebuah dokumen. Diberikan sebuah dokumen sumber dan dokumen yang dicurigai mengandung plagiarisme. Maka tahap-tahap yang harus dilakukan hingga dapat menemukan kesimpulan dokumen tersebut mengandung plagiarism adalah :

1. Tahap Preproses

Sebelum bisa dilakukan pencocokan string, kedua dokumen harus melalui tahap preproses. Tujuan dari tahap ini adalah untuk menyamakan format penulisan dari kedua dokumen, sehingga format penulisan tidak menjadi hambatan untuk tahap selanjutnya. *Parser* dibuat dan digunakan pada tahap ini.

Tahap ini terbagi menjadi tiga tahap yaitu :

A. Penyamaan ukuran huruf (lower case/upper case)

Penyamaan ukuran huruf pada dokumen sumber dan dokumen yang dicurigai mengandung plagiarisme sangat penting sebab seringkali program mengenali huruf melalui karakter ASCII, di mana huruf kecil dan huruf besar memiliki nomor karakter yang berbeda. Hal ini dilakukan untuk menghindari kesalahan pada penghitungan fungsi *hash*.

Contoh :
Sebelum tahap 1A :

```
Budi bermain bola, Ibu memasak,
Ayah pergi bekerja
```

Setelah tahap 1A (*upper case*) :
`BUDI BERMAIN BOLA, IBU MEMASAK, AYAH PERGI BEKERJA`

B. Penghilangan tanda baca, spasi dan *newline* pada kedua dokumen

Penghilangan tanda baca, spasi dan *newline* dilakukan, sebab karakter pada tanda baca, spasi dan *newline* dihitung sebagai karakter tersendiri. Kemungkinan pelaku plagiarism membedakan tanda baca, spasi dan *newline* untuk mengecoh pembaca.

Contoh :
Sebelum tahap 1A :
`budi bermain bola, ibu memasak, ayah pergi bekerja.`

Setelah tahap 1A :
`budibermainbolaibumemasakayahpergibekerja`

C. Penghilangan *stopwords* pada kedua dokumen

Stopwords adalah kata-kata yang sering sekali digunakan dan tidak memiliki makna bila berdiri sendiri, biasanya *stopwords* berupa kata ganti, kata penghubung, atau kata depan. Contoh *stopwords* dalam bahasa Inggris adalah `"of"`, `"the"`, `"a"`, `"and"`, dan `"or"`. Sedangkan untuk bahasa Indonesia diantaranya `"yang"`, `"di"`, `"ke"`, `"karena"`, `"jika"`.

Stopwords perlu dihilangkan karena dua alasan, yaitu menghemat tempat dan waktu pencarian, dan untuk menghindari kesalahan pemaknaan oleh parser. Sebagai contoh *`"karena"`* dan *`"sebab"`* adalah dua *stopword* yang memiliki makna yang sama. Jika pelaku plagiarisme mengganti `"karena"` dengan `"sebab"`, dan *stopword-stopword* tersebut tidak dihilangkan, parser akan mengenali kedua kata tersebut sebagai kata yang berbeda dan *plagiarisme* bisa tidak terdeteksi.

2. Tahap Pencocokan String

Setelah melalui tahap preproses, dokumen sumber dan dokumen yang dicurigai mengandung plagiarism akan memiliki format yang sama, yaitu ukuran huruf sama (*lower case* atau *upper case*) dan tidak mengandung *stopwords*, tanda baca, spasi dan karakter *newline*. Setelah itu, barulah dapat dilakukan pemeriksaan dengan menggunakan algoritma pencocokan string, apakah dokumen yang dicurigai mengandung plagiarisme atau tidak. Jika ternyata ada bagian dari dokumen sumber yang terdapat pada dokumen yang dicurigai mengandung plagiarisme, tanpa ada rujukan penulis dokumen sumber, maka dapat disimpulkan bahwa dokumen tersebut mengandung plagiarisme.

Skema di atas adalah skema dasar dari proses pemeriksaan plagiarism menggunakan algoritma pencocokan string dengan fungsi hash. Ke depannya, proses tersebut dapat dikembangkan lagi, misalnya dengan menggunakan teknik *natural language processing* dan kakas penerjemah yang handal.

IV. KESIMPULAN

1. Fungsi *hash* memiliki banyak sekali kegunaan, terutama dalam bidang kriptografi.
2. Untuk dapat digunakan dalam kriptografi, fungsi has harus memiliki karakteristik sebagai berikut : mudah dikomputasi untuk masukan string apapun, tidak mungkin menemukan pesan dari *hash* value yang diberikan, tidak mungkin memodifikasi pesan tanpa mengubah nilai *hash*nya, dan tidak mungkin menemukan dua buah pesan dengan nilai *hash* yang sama.
3. Penggunaan fungsi *hash* dalam algoritma pencocokan string akan mengurangi jumlah perbandingan karakter, seperti pada algoritma Rabin-Karp. Untuk ukuran file yang besar, akan sangat efektif bila kita menggunakan fungsi *hash*.
4. Pemeriksaan plagiarism memiliki dua tahap, yaitu preproses dan tahap pencocokan string.
5. Tahap preproses terdiri tiga tahap, yaitu : penyamaan ukuran huruf; penghilangan tanda baca, spasi, dan *newline*; dan penghilangan *stopwords.*
6. Tahap pencocokan string adalah tahap mencocokkan karakter pada dokumen sumber dan dokumen yang dicurigai mengandung plagiarisme menggunakan algoritma pencocokan string dan fungsi *hash*.
7. Ke depannya, pemeriksaan plagiarism dengan fungsi hash dapat dikembangkan lagi, misalnya dengan menggunakan teknik *natural language processing* dan kakas penerjemah yang handal.

REFERENSI

[1] Rinaldi Munir, "Diktat Kuliah IF3038 Kriptografi", Teknik Informatika ITB, 2006.

[2] Cormen, Thomas, et al, "*Introduction to Algorithm"*, McGraw-Hill,2001

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, 17 Mei 2010

Dian Perdhana Putra
13507096