Visualisasi Beberapa Algoritma Pencocokan *String* Dengan Java

Gozali Harda Kumara (13502066)
Teknik Informatika
Sekolah Teknik Elektro Informatika
Institut Teknologi Bandung

Abstraksi

Algoritma pencocokan *string* yang banyak dikembangkan membuat pemahaman terhadap cara kerja setiap algoritma tersebut sulit. Sebuah aplikasi yang dapat membantu pemahaman dengan memvisualisasikan algoritma-algoritma pencocokan *string* tentu akan membantu pemahaman atas algoritma yang divisualisasikan.

Pada makalah ini, akan dideskripsikan pengembangan sebuah perangkat lunak untuk memvisualisasikan algoritma pencocokan *string* yang dibangun dengan *platform* Java dan dapat dijalankan sebagai *applet* dan aplikasi. Animasi yang dilakukan oleh aplikasi ini dibangun di atas pustaka Java2D. Implementasi dan pengujian aplikasi ini dilakukan pada *platform* Windows maupun Linux.

Aplikasi ini dibangun berdasarkan analisis tentang apa saja yang diperlukan agar sebuah visualisasi terhadap algoritma pencocokan *string* mampu meningkatkan pemahaman terhadap algoritma tersebut. Hasil dari analisis tersebut antara lain: visualisasi teks dan *pattern*, penampilan algoritma dan bagian yang dieksekusinya, dan penampilan variabel yang dimiliki algoritma pada setiap langkahnya.

Selain memvisualisasikan algoritma pencocokan *string*, aplikasi ini juga mampu membandingkan algoritma-algoritma tersebut atas jumlah perbandingan karakter dan waktu eksekusi yang diperlukan. Algoritma-algoritma pencocokan *string* yang divisualisasikan oleh aplikasi ini antara lain: *brute force*, Knuth-Morris-Pratt, Boyer-Moore, Turbo Boyer-Moore, Colussi, dan Crochemore-Perrin.

Kata Kunci: Algoritma Pencocokan *String*, Visualisasi Algoritma, Java *Applet*, Java2D.

1 Pendahuluan

Algoritma pencocokan *string* yang paling naif adalah algoritma *brute force*, algoritma ini menggeser *pattern* di setiap karakter teks, lalu membandingkan setiap karakter pada *pattern*, sampai *pattern* itu ditemukan di teks atau terjadi ketidakcocokan. Banyak Algoritma-algoritma lain yang dikembangkan untuk memperbaiki kinerja algoritma *brute force* ini, dan dapat diklasifikasikan menjadi tiga kategori berdasarkan arah pencocokan karakter.

Tiga kategori itu adalah, pertama, arah yang paling alami, yaitu dari kiri ke kanan, yang merupakan arah ketika membaca, algoritma yang termasuk kategori ini adalah algoritma *brute force* itu sendiri, algoritma dari Knuth, Morris, dan Pratt, algoritma dari Karp dan Rabin, serta algoritma dari Apostolico dan Crochemore. Kategori selanjutnya dari kanan ke kiri, arah yang biasanya menghasilkan hasil terbaik di praktek, contohnya adalah algoritma dari Boyer dan Moore, yang kemudian dikembangkan menjadi Turbo Boyer Moore, Tuned Boyer Moore, dan Zhu-Takaoka. Dan kategori terakhir, dari urutan yang ditentukan secara spesifik oleh algoritma tersebut, arah ini menghasilkan hasil terbaik secara teoritis, algoritma yang termasuk kategori ini adalah Colussi, Galil-Seiferas, dan Crochemore-Perrin [CHA01].

Akibat dari banyaknya algoritma-algoritma yang dikembangkan adalah kesulitan untuk memahami semua algoritma tersebut karena cara kerja yang sangat berbeda. Keunikan cara kerja algoritma-algoritma tersebut sangatlah menarik untuk dipelajari dan dipahami. Sebuah algoritma haruslah dilihat untuk dipercaya, dan cara yang paling baik untuk mempelajari sebuah algoritma adalah dengan mencobanya [KNU68]. Untuk itu, sebuah aplikasi

yang dapat membantu pemahaman cara kerja algoritma pencocokan *string* dengan sebuah visualisasi diperlukan.

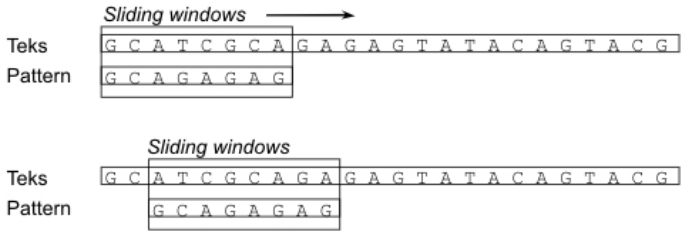
Java merupakan *platform* pengembangan yang cocok untuk aplikasi yang akan dikembangkan ini. Java2D dari Java dapat menganimasikan pergerakan *pattern* dalam pencocokan *string* dengan baik, dan aplikasi Java juga dapat dibuat menjadi sebuah *applet* agar dapat diakses dan dijalankan dari *web* sehingga memudahkan pengguna untuk mengakses aplikasi.

Pada makalah ini, akan dideskripsikan pengembangan aplikasi yang pemvisualisasi algoritma pencocokan *string*. Makalah ini disusun sebagai berikut, pada Bagian 2, akan dijelaskan algoritma pencocokan *string.* Kemudian, Bagian 3 akan dijelaskan hasil analisis tentang apa yang diperlukan untuk memvisualisasikan pencocokan *string*. Bagian 4 akan menjelaskan pengembangan perangkat lunak visualisasi algoritma pencocokan *string* yang dikembangkan. Pada akhirnya, di Bagian 5 akan disertakan kesimpulan dan saran yang berkaitan dengan makalah ini.

2 Algoritma Pencocokan *String*

Pencocokan *string* atau *string matching* adalah proses pencarian semua kemunculan *string* pendek *P[0..n-1]* yang disebut *pattern* di *string* yang lebih panjang *T[0..m-1]* yang disebut teks. Pencocokan *string* merupakan permasalahan paling sederhana dari semua permasalahan *string* lainnya, dan merupakan bagian dari pemrosesan data, pengkompresian data, *lexical analysis*, dan temu balik informasi. Teknik untuk menyelesaikan permasalahan pencocokan *string* biasanya akan menghasilkan implikasi langsung ke aplikasi *string* lainnya [BRE92].

Proses pencocokan *string* menggunakan sebuah *window* yang bergeser di teks. *Window* itu mempunyai panjang yang sama dengan panjang *pattern*. Pada permulaan pencocokan *string, window* itu diletakkan di ujung kiri teks, lalu karakter-karakter di *window* dibandingkan dengan karakter-karakter di *pattern*. Pencocokan karakter-karakter di sebuah *window* yang sama selanjutnya disebut sebagai suatu *attempt* [CHA98]. Setelah setiap *attempt*, *window* akan digeser ke kanan dengan jarak tertentu di teks, dan baru akan berhenti bila *window* tersebut sampai di ujung kanan teks. Contoh *sliding windows* yang sedang menggeser *pattern* di teks dapat dilihat di Gambar 1.



Gambar 1 Mekanisme *Sliding Windows*

2.1 Algoritma *Brute Force*

Metode yang paling mudah dan jelas untuk menemukan kemunculan *pattern* di teks adalah dengan mencoba setiap posisi *pattern* di teks, lalu mencocokkan karakter *pattern* dan teks pada posisi tersebut. Algoritma *brute force* menggunakan metode ini, sehingga mempunyai kompleksitas yang kuadratik.

```
Procedure BruteForceSearch(
          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: integer

Algoritma:
     for (i:=0 to  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
      endfor
```

Gambar 2 *Pseudocode* Algoritma *Brute Force*

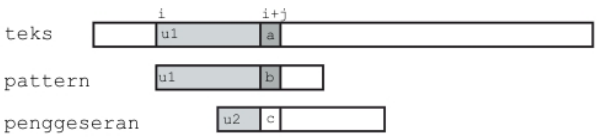
Pseudocode algoritma *brute force* diilustrasikan oleh Gambar 2. Algoritma *brute force* pada *pseudocode* tersebut menerima masukan berupa sebuah teks *T[0..m-1]*, sebuah *pattern P[0..n-1],* dan akan mengeluarkan array dari boolean *ketemu[0..m-1]*. Variabel-variabel yang dideklarasikan adalah *i* untuk posisi *pattern* terhadap teks, dan *j* untuk posisi karakter di *pattern* yang sedang dicocokan.

2.1.1 Kompleksitas

Kompleksitas waktu algoritma *brute force* pada fase pencocokan adalah $O(mn)$ dengan m adalah panjang dari teks dan n adalah panjang dari *pattern* [CHA01].

2.2 Algoritma Knuth-Morris-Pratt

Jika algoritma *brute force* dilihat dengan lebih mendalam, dapat diketahui bahwa dengan mengingat beberapa pembandingan karakter yang telah dilakukan sebelumnya, besar pergeseran yang dilakukan dapat ditingkatkan.



Gambar 3 Penyejajaran *u2* dengan Akhiran dari *u1*

Perhitungan penggeseran pada algoritma ini adalah sebagai berikut, bila terjadi ketidakcocokan pada saat *pattern* sejajar dengan *T[i..i+n-1]*, anggap ketidakcocokan pertama terjadi diantara *T[i+j]* dan *P[j]*, dengan $0<j<n$. Berarti, *T[i..i+j-1]=P[0..j-1]* dan *a=T[i+j] b=P[j]*. Bila *u1* adalah awalan dari *pattern* sampai ketidakcocokan terjadi, dan *u2* adalah awalah dari *u1*, maka ketika pergeseran dilakukan, sangat beralasan bila *u2* akan sama dengan akhiran dari *u1*. Sehingga *pattern* bias digeser agar *u2* tersebut sejajar dengan akhiran dari *u1*. Hal ini diilustrasikan oleh Gambar 3.

```
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]))
         j := kmpNext[j]
      i:= i+1
      j:= j+1
      if (P[i] = P[j])
         kmpNext[i] := kmpNext[j]
      else
         kmpNext[i] := j
      endif
   endwhile
```

Gambar 4 *Pseudocode* Penghitungan Tabel *kmpNext*

Dengan kata lain, pencocokan *string* akan berjalan secara efisien bila didefinisikan tabel yang menentukan berapa panjang penggeseran seandainya terdeteksi ketidakcocokan di karakter ke-*j* dari *pattern*. Tabel itu harus memuat *next*[*j*] yang merupakan posisi karakter *P[j]* setelah digeser, sehingga *pattern* bias digeser sebesar *j-next*[*j*] relatif terhadap teks [KNU77]. Pseudocode fase inisialisasi dan pencarian algoritma Knuth-Morris-Pratt ditunjukkan oleh Gambar 4 dan 5.

```
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 integer

Algoritma:
   preKMP(n, P, kmpNext)
   i:=0
   j:=0
   while (i<= m-n) do
      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]
      if(kmpNext[j]>0)
         j:= kmpNext[j]
      else
         j:=0
      endif
      i:= i+next
   endwhile
```

Gambar 5 Algoritma Knuth-Morris-Pratt

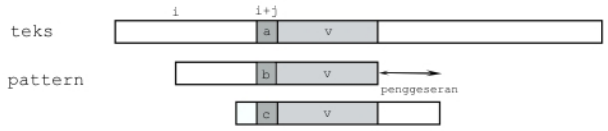
2.2.1 Kompleksitas

Algoritma ini menemukan semua kemunculan dari *pattern* dengan panjang n di dalam teks dengan panjang m dengan kompleksitas waktu $O(m+n)$. Algoritma ini hanya membutuhkan $O(n)$ ruang dari *memory* internal jika teks dibaca dari *file* eksternal. Semua besaran O tersebut tidak tergantung pada besarnya ruang alfabet [KNU77].

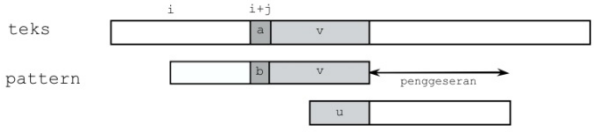
2.3 Algoritma Boyer-Moore

Algoritma Boyer-Moore dipublikasikan oleh Robert S. Boyer, dan J. Strother Moore pada tahun 1977. Algoritma ini dianggap sebagai algoritma yang paling efisien pada aplikasi umum [CHA01]. Tidak seperti dua algoritma sebelumnya, algoritma Boyer-Moore memulai mencocokkan karakter dari sebelah kanan *pattern*. Ide dibalik algoritma ini adalah bahwa dengan memulai pencocokan karakter dari kanan, dan bukan dari kiri, maka akan lebih banyak informasi yang didapat [BOY77].

Misalnya ada sebuah *attempt* pencocokan yang terjadi pada *T[i..i+n-1]*, dan anggap ketidakcocokan pertama terjadi diantara *T[i+j]* dan *P[j]*, dengan *0<j<n*. Berarti, *T[i+j+1..i+n-1]=P[j+1..n-1]* dan *a=T[i+j]≠b=P[j]*. Jika *v* adalah akhiran dari *pattern* setelah *b* dan *u* adalah sebuah awalan dari *pattern*, maka penggeseran-penggeseran yang mungkin adalah Penggeseran *good-suffix* dan penggenseran *bad-character*.



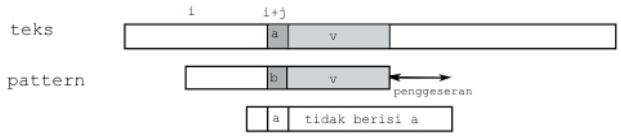
Gambar 6 Penggeseran *good-suffix* Bila Ada Potongan *u* yang Sama.



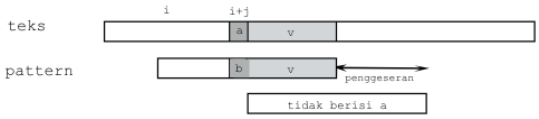
Gambar 7 Penggeseran *good-suffix* Jika Hanya Ada Awalan *v* dari *Pattern* yang Sama dengan Akhiran *u* dari

Penggeseran *good-suffix* yang terdiri atas mensejajarkan potongan *T[i+j+1..i+n-1]=P[j+1..n-1]=v* dengan kemunculannya paling kanan di *pattern* yang didahului oleh karakter yang berbeda dengan *P[j]*, penggeseran tersebut diilustrasikan di Gambar 6. Namun, jika tidak ada potongan seperti itu, maka algoritma akan mensejajarkan akhiran dari *v* dari *T[i+j+1..i+n-1]* dengan awalan *u* dari *pattern* yang sama, seperti yang diilustrasikan di Gambar 7.

Penggeseran *bad-character* yang terdiri dari mensejajarkan *T[i+j]* dengan kemunculan paling kanan karakter tersebut di *pattern*, penggeseran ini diilustrasikan oleh Gambar 8. Dan bila karakter tersebut tidak ada di *pattern*, maka *pattern* akan disejajarkan dengan *T[i+n+1]*, seperti yang diilustrasikan oleh Gambar 9. Penggeseran *bad-character* ini akan sering terjadi pada pencocokan *string* dengan ruang alfabet yang besar dan dengan *pattern* yang pendek yang sering terjadi di praktek pada umumnya. Hal ini terjadi karena akan banyak karakter di teks yang tidak muncul di *pattern*. Namun, untuk *file* biner, yang mempunyai alfabet Σ ={0, 1}, penggeseran ini kemungkinan besar tidak akan membantu sama sekali. Hal ini dapat diatasi dengan membandingkan beberapa bit sekaligus.



Gambar 8 Penggeseran Bad Character Bila di *Pattern* Terdapat Karakter *a*



Gambar 9 Penggeseran *Bad-Character* Bila *Pattern* Tidak Mengandung Karakter *a*

```
Procedure preBmBc(
   input P : array[0..n-1] of char,
   input n : integer,
   input/output bmBc : array[0..ASIZE-1] of integer
)

Deklarasi:
   i: integer

Algoritma:
   for (i := 0 to ALPHABETSIZE-1)
      bmBc[i] := n
   endfor
   for (i := 0 to n - 2)
      bmBc[P[i]] := n - i - 1
   endfor
```

Gambar 10 *Pseudocode* Penghitungan Tabel *bmBc*

```
Procedure BoyerMooreSearch(
   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, shift, bmBcShift, bmGsShift: integer
   BmBc : array[0..ALPHABETSIZE] of integer
   BmGs : array[0..n-1] of integer

Algoritma:
   preBmBc(n, P, BmBc)
   preBmGs(n, P, BmGs)
   i:=0
   while (i<= m-n) do
      j:=n-1
      while (j >=0 n and T[i+j] = P[j]) do
         j:=j-1
      endwhile
      if(j < 0) then
         ketemu[i]:=true
         shift := bmGs[0]
      else
         bmBcShift:= BmBc[chartoint(T[i+j])]-n+j+1
         bmGsShift:= BmGs[j]
         shift:= max(bmBcShift, bmGsShift)
      endif
      i:= i+shift
   endwhile
```

Gambar 11 *Pseudocode* Algoritma Boyer-Moore

2.3.1 Kompleksitas

Tabel untuk penggeseran *bad-character* dan *good-suffix* dapat dihitung dengan kompleksitas waktu dan ruang sebesar *O*(*n*+) dengan adalah besar ruang

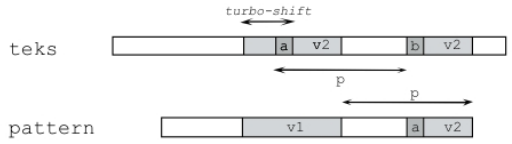
alfabet. Sedangkan pada fase pencocokan, algoritma ini mempunyai kompleksitas waktu sebesar $O(mn)$, pada kasus terburuk, algoritma ini akan melakukan $3m$ pencocokan karakter, namun pada performa terbaiknya algoritma ini hanya akan melakukan $O(m/n)$ pencocokan [CHA01].

2.4 Algoritma Turbo Boyer-Moore

Algoritma Turbo Boyer-Moore adalah sebuah variasi dari algoritma Boyer-Moore , Bila dibandingkan dengan algoritma Boyer-Moore, algoritma ini tidak membutuhkan pemrosesan ekstra. Berbeda dengan algoritma Boyer-Moore, algoritma ini mengingat faktor dari teks yang cocok dengan akhiran dari *pattern* selama *attempt* terakhir. Dengan demikian, teknik ini mempunyai dua keunggulan[CHA01]:

1. Teknik ini memungkinkan untuk melompati faktor dari teks tersebut.
2. Teknik ini mengijinkan sebuah penggeseran *turbo*.

Penggeseran *bad character* dan *good suffix* pada algoritma Turbo-Boyer-Moore sama dengan penggeseran yang dilakukan algoritma Boyer-Moore. Sedangkan sebuah penggeseran *turbo* dapat terjadi bila pada *attempt* yang sedang dilakukan, akhiran dari *pattern* yang cocok dengan teks lebih pendek dari bagian dari teks yang diingat dari *attempt* sebelumnya. Pada kasus ini, anggap *v1* adalah faktor yang diingat dari *attempt* sebelumnya, dan *v2* adalah bagian dari *pattern* yang cocok pada *attempt* yang sedang dilakukan, sehingga *v1zv2* adalah akhiran dari *pattern*.



Gambar 12 Penggeseran *Turbo*

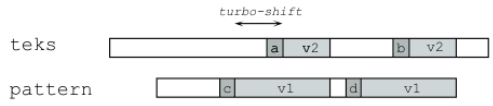
Lalu anggap *a* adalah karakter di teks dan *b* adalah karakter dari *pattern* yang sedang dicocokan pada *attempt* tersebut. Maka *av2*, adalah akhiran dari *pattern*, dan juga akhiran dari *v1* karena $|v2|<|v1|$. Dua karakter *a* dan *b* muncul dengan jarak *p* di teks, dan akhiran dari *v1zv2* dari *pattern* mempunyai periode $p=|zv2|$, karena *v1* merupakan pinggiran dari *v1zv2*, sehingga tidak mungkin melewati dua kemunculan karakter *a* dan *b* di teks. Penggeseran terkecil yang mungkin dilakukan adalah sebesar $|v1|-|v2|$, yang disebut sebagai penggeseran *turbo* dan diilustrasikan oleh Gambar 13.

```
Procedure TurboBoyerMooreSearch(
     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, ,v1, v2, shift, bmBcShift, bmGsShift, turboShift: integer
 BmBc : array[0..255] of interger
 BmGs : array[0..n-1] of interger

Algoritma:
 preBmBc(n, P, BmBc)
 preBmGs(n, P, BmGs)
 i:= v1:= 0
 shift:= n
 while (i<= m-n) do
     j:=n-1
     while (j >=0 n and T[i+j] = P[j]) do
          j:=j-1
          if (not(v1 = 0) and (i = m - 1 - shift))
               j:=j-v1
     endwhile
     if(j < 0) then
          ketemu[i]:=true
          shift:= bmGs[0]
          v1=n-shift
     else
          v2:= =n-1-j
          turboShift:= v1-v2
          bmBcShift:= BmBc[chartoint(T[i+j])]-n+j+1
          bmGsShift:= BmGs[j]
          shift:= max(bmBcShift, bmGsShift)
          shift:= max(shift, turboShift)
          if (shift = bmGs[i]) then
               v1:=min(m-shift, v2)
          else
               if (turboShift<bmBcshift)
                    shift:= max(shift,v1+1)
               endif
               v1:=0
          endif
     endif
     i:= i+shift
endwhile
```

Gambar 13 *Pseudocode* Algoritma Turbo Boyer-Moore



Gambar 14 Penggeseran Harus Lebih dari $|v1|+1$

Jika terjadi kasus dimana $|v2|>|v1|$, dan panjang dari penggeseran *bad-character* lebih besar dari penggeseran *good-suffix* maupun penggeseran *turbo*. Pada kasus ini, seperti yang diilustrasikan pada Gambar 14, dua karakter *c* dan *d* pastilah berbeda karena disyaratkan bahwa jika $v1 \neq 0$ maka penggeseran sebelumnya adalah penggeseran *good-suffix*. Sebagai akibatnya, jika penggeseran dengan panjang yang lebih besar dari penggeseran *turbo* namun lebih kecil dari $|v1|+1$ maka *c* dan *d* akan disejajarkan dengan karakter yang sama di teks. Oleh karena itu, dalam kasus ini panjang penggeseran minimal adalah $|v1|+1$.

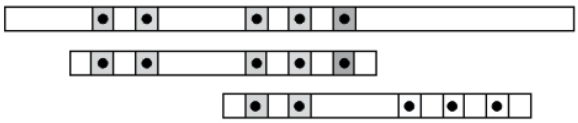
2.4.1 **Kompleksitas**

Fase inisialisasi pada algoritma ini sama dengan fase inisialisasi pada algoritma Boyer-Moore, yaitu mempunyai kompleksitas waktu dan ruang sebesar $O(n + \sigma)$ dengan σ adalah besar ruang alfabet. Sedangkan pada fase pencocokan, algoritma ini mempunyai kompleksitas waktu sebesar $O(m)$, Jumlah pencocokan karakter pada algoritma ini adalah $2m$ [CHA01].

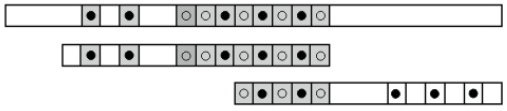
2.5 Algoritma Colussi

Algoritma Colussi dipublikasikan oleh Livio Colussi pada tahun 1994. Algoritma ini dirancang berdasarkan analisis terhadap algoritma Knuth-Morris-Pratt. Pada algoritma ini, *pattern* dibagi menjadi dua subhimpunan, yang disebut lubang dan non-lubang. Sebuah posisi *h* disebut lubang jika semua periode dari *P[0..h-1]* berlanjut sampai *P[0..h]*, atau dengan kata lain sebuah posisi *h* disebut lubang jika tidak ada periode dari awalan *pattern* yang berhenti pada posisi *h* [BRE92]. Dalam pencocokan, algoritma Colussi membagi pencocokan menjadi dua fase, yaitu mencocokkan karakter-karakter non-lubang dari kiri ke kanan, lalu baru mencocokkan karakter lubang dari kanan ke kiri. Penghitungan penggeseran adalah sebagai berikut:

Bila ketidakcocokan terjadi pada fase pertama, algoritma bisa menggeser *pattern* sehingga tidak perlu lagi mencocokkan karakter teks yang sejajar dengan non-lubang pada *attempt* sebelumnya. Hal ini diilustrasikan oleh Gambar 16.



Gambar 15 Penggeseran Bila Ketidakcocokan Terjadi Pada Fase Pertama Algoritma Colussi



Gambar 16 Penggeseran Bila Ketidakcocokan Terjadi Pada Fase Kedua Algoritma Colussi

Bila ketidakcocokan terjadi pada fase kedua, itu berarti sebuah akhiran dari *pattern* cocok dengan sebuah faktor dari teks. Dan setelah sebuah penggeseran yang bersesuaian, tidak lagi dibutuhkan untuk mencocokkan faktor tersebut lagi. Hal ini diilustrasikan oleh Gambar 17.

```
Procedure ColussiSearch(
     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, nd, last: integer
     h : array[0..n] of interger
     next : array[0..n] of interger
     shift : array[0..n] of interger

Algoritma:
     nd = preColussi(n, P, h, next, shift)
     i:= j:= 0
     last:= -1
     while (i<= m-n) do
          while (j < n and last < i+h[j] and T[i+h[j]] = P[h[j]]) do
               j:=j+1
          endwhile
          if(j >= n or last >= i+h[j]) then
               ketemu[i]:=true
          endif
          if (j>nd) then
               last:= i+m-1
          endif
          i:= i+shift[j]
          j:= next[j]
     endwhile
```

Gambar 17 *Pseudocode* Algoritma Colussi

2.5.1 **Kompleksitas**

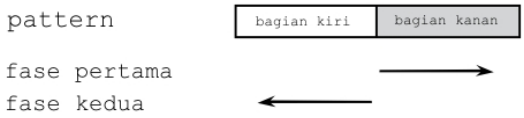
Fase inisialisasi pada algoritma ini mempunyai kompleksitas waktu dan ruang $O(n)$, sedang fase pencocokan dapat dilakukan dengan kompleksitas waktu O(m), atau lebih tepatnya, paling banyak hanya dilakukan $3/2m$ pencocokan karakter [CHA01].

2.6 Algoritma Crochemore-Perrin

Algoritma *Crochemore-Perrin*, yang sering juga disebut algoritma *Two Way Algorithm*, atau Algoritma Dua Arah dipublikasikan Maxime Crochemore dan Dominique Perrin pada tahun 1991. Algoritma ini memfaktorkan *pattern* menjadi dua bagian $pattern_{kiri}$, dan $pattern_{kanan}$ sehingga $pattern = pattern_{kiri} pattern_{kanan}$. Fase pencocokan pada algoritma ini terdiri dari dua bagian, pertama mencocokkan karakter $pattern_{kanan}$ dari kiri ke kanan, lalu mencocokkan karakter $pattern_{kiri}$ dari kanan ke kiri [CHA01]. Hal ini diilustrasikan pada Gambar 21.

Fase inisialisasi pada algoritma ini menghitung faktorisasi yang baik dari *pattern* atas $pattern_{kiri}$ dan $pattern_{kanan}$. Jika (u, v) merupakan sebuah faktorisasi dari *pattern*, maka sebuah pengulangan di (u, v) adalah sebuah kata w, sehingga dua persyaratan ini terpenuhi:

1. w adalah akhiran dari u atau u adalah akhiran dari w
2. w adalah awalan dari v atau v adalah awalan dari w



Gambar 18 Pembagian *Pattern* Pada Algoritma Crochemore-Perrin

```
Procedure CPSearch(
     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, k, shift, ell, memory, per, q, r: integer

Algoritma:
      i:= maxSuf(P, n, q)
      j:= maxSufTilde(P, n, r)
      shift:= 0

      if (i > j) then
         ell:= i
         per:= q
      else
         ell = j
         per = r
      endif

      k:=0
      while (P[k]=P[per+k]) do k:= k+1

      if (k>=ell+1) {
         i:= 0
         memory:= -1
         while (i <= m - n) do
             j:= max(ell,memory) + 1
             while (j < n and T[i+j]=P[j]) do j:=j+1
             if (j >= n) then
                 j:= ell
                 while (j > memory and T[i+j]=P[j]) do j:=j-1
                 if (j <= memory) then ketemu[i]=true
                 i:= i+per
                 memory:= n - per - 1
              else
                 i:= i+(j - ell)
                 memory:= -1
              endif
         endwhile
      else
         i:= 0
         per:= max((ell+1),(n-ell-1)) + 1;
         while (i <= m - n) do
             j:= ell + 1
             while (j < n and T[i+j]=P[j]) do j:=j+1
             if (j >= n) then
                 j:= ell
                 while (j >= 0 and T[i+j]=P[j]) do j:=j-1
                 if (j < 0) then ketemu[i]=true
                 i:= i+per
             else
                 i:= i+(j - ell)
             endif
         endwhile
      endif
```

Gambar 19 *Pseudocode* Algoritma Crochemore-Perrin

Dengan kata lain, kata *w* muncul di kedua sisi dari potongan *u* dan *v* dengan kemungkinan *overflow* di kedua sisi. Panjang dari pengulangan terkecil di (*u*, *v*) disebut periode lokal, dan dinotasikan dengan *r*(*u*, *v*).

Setiap faktorisasi dari (*u*, *v*) paling tidak mempunyai satu pengulangan. Dapat dilihat dengan mudah bahwa 1 $r(u, v)$ $|x|$.

Faktorisasi (*u*, *v*) dari *x* sehingga *r*(*u*, *v*) = *per*(*x*) disebut faktorisasi kritis dari *x*. Jika (*u*, *v*) adalah faktorisasi kritis dari *x*, maka pada posisi pada |*u*| di *x*, periode lokal dan periode global akan sama. Algoritma Crochemore-Perrin memilih faktorisasi kritis ($pattern_{kiri}$, $pattern_{kanan}$) sehingga $|pattern_{kiri}| < per(x)$ dan $|pattern_{kiri}|$ mempunyai nilai minimal.

2.6.1 **Kompleksitas**

Fase inisialisasi pada algoritma ini mempunyai kompleksitas waktu dan ruang O(*n*), sedang fase pencocokan dapat dilakukan dengan kompleksitas waktu O(*m*), dan pada kasus terburuk, algoritma ini melakukan 2*m*-*n* pencocokan karakter [CHA01]. Pada contoh kasus di subbab sebelunya, algoritma ini melakukan 20 pembandingan karakter pada m=*'24'* dan *n='6'*.

3 Visualisasi Algoritma Pencocokan String

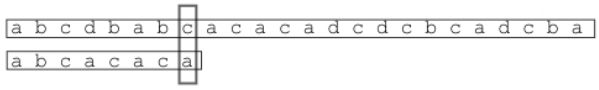
Tolak ukur dalam masalah tentang bagaimana membuat proses visualisasi pencocokan *string* oleh algoritma pencocokan *string* adalah kemajuan pemahaman pengguna setelah melihat visualisasi proses pencocokan *string*. Kemajuan pemahaman pengguna dari visualisasi yang dilakukan tentulah berbeda-beda, karena setiap pengguna mempunyai cara yang berbeda-beda dalam memahami sesuatu. Oleh karena itu, agar dapat bermanfaat bagi berbagai kalangan pengguna, visualisasi pencocokan *string* yang baik seharusnya dapat mengungkapkan secara detail dan intuitif apa yang dilakukan algoritma pencocokan *string*.

Semua Algoritma pencocokan *string* yang akan visualisasikan menggunakan mekanisme *sliding-windows* yang telah dijelaskan di subbab 2.1.1. Dalam mekanisme tersebut, langkah-langkah yang dilakukan oleh sebuah algoritma dibagi menjadi *attempt*, serta pembandingan-pembandingan karakter yang dilakukan di dalam setiap *attempt*.

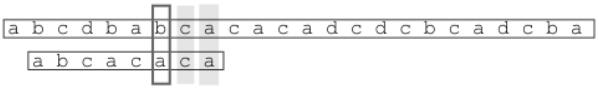
3.1 Animasi Pattern yang Bergerak Terhadap Teks

Untuk membuat visualisasi algoritma pencocokan *string* yang mengacu pada mekanisme *sliding-windows*, sebuah *pattern* harus digambarkan bergerak maju terhadap teks untuk melakukan sebuah *attempt*. Sebuah *attempt* yang satu dapat dibedakan dengan *attempt* yang lain dengan melihat posisi *pattern* terhadap teks. Dengan demikian, visualisasi yang membuat *pattern* bergerak terhadap teks akan dapat memudahkan pemahaman akan hal tersebut.

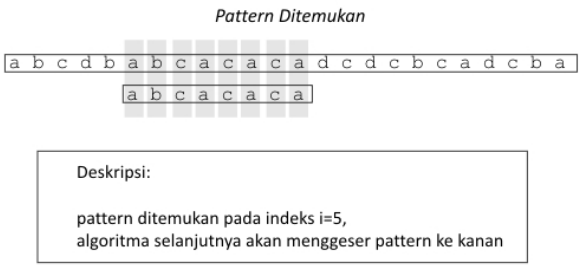
Dan di dalam setiap *attempt*, algoritma akan membandingkan sebuah karakter di *pattern* dengan karakter yang sejajar di teks. Untuk itu, dalam visualisasi yang akan dibangun, karakter-karakter yang sedang dibandingkan harus ditandai secara jelas, seperti yang dicontohkan pada Gambar 23, dimana karakter ‘c’ sedang dibandingkan dengan karakter ‘a’.



Gambar 20 Penandaan Karakter yang Sedang Dibandingkan



Gambar 21 Penandaan Karakter yang Telah Dibandingkan



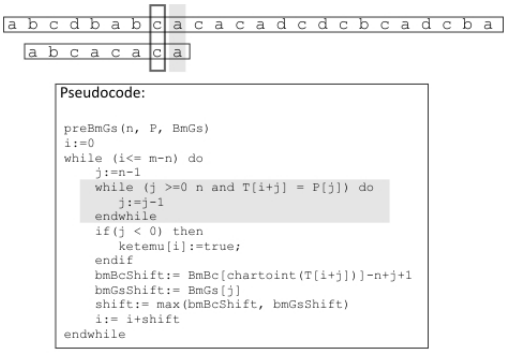
Gambar 22 Pemberitahuan dan Pendeskripsian Penemuan *Pattern*

Ketika dua karakter dinyatakan cocok dalam sebuah pembandingan, maka algoritma akan membandingkan karakter-karakter selanjutnya. Karakter-karakter yang telah dibandingkan seharusnya divisualisasikan secara berbeda dari karakter-karakter lainnya. Gambar 24mencontohkan sebuah visualisasi yang menandai karakter-karakter yang telah dibandingkan. Pada gambar tersebut, dua pembandingan karakter sebelumnya dinyatakan cocok, dan kini algoritma sedang membandingkan karakter ‘b’ dan karakter ‘a’.

Dalam pencocokan *string*, ada tiga kejadian istimewa yang terjadi, awal pencocokan, penemuan *pattern* di teks, dan berakhirnya pencocokan karena akhir dari teks telah dicapai. Kejadian-kejadian tersebut seharusnya diberitahukan dan dideskripsikan secara jelas dalam visualisasi. Selain itu visualisasi juga dapat memberikan deskripsi yang jelas terhadap kejadian lain, baik itu penggeseran *pattern* maupun pencocokan karakter. Gambar 25 mengilustrasikan pemberitahuan penemuan *pattern* beserta deskripsinya.

3.2 Visualisasi Bagian Algoritma Yang Tereksekusi

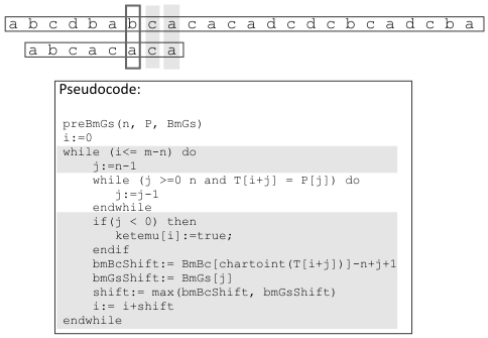
Dalam setiap *attempt* dan pembandingan karakter, sebuah algoritma pencocokan *string* melakukan eksekusi pada bagian tertentu dari algoritma tersebut. Sebagai contoh, Gambar 25 akan menampilkan sebuah pembandingan karakter dalam pencocokan *string* dengan algoritma Boyer-Moore, dan bagian algoritma yang sedang dieksekusi. Bagian algoritma yang sedang dieksekusi ditandai dengan warna abu-abu.



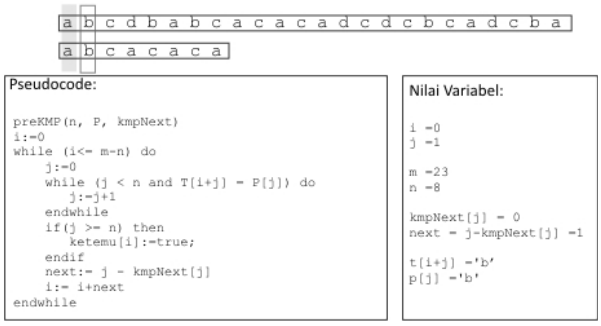
Gambar 23 Bagian Algoritma yang Dieksekusi Ketika Karakter Cocok

Dan untuk langkah lain, sebuah algoritma mungkin mengeksekusi bagian lain dari algoritma tersebut. Gambar 26 menampilkan sebuah langkah pencocokan *string* yang dilakukan juga dengan algoritma Boyer-Moore, dengan teks dan *pattern* yang sama. Namun kali ini, algoritma sedang

menemukan ketidakcocokan pada pencocokan karakter.



Gambar 24 Bagian Algoritma yang Dieksekusi Ketika Karakter Tidak Cocok.



Gambar 27 Nilai Variabel-Variabel Algoritma Knuth-Morris-Pratt

3.3 Nilai Variabel Dalam Sebuah Langkah

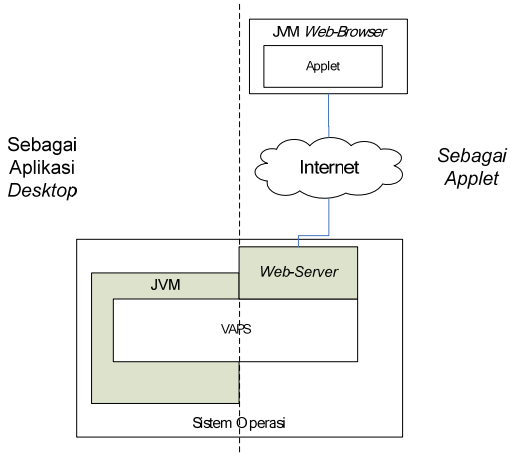
Ketika melakukan eksekusi dalam sebuah langkah, sebuah algoritma memiliki nilai-nilai pada variabelnya. Gambar 27 akan mengilustrasikan algoritma Knuth-Morris-Pratt ketika membandingkan dua karakter. Dalam gambar tersebut juga diuraikan nilai setiap variabel yang dimiliki algoritma pada langkah yang sedang dijalankan.

4 Pengembangan Perangkat Lunak

Perangkat lunak yang akan dibangun adalah sebuah aplikasi *desktop* yang mampu memvisualisasikan beberapa algoritma pencocokan *string* dan dibangun pada *platform* Java. Agar lebih portabel, perangkat lunak ini juga dapat dijalankan sebagai sebuah Java *applet*. Perangkat lunak ini memvisualisasikan pencocokan *string* oleh algoritma *brute force*, Knuth-Morris-Pratt, Boyer-Moore, Turbo Boyer-Moore, Colussi, dan Crochemore-Perrin. Perangkat lunak ini selanjutnya disebut VAPS.

4.1 Arsitektur Perangkat Lunak

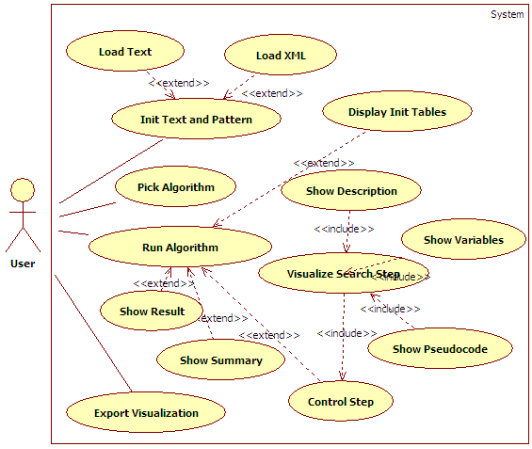
VAPS merupakan sebuah aplikasi *desktop* yang akan dijalankan di atas Java Virtual Machine (JVM). VAPS juga dapat dijalankan sebagai aplikasi *client-side* berbasis web yang menggunakan Java *applet*. Untuk dijalankan sebagai *applet*, VAPS disimpan di sebuah *web-server* dan dijalankan sepenuhnya oleh JVM pada *web-browser*. Arsitektur VAPS dapat dilihat pada Gambar 28..



Gambar 28 Arsitektur VAPS

4.2 Kebutuhan Perangkat Lunak

Dari analisis terhadap kebutuhan fitur-fitur sistem, didefinisikan 13 *use-case* yang terkait dengan fitur-fitur tersebut. Definisi *use-case* dapat dilihat pada Gambar 29.

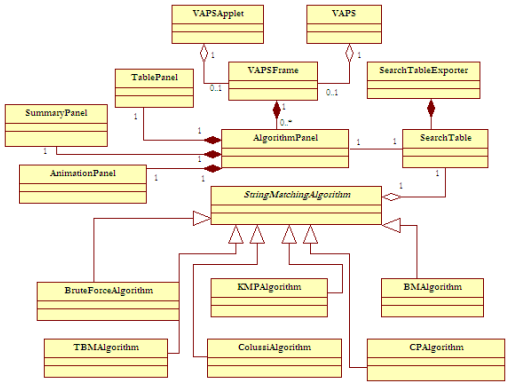


4.3 Analisis Kelas

Dari diagram *use-case* dan skenario-skenario yang ada dapat dianalisis objek-objek yang akan diimplementasikan. Pola objek ini dirancang dalam bentuk kelas yang mempunyai asosiasi satu sama lain membentuk sebuah diagram kelas. Gambar 30 mengilustrasikan diagram kelas analisis VAPS.

Karena VAPS dapat dijalankan sebagai aplikasi *desktop* maupun *applet,* maka didefinisikan dua kelas utama untuk menjalankan VAPS, kelas `VAPS` yang mempunyai fungsi `main` akan menangani eksekusi VAPS sebagai aplikasi *desktop*, dan kelas `VAPSApplet` yang mempunyai fungsi `init` yang menangani eksekusi VAPS sebagai *applet.*

Generalisasi kelas dilakukan pada kelas-kelas algoritma pencocokan *string*, kelas `StringMatchingAlgorithm` merupakan kelas abstrak yang akan diturunkan oleh kelas-kelas yang mengimplementasikan algoritma pencocokan *string*, yaitu kelas `BruteForceAlgorithm`, `KMPPrattAlgorithm`, `BMAlgorithm`, `TBMAlgorithm`, `ColussiAlgorithm`, dan `CPAlgorithm`. Kelas-kelas turunan tersebut selanjutnya harus mengimplementasikan fungsi-fungsi abstrak yang didefinisikan oleh kelas `StringMatchingAlgorithm`. Kelas-kelas algoritma selanjutnya menghasilkan sebuah objek dari kelas `SearchTable` yang akan digunakan oleh kelas antarmuka `AlgorithmPanel`.



Gambar 25 Diagram Kelas

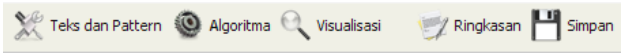
4.4 Implementasi Antarmuka

Implementasi antarmuka dilakukan sesuai perancangan antarmuka pada subbab 4.2.3, dan dibangun dengan bantuan IDE (*Integrated Development Environment*) NetBeans 6.5, yang memudahkan dalam pembuatan *Graphical User Interface* (GUI). Berikut ini akan dijelaskan secara rinci hasil implementasi antarmuka VAPS.

4.4.1 Implementasi Antarmuka MainView

Tampilan antarmuka MainView secara keseluruhan dapat dilihat pada bagian Lampiran E. Pada subbab ini akan dijelaskan fungsi-fungsi objek yang ada pada antarmuka MainView. Antarmuka MainView ini dapat dibagi ke dalam empat kelompok besar yaitu:

Kelompok *Toolbar*



Gambar 26 *Screenshot Toolbar*

Gambar 31 merupakan *screenshot* dari toolbar, pada *toolbar* terdapat tombol-tombol:

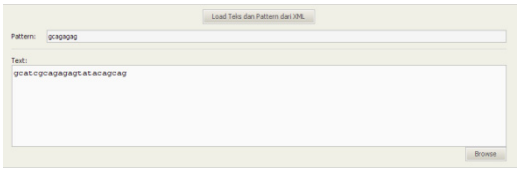
1. Visualisasi, yang akan menjalankan algoritma pencocokan *string* yang dipilih.
2. Teks dan Pattern, yang akan menampilkan panel konfigurasi teks dan *pattern*.
3. Algoritma, yang akan menampilkan panel konfigurasi algoritma.
4. Ringkasan, yang akan menampilkan ringkasan dari seluruh visualisasi yang dilakukan.

Kelompok Konfigurasi Teks dan *Pattern*.

Pada kelompok ini terdapat sebuah panel yang akan muncul bila pengguna menekan tombol "Teks dan Pattern" pada *toolbar*, dan berguna untuk inisialisasi teks dan *pattern*. Gambar 32 merupakan *screenshot* dari panel konfigurasi teks dan pattern, pada panel ini terdapat:

1. Sebuah tombol "Load Teks dan Pattern dari XML" yang bila ditekan akan mengeluarkan sebuah jendela untuk memilih *file* XML yang ingin di-*load*.
2. Sebuah *textbox* yang menampilkan *pattern* dimana pengguna juga bisa langsung mengeditnya.
3. Sebuah *textarea* yang menampilkan teks dimana pengguna juga bisa langsung mengeditnya.

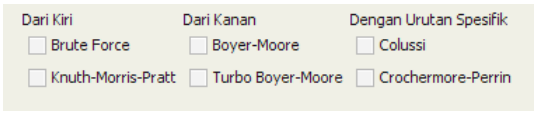
4. Sebuah tombol “Browse” yang bila ditekan akan mengeluarkan sebuah jendela untuk memilih *file* untuk memilih teks yang ingin di-*load*.



Gambar 27 *Screenshot* Panel Konfigurasi Teks dan *Pattern*

Kelompok Konfigurasi Algoritma

Pada kelompok ini terdapat sebuah panel yang akan muncul bila pengguna menekan tombol “Algoritma” pada *toolbar*. Gambar 33 menampilkan *screenshot* panel ini. Pada panel ini akan ditampilkan *checkbox* untuk memilih algoritma apa saja yang diinginkan pengguna untuk melakukan visualisasi. *Checkbox* algoritma-algoritma dibagi atas klasifikasi arah darimana algoritma tersebut melakukan pencocokan.



Gambar 28 *Screenshot* Panel Konfigurasi Algoritma

Kelompok *Statusbar*

Statusbar akan menampilkan status dari aplikasi. Gambar 34 merupakan *screenshot* dari statusbar, pada statusbar terdapat:

1. Sebuah teks yang menampilkan deskripsi dari *task* yang sedang dijalankan aplikasi.
2. Sebuah *progressbar* yang menampilkan *progress* dari *task* yang sedang dijalankan.

Gambar 29 *ScreenShot Statusbar*

4.4.2 Implementasi Antarmuka AlgorithmPanel

Antarmuka AlgorithmPanel akan muncul untuk menampilkan visualisasi dengan algoritma yang dipilih untuk teks dan *pattern* yang diinisialisasi. Antarmuka AlgorithmPanel ini dapat dibagi ke dalam tiga *tab* yaitu:

***Tab* Animasi**

Tampilan antarmuka *tab* animasi secara keseluruhan dapat dilihat pada Lampiran D pada *tab* ini terdapat beberapan panel, yaitu:

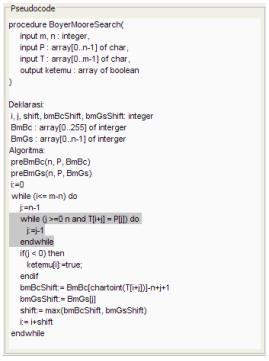
Panel Visualisasi Teks dan *Pattern*

Panel ini merupakan tempat menganimasikan teks dan *pattern* pada setiap langkah yang dilakukan algoritma. Gambar 35 merupakan empat *screenshot* dari panel ini. Empat *screenshot* tersebut adalah, pada bagian atas *screenshot* penggaris pada visualisasi, sedangkan pada bagian bawah, dari kiri ke kanan, *screenshot* ketika algoritma menggeser *pattern*, *screenshot* ketika algoritma membandingkan karakter, dan *screenshot* ketika algoritma menemukan kecocokan *pattern* di teks.

Gambar 30 *Screenshot* dari Panel Visualisasi Teks dan *Pattern*.

Panel *Pseudocode*

Panel ini menampilkan *pseudocode* algoritma, bagian algoritma yang sedang dieksekusi ditampilkan dengan latar belakang abu-abu. Gambar 36 merupakan *screenshot* dari panel ini.

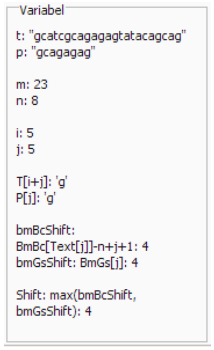


Gambar 31 Screenshot Panel Pseudocode

Panel Variabel

Panel ini menampilkan nilai dari variabel yang dimiliki algoritma pada langkah yang dijalankan, variabel-variabel ditampilkan berdasarkan

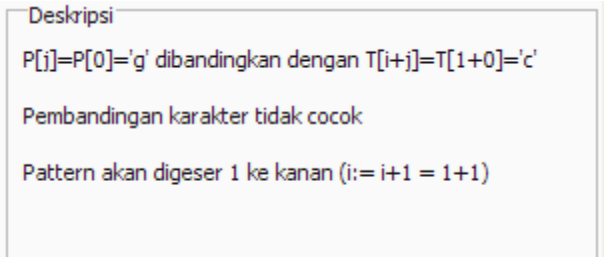
kelompoknya. Gambar 37 merupakan *screenshot* dari panel ini.



Gambar 37 ***Screenshot* Panel Variabel**

Panel Deskripsi

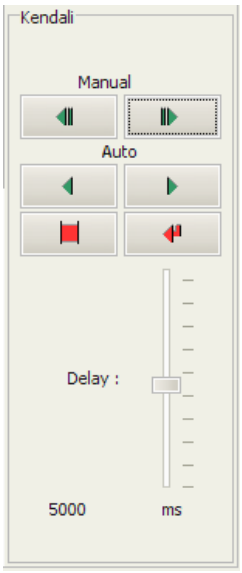
Panel ini menampilkan deskripsi atas apa yang dilakukan algoritma pada langkah yang sedang dieksekusi. Gambar 38 merupakan *screenshot* dari panel ini.



Gambar 38 ***Screenshot* Panel Deskripsi**

Panel Kendali

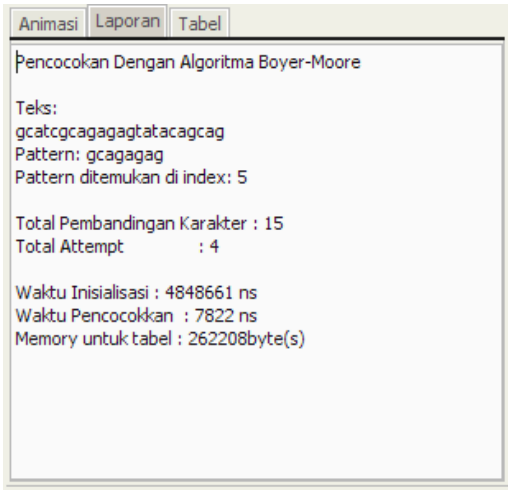
Panel ini digunakan untuk mengendalikan langkah yang ingin dieksekusi. Gambar 39 merupakan *screenshot* dari panel kendali. Komponen dari panel ini dibagi menjadi dua kelompok.Kelompok pertama adalah kelompok kendali manual, pada kelompok ini terdapat dua tombol yang masing-masing digunakan untuk memundurkan atau memajukan satu langkah.Kelompok kedua adalah kelompok kendali automatis, pada kelompok ini terdapat empat tombol untuk kendali automatis, dan sebuah *slider* untuk mengatur kecepatan pergerakan per langkahnya.



Gambar 32 Screenshot Panel Kendali

***Tab* Laporan**

Pada *tab* ini akan ditampilkan laporan pencocokan *pattern* di teks dengan algoritma yang dipilih. Gambar 40 merupakan *screenshot* dari *tab* laporan, pada *tab* ini terdapat informasi:

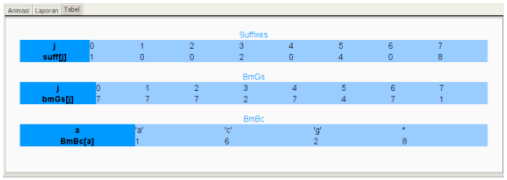


Gambar 33 *Screenshot* Tab Laporan

1. Algoritma yang dipakai, beserta teks dan *pattern* yang digunakan.
2. Total pembandingan karakter dan total *attempt* yang dilakukan untuk menyelesaikan pencocokan.
3. Waktu inisialisasi dan waktu pencocokan yang diperlukan untuk melakukan pencocokan.
4. Besar ruang memori yang dibutuhkan untuk penyimpanan di tabel.

***Tab* Tabel**

Pada *tab* ini akan ditampilkan isi tabel yang dihasilkan algoritma pada fase inisialisasi. Isi dari *tab* ini akan berbeda-beda untuk setiap algoritma.. Gambar 34 merupakan *screenshot* dari *tab* tabel.



Gambar 34 *Screenshot* Tab Tabel

5 Kesimpulan dan Saran

Pada bagian ini dipaparkan mengenai kesimpulan dan saran yang berkaitan dengan makalah yang ditulis.

5.1 Kesimpulan

Kesimpulan yang didapat dari Makalah ini adalah:

1. Visualisasi terhadap pencocokan *string* dapat dilakukan dan dapat membantu pemahaman akan cara kerja algoritma pencocokan *string* yang beragam.
2. Visualisasi algoritma haruslah dilakukan secara detail dengan menampilkan *pseudocode* dan bagian yang dieksekusinya, animasi teks dan *pattern*, serta nilai-nilai variabel pada sebuah langkah.
3. Karena visualisasi dilakukan terhadap langkah demi langkah dari algoritma, maka aplikasi pemvisualisasi seharusnya dapat membiarkan pengguna untuk memilih langkah yang diinginkannya.
4. Algoritma Boyer-Moore dan Turbo Boyer-Moore unggul dalam jumlah pembandingan karakter di tiga contoh kasus yang dilakukan.

5.2 Saran

Saran untuk pengembangan lebih lanjut dari Makalah ini:

1. Membangun sebuah *framework* yang dapat memudahkan pembuatan aplikasi untuk memvisualisasikan algoritma-algoritma secara umum, baik itu algoritma sorting, pencocokan string, algoritma aritmatika, dan sebagainya.
2. Membuat aplikasi VAPS ektensibel, sehingga untuk menambahkan algoritma yang divisualisasikan hanya diperlukan menambahkan sebuah file yang secara formal mendeskripsikan apa yang dilakukan algoritma tersebut. Karena pada VAPS, untuk menambahkan sebuah algoritma pencocokan string, masih diperlukan menambahkan sebuah kelas turunan dari StringMatchingAlgorithm.
3. Membandingkan kinerja algoritma berdasarkan jumlah instruksi pada CPU yang dilakukan.

6 Daftar Referensi

[BRE92] Breslauer, D. 1992. Efficient String Algorithmics, PhD Thesis, Report CU-024-92. CS Department, Columbia University.

[BOY77] Boyer, R. S, and J. S. Moore. 1977. A Fast String Searching Algorithm. Communications of the ACM.

[CHA01] Charras, C, and T. Lecroq. 2001. Handbook of Exact String Matching Algorithm. Oxford University Press.

[CHA98] Charras C, T. Lecroq, and J. D. Pehousek. 1998. A Very Fast String Matching Alglorithm for Small Alphabets and Long Patterns. Laboratoire D'informatique de Rouen.

[COO72] COOK S. A. 1972. Linear Time Simulation of Deterministic Two-Way Pushdown Automata. Information Processing 71.

[KNU68] Knuth, D. E. The Art of Computer Programming. Volume 1: Fundamental Algorithms. Addison-Wesley.

[KNU77] Knuth, D, J. Morris, and V. Pratt. 1977. Fast Pattern Matching in Strings. SIAM Journal of Computing.

[MER08] Merriam-Webster Online Dictionary

http://www.merriam-webster.com/dictionary/algorithm

Tanggal akses: 18 Juli 2008

[PRE97] Pressman, Roger S. 1997. Software Engineering (A Practitional Approach). McGraw-Hill.

[ROB03] Robinson, Matthew, and Pavel Vorobiev. 2003. Swing Second Edition. Manning.

[SED83] Sedgewick, R. 1983. Algorithms. Addison-Wesley.

[SUN08] Java 2 Platform Standard Edition 6.0 API Reference

http://java.sun.com/api/reference/docs/index.html

Tanggal akses: 20 Desember 2008