Deteksi Kemiripan Kode Program dengan Metode *Preprocessing* dan Penghitungan *Levenshtein Distance*

Dani[1], Timotius Grady Limandra[2], Lie Roberto Eliantono Adiseputra[3]

Program Studi Informatika
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail: if14060@students.if.itb.ac.id[1], if14082@students.if.itb.ac.id[2],
if14128@students.if.itb.ac.id[3]

Abstrak

Ada berbagai teknik untuk mendeteksi kemiripan kode program. Metode sederhana yang cukup mangkus untuk melakukan hal ini adalah dengan menghitung *levenshtein distance* dari kedua kode program yang diuji. Namun, metode ini hanya sangkil mendeteksi kemiripan program yang singkat dan hanya mengalami sedikit perubahan leksikal. Dengan melakukan *preprocessing* terhadap kode program, tingkat akurasi pendeteksian dapat ditingkatkan secara signifikan.

***Kata kunci**: preprocessing, source code similarity, string matching, edit distance, levenshtein distance, plagiarism*

1. Pendahuluan

Dalam ranah ilmu informatika, kemampuan untuk mendeteksi kemiripan kode program sangat dibutuhkan. Salah satu aplikasi yang paling populer atas teknik deteksi ini adalah untuk mendeteksi terjadinya praktik plagiarisme dalam lingkungan akademis serta kompetisi pemrograman. Selain itu, dengan sedikit modifikasi, teknik ini dapat pula diterapkan untuk mendeteksi kemiripan antarbagian program untuk memudahkan proses modularisasi program yang sudah ada.

Salah satu metode yang cukup mangkus dalam melakukan deteksi kemiripan kode program adalah dengan melakukan penghitungan *levenshtein distance* dari dua kode program**,** seperti yang diterapkan pada program *diff* pada sistem operasi *NIX. Sayangnya, tingkat akurasi metode ini kurang baik dalam mendeteksi kemiripan program yang berskala cukup besar (terdiri atas lebih dari sekitar 100 baris) maupun program yang telah mengalami banyak perubahan leksikal.

Untuk meningkatkan akurasi teknik tersebut, kami mengembangkan sebuah algoritma *preprocessing* untuk ditambahkan sebelum dilakukan penghitungan *levenshtein distance* kedua kode yang diuji.

2. Transformasi Kode Program

Ada banyak cara yang dapat diterapkan untuk melakukan transformasi atas suatu kode program. Dua strategi paling umum untuk melakukan transformasi ini adalah:

1. Transformasi leksikal: untuk melakukan transformasi ini, tidak diperlukan pengetahuan akan bahasa pemrograman yang digunakan dalam kode program tersebut. Transformasi jenis ini meliputi antara lain: pengubahan nama *identifier*, penambahan/penghapusan komentar serta pengubahan format/indentasi program.
2. Transformasi struktural: untuk melakukan transformasi ini, diperlukan pengetahuan akan bahasa pemrograman yang digunakan dalam kode program tersebut. Metode ini sangat bergantung pada keahlian seseorang dalam menggunakan bahasa pemrograman tersebut. Pendekatan ini meliputi antara lain: pengubahan bentuk pengulangan yang digunakan (*repeat..until* menjadi *while..do*), penggunaan *switch..case* untuk menangani *if* bersarang, mengubah potongan kode menjadi prosedur (atau sebaliknya) serta pengubahan susunan dari pernyataan yang tidak mempengaruhi jalannya program secara keseluruhan.

3. Preprocessing

Sebagian besar teknik deteksi kemiripan kode program saat ini melakukan *preprocessing* untuk menghasilkan tingkat akurasi yang lebih baik. Algoritma *preprocessing* yang dibahas pada [5] didasarkan pada pembangkitan indeks dari berkas kode program yang dibandingkan. Meski algoritma tersebut memiliki performa yang cukup baik, algoritma tersebut melibatkan struktur data khusus yaitu *sparse suffix tree* yang mungkin sulit dipahami

dan diimplementasi oleh para *programmer* pemula dan menengah.

Algoritma *preprocessing* yang kami kembangkan berbasis pada pendekatan pragmatis atas transformasi kode yang dijelaskan pada bagian 2.

Secara umum, algoritma *preprocessing* ini dapat dituliskan sebagai berikut:

prosedur *preprocess* (**input** *berkas1*, *berkas2*: **berkas** kode program; **output** *s1*, *s2*: **string**)
1. **untuk setiap** berkas **lakukan**
   1.1. **hapus** semua komentar
   1.2. **ubah** semua string menjadi string kosong
   1.3. **ubah** semua identifier konstanta menjadi bentuk literalnya, kemudian hapus deklarasi konstanta
   1.4. **ubah** semua identifier tipe bentukan menjadi deklarasi tipe sederhananya, kemudian hapus deklarasi tipe bentukan
   1.5. **ubah** semua konstruksi kalang menjadi bentuk yang seragam
   1.6. **ubah** semua bentuk *case* menjadi bentuk *if* biasa
   1.7. **tambahkan** penanda blok (*begin-end*) pada kalang maupun *if* yang hanya diikuti satu pernyataan
   1.8. **ubah** semua fungsi menjadi prosedur
   1.9. **untuk setiap** prosedur yang ada **lakukan**
      1.9.1. **cek** apakah prosedur tersebut rekursif
      1.9.2. jika tidak, **ganti** setiap pemanggilan prosedur dengan isi dari prosedur yang bersangkutan
   1.10. **reformat** kode program menjadi bentuk yang seragam
2. **untuk setiap** pernyataan pada berkas yang lebih sedikit jumlah barisnya (atau salah satu, jika jumlah baris kedua berkas sama) **lakukan**
   2.1. **cari** semua pernyataan yang mempunyai konstruksi yang berkesesuaian dengan pernyataan baris ini pada berkas yang lain
      2.1.1. **untuk setiap** identifier yang terdapat pada pernyataan tersebut, **catat** nama identifier dengan posisi yang berkesesuaian yang terdapat pada berkas lain tersebut sebagai kandidat pengganti, **catat** juga berapa kali identifier tersebut diajukan sebagai kandidat
      2.1.2. **untuk setiap** identifier, **ganti** nama identifier tersebut dengan identifier pada kandidat pengganti yang mempunyai jumlah pengajuan paling banyak
3. **untuk setiap** pernyataan berkas yang lebih sedikit jumlah barisnya (atau salah satu, jika jumlah baris kedua berkas sama) **lakukan**
   3.1. **pilih** sebuah nama unik untuk pernyataan ini
   3.2. **cari** semua pernyataan di berkas lain yang persis sama dengan pernyataan saat ini
      3.2.1. **untuk setiap** pernyataan yang ditemukan, **ganti** pernyataan tersebut dengan nama unik
   3.3. **ganti** pernyataan dengan nama uniknya
4. **untuk setiap** pernyataan pada berkas yang lebih banyak jumlah barisnya yang belum diubah menjadi nama unik, **pilih** sebuah nama unik kemudian gantikan
5. **konkatenasi** tiap pernyataan pada masing-masing berkas, **simpan** hasilnya ke *s1* (untuk *berkas1*) dan *s2* (untuk *berkas2*)

Pada praktiknya, implementasi tahapan-tahapan tersebut bergantung pada bahasa yang digunakan dalam kode program. Dalam bagian 6 makalah ini, digunakan program dalam bahasa Pascal sebagai contoh.

Prosedur *preprocess* di atas melibatkan cukup banyak pencocokan string, oleh sebab itu, algoritma pencocokan string yang mangkus juga diperlukan untuk menghasilkan performa global yang baik. Algoritma-algoritma pencocokan string standar seperti Knuth-Morris-Pratt [4, upa-bab 10.2], Horspool, Boyer-Moore [3, upa-bab 7.2], dan Rabin-Karp [2, upa-bab 34.2] sudah cukup baik untuk keperluan ini.

Langkah 1.5 – 1.9 berhubungan dengan transformasi struktural. Karena pada praktiknya, transformasi struktural jauh lebih jarang dilakukan daripada transformasi leksikal, langkah-langkah tersebut dapat diabaikan apabila dibutuhkan performa yang lebih gegas.

4. *Levensthein Distance*

Levensthein distance dari dua buah string *s1* dan *s2* didefinisikan sebagai jumlah minimum mutasi-mutasi titik yang dibutuhkan untuk mengubah *s1* menjadi *s2*. Sebuah mutasi titik dapat berupa:
- pengubahan karakter
- menambahan karakter
- penghapusan karakter

Fungsi penghitungan *levenshtein distance* dari *s1* dan *s2* – *d*(*s1*, *s2*) – dapat dirumuskan dalam persamaan rekursif [1]:

$$\begin{cases} \mathrm{d}(\varepsilon,\varepsilon) & = 0 \\ \mathrm{d}(s,\varepsilon) = \mathrm{d}(\varepsilon,s) & = |s| \\ \mathrm{d}(s1+ch1, s2+ch2) = \min(\mathrm{d}(s1,s2) + & \\ \qquad (ch1 == ch2\ ?\ 0 : 1), & \\ \qquad \mathrm{d}(s1+ch1, s2)+1, & \\ \qquad \mathrm{d}(s1, s2+ch2)+1) & \end{cases}$$

… (1)

Jelas bahwa dari persamaan rekursif (1) tersebut kita dapat mendapatkan sebuah fungsi rekursif untuk menghitung nilai *levensthein distance* yang memiliki kompleksitas $O(3^n)$, dengan $n = \max(|s1|, |s2|)$.

Dengan sedikit observasi, kita dapat melihat bahwa d(*s1*, *s2*) hanya bergantung dari d(*s1'*, *s2'*) di mana $|s1| > |s1'|$ dan $|s2| > |s2'|$. Dengan demikian, kita dapat menggunakan teknik *dynamic programming* untuk memperbaiki kompleksitas waktu penghitungan *levenshtein distance* tersebut.

Dengan bantuan sebuah matriks dua dimensi, *m*[0..|*s1*|, 0..|*s2*|], kita dapat membuat fungsi berikut:

```
fungsi levenstheinDistance (input s1, s2: string)
variabel m: matriks integer [0..|s1|, 0..|s2|]
  1. m[0, 0] = 0
  2. i menelusur 1..|s1|
     2.1. m[i, 0] = i
  3. j menelusur 1..|s2|
     3.1. m[0, j] = j
  4. i menelusur 1..|s1| dan j menelusur 1..|s2|
     4.1. m[i,j] = min(m[i-1, j-1] + s1[i]==s2[j]
          ? 0:1, m[i-1, j] +1, m[i, j-1] + 1)
  5. kembalikan nilai m[|s1|, |s2|]
```

Kompleksitas waktu fungsi di atas adalah $O(|s1||s2|) \approx O(n^2)$, $n = \max(|s1|, |s2|)$, demikian pula kompleksitas memorinya. Dari pengamatan lebih lanjut, terlihat bahwa untuk setiap lelaran, elemen matriks yang diakses hanyalah matriks pada baris saat ini dan 1 baris sebelumnya. Dari fakta ini, untuk menghemat memori, dapat dilakukan penyimpanan hasil secara *rolling* – baris matriks yang disimpan hanya 2 baris terakhir saja. Perubahan ini mengubah kompleksitas memori menjadi $O(n)$.

Kode program fungsi penghitungan *levenshtein distance* yang diimplementasikan dengan bahasa pemrograman Java tersedia di **http://students.if.itb.ac.id/~if14060/stmik.java**

5. Algoritma Keseluruhan

Karena fungsi levenstheinDistance menghitung berapa banyak perubahan yang dibutuhkan untuk mengubah s1 menjadi s2, nilai kemiripan (*similarity*) dapat didefinisikan sebagai:

$$similarity = \left(1 - \frac{levenstheinDistance(s1, s2)}{\max(|s1|, |s2|)}\right) \times limit$$

… (2)

Konstanta *limit* digunakan untuk mengatur batas atas skala distribusi nilai kemiripan (batas bawah selalu bernilai 0)

Dengan demikian, algoritma keseluruhan untuk melakukan deteksi kemiripan adalah:

```
program codeSimilarityDetector (input berkas1, berkas2: berkas kode program)
variabel s1, s2: string; similarity: float
konstanta limit: float = 90
        preprocess (berkas1, berkas2, s1, s2)
        similarity = (1 - levenstheinDistance(s1, s2)/max(|s1|, |s2|)) * limit
        output(similarity)
```

Untuk kepentingan praktis, dapat pula ditambahkan sebuah konstanta *treshold* yang menyatakan batas bawah nilai *similarity* yang diperkenankan untuk menyatakan kedua kode program mirip. Dengan kata lain, dua kode program dengan nilai *similarity* < *treshold* dinyatakan tidak mirip, sedangkan sisanya dianggap mirip. Penambahan ini utamanya diperlukan apabila algoritma ini digunakan untuk mendeteksi plagiarisme.

6. Contoh Instansiasi

Untuk memudahkan pemahaman, dapat dilihat contoh berikut. Diberikan dua buah kode program dalam bahasa Pascal:

berkas1

```
program cari_maksimum;

const
        Nmax = 5; { banyak elemen
array maksimum }
type
        tabint = array [1..Nmax] of
integer;
var
        T : tabint;    m : integer;

function max (var T : tabint) :
integer;  { mengembalikan nilai
maksimum dalam array }
var
        i,max1 : integer;
begin
        max1 := T[1];  i := 2;
while i <= Nmax do
        begin
        if (max1 < T[i]) then
        begin
                max1 := T[i];
        end;
                i := i + 1;
        end;
        max := max1;
end;

procedure input (var T : tabint);
var
        i,j : integer;
begin
        writeln('Masukan nilai tiap
element array');
        for i:=1 to Nmax do
        begin
                readln(j);
                T[i] := j;
```

```
        end;
end;

begin
        input(T);
        writeln('Nilai      maksimum
array tabint');
        m := max(T);
        writeln(m);
end.
```

berkas2

```
program carimaks;

type
  tabelinteger  =  array  [1..5]  of
integer;
var
  tabel: tabelinteger;
  max: integer;
  j: integer;
  i: integer;

procedure Nmax (T:tabelinteger; var
m:integer);
var
  j, temp : integer;
begin
  temp := T[1];
  j := 2;
  while j <= 5 do
  begin
    if (temp < T[j]) then
      temp := T[j];
    j := j + 1;
  end;
  m := temp;
end;

begin
  for i:=1 to 5 do
  begin
    writeln('N[', i,  '] = ');
    readln(j);
    tabel[i] := j;
  end;

  writeln('MAX = ');
  Nmax(tabel,max);
  writeln(max);
end.
```

Secara konsep, kedua program di atas melakukan hal yang sama, yaitu mencari nilai maksimal dari 5 buah bilangan.

Hasil langkah-per-langkah eksekusi algoritma pendeteksian kemiripan kode program:

1. Eksekusi prosedur *preprocess* bagian 1.1 – 1.4

berkas1

```
program cari_maksimum;

var
        T   :   array   [1..5]   of
integer;       m : integer;

function max (var T : array [1..5]
of integer) : integer;
var
        i,max1 : integer;
begin
        max1 := T[1];  i := 2;
while i <= 5 do
        begin
        if (max1 < T[i]) then
        begin
                max1 := T[i];
        end;
                i := i + 1;
        end;
        max := max1;
end;

procedure  input  (var  T  :  array
[1..5] of integer);
var
        i,j : integer;
begin
        writeln('');
        for i:=1 to 5 do
        begin
                readln(j);
                T[i] := j;
        end;
end;

begin
        input(T);
        writeln('');
        m := max(T);
        writeln(m);
end.
```

berkas2

```
program carimaks;

type
  tabelinteger  =  array  [1..5]  of
integer;
var
  tabel: array [1..5] of integer;
  max: integer;
  j: integer;
  i: integer;

procedure Nmax (T: array [1..5] of
integer; var m:integer);
var
  j, temp : integer;
begin
  temp := T[1];
  j := 2;
  while j <= 5 do
  begin
    if (temp < T[j]) then
      temp := T[j];
    j := j + 1;
  end;
  m := temp;
end;

begin
  for i:=1 to 5 do
  begin
    writeln('', i,  '');
    readln(j);
    tabel[i] := j;
  end;

  writeln('');
  Nmax(tabel,max);
  writeln(max);
end.
```

2. Eksekusi prosedur *preprocess* bagian 1.5 – 1.10
   Dalam contoh ini, semua konstruksi kalang diubah menjadi bentuk *while..do*. Tidak terdapat

bentuk *case* sehingga langkah 1.6 tidak mengubah apapun.

berkas1

```
program cari_maksimum;
var
T:array[1..5] of integer;
m:integer;
i,max1:integer;
i,j:integer;
begin
writeln('');
i:=1;
while i<=5 do
begin
readln(j);
T[i]:=j;
i:=i+1;
end;
writeln('');
max1:=T[1];
i:=2;
while i<=5 do
begin
if (max1<T[i]) then
begin
max1:=T[i];
end;
i:=i+1;
end;
m:=max1;
writeln(m);
end.
```

berkas2

```
program carimaks;
var
tabel:array[1..5] of integer;
max:integer;
j:integer;
i:integer;
j,temp:integer;
begin
i:=1;
while i<=5 do
begin
writeln('',i,'');
readln(j);
tabel[i]:=j;
i:=i+1;
end;
writeln('');
temp:=tabel[1];
j:=2;
while j<=5 do
begin
if (temp<tabel[j]) then
begin
temp:=tabel[j];
end;
j:=j+1;
end;
max:=temp;
writeln(max);
end.
```

3. Eksekusi prosedur *preprocess* bagian 2

berkas1

```
program carimaks;
var
tabel:array[1..5] of integer;
max:integer;
j,temp:integer;
j,j:integer;
begin
writeln('');
j:=1;
while j<=5 do
begin
readln(j);
tabel[j]:=j;
j:=j+1;
end;
writeln('');
temp:= tabel[1];
j:=2;
while j<=5 do
begin
if (temp<tabel[j]) then
begin
temp:=tabel[j];
end;
j:=j+1;
end;
max:=temp;
writeln(max);
end.
```

berkas2

```
program carimaks;
var
tabel:array[1..5] of integer;
max:integer;
j:integer;
i:integer;
j,temp:integer;
begin
i:=1;
while i<=5 do
begin
writeln('',i,'');
readln(j);
tabel[i]:=j;
i:=i+1;
end;
writeln('');
temp:=tabel[1];
j:=2;
while j<=5 do
begin
if (temp<tabel[j]) then
begin
temp:=tabel[j];
end;
j:=j+1;
end;
max:=temp;
writeln(max);
end.
```

4. Eksekusi prosedur *preprocess* bagian 3

Untuk kepentingan contoh, kita asumsikan karakter A – Z dan _ tidak digunakan sebagai nama identifier dalam program sehingga dapat digunakan sebagai nama unik. Pada praktiknya, nama unik bisa dipilih sebagai rangkaian karakter acak yang bukan merupakan identifier pada kedua kode program.

s1	ABCDEFGHIJGKLMNHOPJGQGRNMNSTN
s2	ABCDUVEGWXGYKZ_NHOPJGQGRNMNSTN

5. Penghitungan *Levenshtein Distance* dan Nilai Kemiripan

Setelah mendapatkan kedua string di atas, kita dapat dengan mudah menghitung *levenshtein distance*nya, yaitu 9. Dengan mengetahui |*s1*|= 29 dan |*s2*| = 30,

nilai kemiripan dapat dihitung. Nilai kemiripan kedua kode program di atas dengan *limit* 100% adalah (1 - (9/30)) * 100% = 70%.

Dengan melihat contoh di atas, jelas bahwa dengan melakukan *preprocessing* akan ada peningkatan akurasi hasil. Sebagai contoh nyata, identifier cari_maksimum dan carimaks yang pada intinya serupa akan dianggap sama (*levenshtein distance*nya 0) setelah dilakukan *preprocessing*, sementara jika tidak dilakukan *preprocessing*, kedua *identifier* tersebut akan dianggap berbeda.

7. Analisis Kompleksitas

7.1 Prosedur *Preprocess*

Kompleksitas waktu prosedur preprocess sangat bergantung pada algoritma pencocokan string yang digunakan. Untuk keperluan analisis, kita asumsikan yang terjadi pada setiap kali pencocokan string adalah kasus terburuk. Kompleksitas waktu kasus terburuk pada algoritma pencocokan string standar adalah O(*nm*), *n* adalah panjang string dan *m* adalah panjang substring. Batas atas nilai *m* adalah *n*, sehingga kompleksitas waktu tersebut dapat kita sederhanakan menjadi O(n^2). Meski terdapat beberapa tahapan *preprocessing* dan setiap tahapan memiliki kompleksitas waktu O(n^2) akibat penggunaan algoritma pencocokan string, jumlah tahapan adalah konstan – tidak bergantung pada *n* – sehingga kompleksitas waktu prosedur *preprocess* adalah O(n^2), *n* = panjang string = jumlah karakter kode program.

7.2 Fungsi *Levensthein Distance*

Seperti yang sudah dijelaskan di bagian 4, fungsi Levensthein Distance versi *dynamic programming* memiliki kompleksitas waktu O(n^2), *n* = max(|*s1*|, |*s2*|).

7.3 Algoritma Keseluruhan

Kompleksitas waktu algoritma secara keseluruhan merupakan penjumlahan dari kedua bagian utama algoritma, yaitu prosedur *preprocess* dan fungsi *levenstheinDistance*. Karena kompleksitas waktu prosedur preprocess bergantung pada banyaknya karakter pada kode program dan fungsi levenstheinDistance bergantung pada panjang string hasil *preprocessing* sementara panjang string selalu lebih kecil atau sama dengan (dan umumnya jauh lebih kecil dari) jumlah karakter kode program, dalam melakukan analisis kompleksitas waktu total, kita dapat mengabaikan kompleksitas waktu dari fungsi levenstheinDistance. Dengan demikian, kompleksitas waktu algoritma keseluruhan adalah O(n^2), *n* = jumlah karakter kode program.

8. Peningkatan Akurasi Lebih Lanjut

Untuk meningkatkan akurasi pendeteksian, kami menyarankan ditambahkannya beberapa hal berikut dalam prosedur *preprocessing*:

- peningkatan analisis struktural: *scope* dan *lifetime* variabel diperhitungkan dalam penggantian *identifier*
- agar detektor dapat mengenali kemiripan antarkode program yang diimplementasi dengan bahasa pemrograman yang berbeda, sebelumnya dapat dilakukan proses deteksi bahasa dan konversi ke dalam satu bahasa standar yang dipilih

Meskipun demikian, patut diingat bahwa pada umumnya upaya peningkatan akurasi harus dibayar dengan meningkatnya kompleksitas waktu algoritma.

9. Kesimpulan

Kemiripan kode program dapat dideteksi dengan cara menghitung *levensthein distance* dari kedua kode tersebut. Teknik *preprocessing* dapat dilakukan sebelum dilaksanakan penghitungan untuk meningkatkan akurasi pendeteksian.

10. Referensi

1. Lloyd Allison (1999), *Dynamic Programming Algorithm (DPA) for Edit-Distance* [online], School of Computer Science & SWE, Monash University.
   Tersedia di:
   http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/
   [Diakses pada 18 Mei 2006]
2. Thomas H. Cormen, Charles E. Leiserson, dan Ronald L. Rivest (1990), *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts London, England.
3. Anany Levitin (2003), *Introduction to the Design & Analysis of Algorithms*, Addison-Wesley.
4. Rinaldi Munir (2006), *Strategi Algoritmik*, Program Studi Informatika, STEI Institut Teknologi Bandung.
5. Erkki Sutinen, et al (2003), *Plagiarism Detection in Software Projects* [online], MirrorWolf.
   Tersedia di:
   http://www.cs.joensuu.fi/edtech/mw/material/plag_article.pdf
   [Diakses pada 17 Mei 2006]