Penerapan Program Dinamis pada Utilitas diff di Aplikasi Version Control System Git

Kadek Surya Mahardika 13519165
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13519165@std.stei.itb.ac.id

***Abstract*—Aplikasi yang terpenting dalam dunia *Software Engineering* adalah aplikasi *Version Control System (VCS)*. Aplikasi ini berfungsi sebagai manajemen suatu projek aplikasi. Salah satu aplikasi VCS yang popular saat ini adalah Git, Git merupakan aplikasi yang berbasis DVCS dengan performa tinggi dan mudah digunakan, Git memiliki beberapa utilitas, salah satunya adalah utilitas *diff* yang berfungsi melihat, membandingkan, dan mengartikan perubahan yang ada pada dua file saat suatu pengembang mengimplementasikan suatu fitur baru atau memperbaiki suatu *bug* pada file tersebut. Utilitas *diff* pada Git ini diimplementasikan dengan algoritma *Longest Common Subsequence (LCS)* yakni pencarian *Subsequence* terpanjang yang terdapat pada dua *string*. Pada umumnya permasalahan LCS diselesesaikan dengan pendekatan program dinamis agar mendapatkan performa yang maksimal. Sehingga, pada makalah ini penulis mencoba mereplikasi utilitas *diff* pada aplikasi VCS Git dan menganalisis seberapa siginifikan performa yang diberikan jika implementasinya berdasarkan program dinamis dibandingkan dengan pendekatan naïve.**

Keywords—VCS; Git; LCS; Kompleksitas Waktu Algoritma; Program Dinamis

I. INTRODUCTION

Pada dunia *Software Engineering*, untuk bisa membuat aplikasi yang *robust* dan *scalable*, pengembang memerlukan suatu system manajemen projek. System manajemen projek ini dikenal dengan nama *Version Control System (VCS)*. Pada era sekarang, tidak ada perusahaan perangkat lunak atau *Open Source Software (OSS)* yang tidak menggunakan *VCS* dalam proses pengembangannya karena fitur dan dampaknya ke pengembangan perangkat lunak sangatlah membantu developer untuk memanajemen *codebase* mereka.

Sekarang, salah satu VCS yang popular yakni dipakai oleh banyak pengembang perangkat lunak adalah *Git. Git* adalah DVCS (Desentralise Version Control System) yang pada awalnya dikembangkan pada tahun 2005 untuk memanajemen codebase Linux Kernel dan popularitasnya terus meningkat karena penggunaaanya yang mudah, *open source*, dan performanya yang tinggi.

Git memiliki banyak perintah atau utilitas, salah satunya adalah utilitas *diff*. Utilitas ini adalah salah satu perintah pada aplikasi git yang membantu pengembang untuk melihat, membandingkan, dan mengartikan perubahan yang ada pada *codebase* mereka, saat mereka melakukan suatu perubahan, misalnya mengimplementasi suatu fungsionalitas baru atau memperbaiki suatu *bug*.

Utilitas *diff* pada Git sangatlah membantu pengembang karena performanya yang sangat cepat untuk mencari perubahan. Contohnya pada *source code* yang memiliki kode sepanjang 100 ribu baris dan 10 ribu modifikasi, fungsionalitasi *diff* dapat mencari perubahan itu dalam waktu kurang lebih 1 detik. Performa tersebut bukanlah hal yang mustahil karena algoritma pada utilitas *diff* menggunakan *Dynamic Programming* sebagai optimasinya.

Secara internal utilitas *diff* menggunakan algoritma *Longest Common Subsequence* (LCS) untuk mencari perubahan antara dua files yang sama, pencarian tersebut dilakukan dengan pembandingan isi dari file tersebut yang dilakukan baris per baris.

Longest Common Subsequence (LCS) adalah suatu algoritma yang mencari *Subsequence* yang sama pada himpunan suatu *sequence* yang biasanya berjumlah dua. Secara naif, algoritma LCS dilakukan dengan rekursi dan akan menghasilkan kompleksitas algoritma yang exponensial. Pada penggunaanya untuk keperluan aplikasi, LCS umumnya diimplementasikan dengan prinsip program dinamis untuk memangkas kompleksitas algoritma yang exponensial tersebut menjadi polinomial kuadratik.

Paper ini adalah *technical report* dari penggunaan LCS pada utilitas *diff* pada aplikasi Git. Penulis mencoba menganalisis dan mereplikasi seberapa siginifikan implementasi LCS jika dilakukan secara naif dan secara program dinamis pada implementasi replika utilitas *diff* . Pembandingan tersebut dilakukan dengan beberapa experimen yang jumlah baris dan jumlah yang termodifikasi meningkat.

II. LANDASAN TEORI

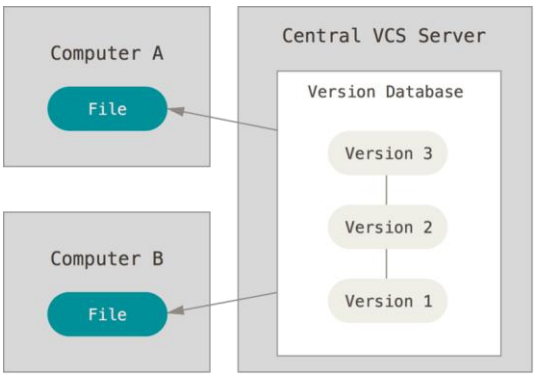
A. *Version Control System*

Version Control System (VCS) adalah suatu kelas sistem yang bertanggung jawab untuk memanajemen perubahan yang terjadi pada suatu *codebase*. Dengan kata lain, VCS adalah sistem yang merekam perubahan apapun yang telah dilakukan oleh pengembang pada *codebase* proyeknya.

Pada proses pengembangan perangkat lunak, perubahan pada beberapa bagian kode atau file yang lain adalah hal yang

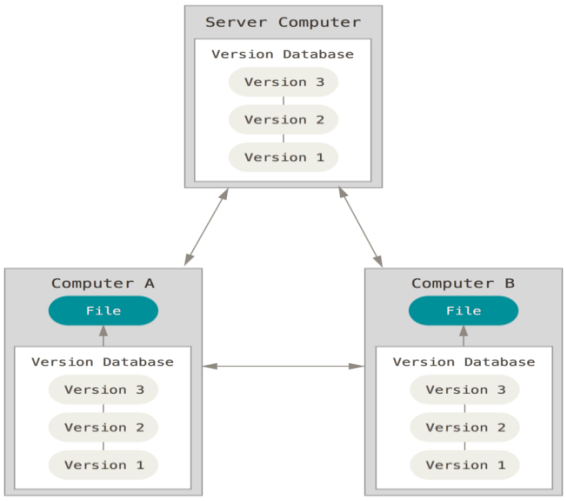
normal dimana pada proses itu pengembang menambah fitur baru ataupun memperbaiki fitur yang sudah ada. Semakin besar aplikasi yang dikembangkan maka akan semakin susah untuk memanajemen kode-kode dan file yang terkait, sehingga dengan adanya VCS akan memudahkan pengembang untuk mempercepat proses pengembangan perangkat lunak. Tanpa VCS pengembang harus membuat file yang sama setiap kali ingin mengubah file tersebut, pada proses tersebut *human error* sangatlah besar kemungkinan terjadinya.

Menurut Otte, ada dua pendekatan pada sistem VCS yakni *Centralized Version Control* (CVCS) dan *Distributed Version Control System*. CVCS adalah VCS dengan pendekatan dengan satu server sebagai repositori utama, segala perubahan disimpan di dalam server tersebut sehingga saat pengembang ingin *commit* suatu perubahan, ia harus mengkontak main server terlebih dahulu. Pendekatan ini memiliki banyak kelemahan salah satunya adalah apabila server utama down, pengembang akan memiliki banyak masalah.



Sumber : (git-scm.com)

Pendekatan yang lain yaitu DVCS, berbeda pada CVCS dimana hanya ada satu repositori yang menyimpan *history* semua perubahan kode, pada DVCS semua *clients* memiliki repositori tersendiri yang merupakan *clone* dari repository utama. Sehingga, apabila server utama mati, pengembang masih bisa menyimpan perubahannya ke repositori lokalnya atau ke server yang lain. [1]

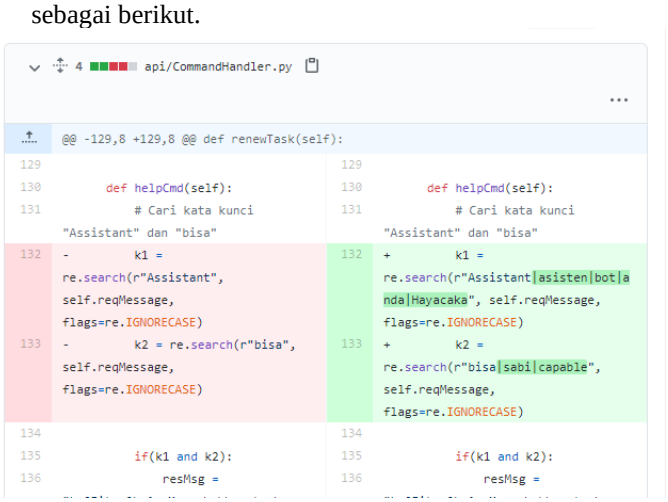


Sumber : (git-scm.com)

B. *Git dan Utilitas diff Git*

Git merupakan suatu aplikasi berbasis VCS (DVCS) yang digunakan untuk manajemen dan kolaborasi pengembang perangkat lunak. Git awalnya dikembangkan pada tahun 2005 untuk memanajemen codebase Linux Kernel dan popularitasnya terus meningkat karena penggunaannya yang mudah, *open source* dan performanya yang tinggi. [2]

Git memiliki beberapa perintah atau utilitas, salah satunya adalah utilitas *diff*. Utilitas ini adalah utilitas untuk melihat, membandingkan, dan mengartikan perubahan yang ada pada dua file yang telah diedit oleh pengembang ketika mereka mengembangkan fitur baru ataupun memperbaiki suatu *bug*. Berikut aplikasinya pada Github, yakni setiap pengembang meng*commit* file yang diubah maka utilitas *diff* akan otomatis dieksekusi dan dapat dilihat perubahannya pada history *commit* yang terlihat sebagai berikut.



Pada umumnya, utilitas *diff* didasarkan pada penyelesaian permasalahan *Longest Common Subsequence* dengan program dinamis. Namun, pada praktiknya akan dilakukan banyak optimasi sehingga performa mendapatkan performa yang baik.

C. *Kompleksitas Waktu Algoritma*

Kompleksitas waktu adalah suatu ukuran yang mendeskripsikan waktu komputer untuk mengeksekusi suatu algoritma. Kompleksitas waktu algoritma umumnya dihitung dengan jumlah operasi elementer yang dilakukan oleh suatu algoritma. Namun, karena suatu algoritma dapat memiliki waktu yang berbeda-beda untuk jumlah masukan yang sama maka yang biasanya ditinjau adalah kasus terburuk kompleksitas waktu algoritma tersebut yang diekspresikan dengan suatu fungsi yang menerima jumlah masukan algoritma.

Pada umumnya untuk mencari fungsi yang secara *exact* menghitung kebutuhan waktu suatu algoritma sangatlah susah, sehingga biasanya yang menjadi fokus adalah fungsi asimtot yang menyatakan perilaku algoritma saat input

bertambah dengan pesat yang diekspresikan dengan notasi big O, contohnya O(1), O(n), O(nlogn), O(2^n), dsb. O(1), O(n), O(n^2), dsb merupakan kompleksitas dengan waktu polinomal, kompleksitas tersebut dianggap kompleksitas yang baik untuk menyelesaikan suatu permasalahan. [3]

D. String

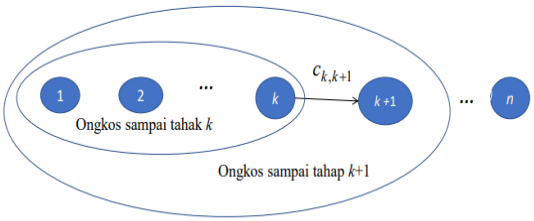
String adalah barisan dari karakter-karakter yang dapat yang dapat konstan ataupun sebagai variabel yang dapat berubah-ubah. String memiliki beberapa properti, salah satunya adalah prefix dan suffix.

Suatu string *s* adalah prefix dari string t, jika t = sx, untuk sembarang string x yang merupakan bagian dari t dan s yang mungkin saja string kosong. Suatu string *s* adalah suffix dari string t, jika t = xs dengan s yang mungkin saja kosong. [4]

E. Program Dinamis

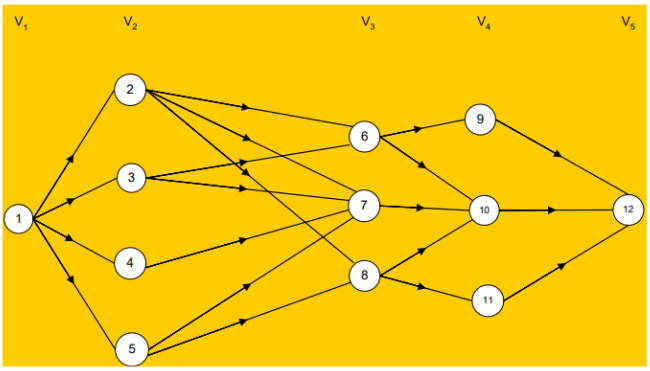
Program dinamis adalah suatu metode penyelesaian suatu permasalahan dengan pendekatan penguraian solusi menjadi beberapa tahapan sehingga solusi yang didapat merupakan serangkaian keputusan yang berkaitan. Program dinamis umumnya digunakan untuk menyelesaikan persoalan-persoalan optimasi (maksimasi atau minimisasi).

Program dinamis menggunakan prinsip optimalitas untuk membentuk rangkaian keputusan yaitu jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal yang berarti bahwa jika kita bekerja dari tahap k ke tahap k+1, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali ke tahap k untuk mendapatkan hasilnya kembali.



Sumber : (Program Dinamis (Dynamic Programming) Bagian 1 (itb.ac.id))

Ada beberapa karakteristik, suatu persoalan dapat diselesaikan dengan program dinamis yaitu pertama, persoalan dapat terbagi atas beberapa tahapan yang saling *overlap* sehingga setiap tahap dapat diambil satu keputusan, kedua, masing masing tahap terdiri dari sejumlah status yang saling berhubungan yang dapat direpresentasikan dengan graf multitahap sebagai berikut.



Sumber : (Program Dinamis (Dynamic Programming) Bagian 1 (itb.ac.id))

Kemudian, pada program dinamis, terdapat dua pendekatan untuk menyelesaikan suatu permasalahan yaitu pendekatan maju *(forward* atau *up-down)* yang melakukan perhitungan mulai dari tahap 1,2, .., n-1, n dan pendekatan mundur (*backward* atau *bottom-up*) yang melakukan perhitungan mulai dari tahap n, n-1, …, 2, 1.

Terakhir, ada beberapa langkah-langkah untuk mengembangkan algoritma program dinamis yakni pertama karakteristikkan struktur solusi optimal yang berisi tahap, variabel keputusan dan status, kedua, definisikan secara rekursif nilai solusi optimal yaitu menghubungkan nilai optimal suatu tahap dengan tahap sebelumnya, kemudian, hitung nilai solusi optimal secara maju atau mundur menggunkan tabel, terakhir rekonstruksi solusi optimal (opsional) [5]

F. Longest Common Subsequence

Longest Common Subsequence (LCS) adalah permasalahan pencarian sub-barisan terpanjang yang sama pada suatu himpunan barisan, umumnya hanya dua barisan yakni mencari LCS dari dua string. Sub-barisan ini tidak harus saling bersebelahan, tetapi urutannya relative terhadap kedua string haruslah sama. Contohnya, ada dua string atau barisan ABCD dan ACBAD, pada contoh ini nilai LCSnya adalah 3 yang dimiliki oleh sub-barisan ABD dan ACD.

Anggaplah dua barisan tersebut didefinisikan dengan X dan Y dimana X = ($x_1x_2…x_m$) dan Y = ($y_1y_2…y_n$) dan prefix dari X dan Y didefinisikan $X_{1,2,..,m}$ dan $Y_{1,2,..m}$ dalam hal ini misalnya X_3 akan menyatakan string ($x_1x_2x_3$). LCS dapat diselesaikan dengan fungsi rekursif berikut:

$LCS(X_i, Y_j) = 0$, jika i = 0 atau j = 0
$LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + 1$, jika i,j > 0 dan $x_i = y_j$
$LCS(X_i, Y_j) = \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\}$,
jika i,j > 0 dan $x_i \neq y_j$

dan fungsi ini memiliki kompleksitas algoritma O($2^{max(n, m)}$). [6]

III. Implementasi dan Eksperimen

A. *Implementasi dan Eksperimen Naïve Algoritma LCS*

Pseudocode LCS Naïve:

```
Function LCS_Naive(X[0..m-1], Y[0..n-1], m, n)
  If m = 0 OR n = 0 Then
    Return 0
  Else
    If X[m] = Y[n] Then
      Return 1 + LCS_Naive(X, Y, m-1, n-1)
    Else
      Return Max(LCS_Naive(X, Y, m, n-1), LCS_Naive(X,Y,m-1,n))
```

Eksperimen:

Eksperimen dilakukan dengan testcase:

- Testcase 15 baris kode yang sudah dimodifikasi

Testing pada algoritma ini hanyalah terbatas pada 15 baris kode, lebih dari itu, algoritma akan memakan waktu lebih dari 1 detik. Berikut hasil eksekusi algoritma tersebut.

```
(venv) C:\Users\kadek\Documents\Paper-Stima\src>python lcs.py
Testcase 5 baris
0 baris mismatch
Waktu eksekusi algoritma: 0.0 miliseconds

Testcase 10 baris
2 baris mismatch
Waktu eksekusi algoritma: 4.999755859375 miliseconds

Testcase 15 baris
4 baris mismatch
Waktu eksekusi algoritma: 1402.72607421875 miliseconds

Testcase 20 baris
```

Analisis:

Terlihat bahwa saat testcase sudah mencapai lebih dari batasan tertentu, algoritma tidak lagi efisien, itu terjadi saat testcase sudah lebih dari 15 baris. Jika diasumsikan setiap baris, memiliki panjang rata-rata 40 karakter, maka saat testcase ke 15, algoritma sudah melakukan operasi sekitar 15.2^{40} operasi. Wajar saja dengan jumlah operasi sebesar itu, algoritma sudah melebihi 1 detik karena pada umumnya 1 detik adalah ketika CPU melakukan operasi sekitar $10^8 \sim 2^{27}$. Terlihat bahwa pendekatan ini sangatlah tidak efisien untuk diterapkan di dunia nyata, khususnya aplikasi utilitas *diff* yang memerlukan jumlah pembandingan yang banyak.

B. *Implementasi dan Eksperimen Program Dinamis LCS*

Pseudocode Program Dinamis LCS:

```
Function LCS_DP(X[0..m-1], Y[0..n-1], m, n)
  { Asumsi lcsTable terinisialisasi dengan 0 }
  lcsTable = array of array m x n

  For i = 0 to m
    For j = 0 to n
      If (i = 0 Or j = 0) then
        lcsTable[i][j] <- 0
      Else
        If (X[i-1] = Y[i-1]) then
          lcsTable[i][j] <- 1 + lcsTable[i-1][j-1]
        Else
          lcsTable[i][j] =
                Max(lcsTable[i-1][j], lcsTable[i][j-1])
  ➔ lcsTable[m][n]
```

Program dinamis LCS dibangun dengan pendekatan *bottom up* yakni pendekatan dengan menyelesaikan subpermasalahan yang lebih kecil ke subpermasalahan yang lebih besar. Hal tersebut direalisasikan dengan matriks *lcsTable* berukuran mxn yang mana lcsTable[i][j] menyatakan *longest common subsequence* untuk prefix *string* $X_{1..i}$ dan prefix *string* $Y_{1..j}$, contohnya lcsTable[0][n] akan bernilai 0 karena jika salah satu string kosong maka jelas tidak ada karakter yang match.

Eksperimen:

Eksperimen dilakukan dengan testcase:

- 10 baris testcase

```
(venv) C:\Users\kadek\Documents\Paper-Stima\src>python lcs.py
Testcase 5 baris
0 baris mismatch
Waktu eksekusi algoritma: 0.0 miliseconds

Testcase 10 baris
2 baris mismatch
Waktu eksekusi algoritma: 0.0 miliseconds
```

- 100 baris testcase

```
(venv) C:\Users\kadek\Documents\Paper-Stima\src>python lcs.py
Testcase 20 baris
5 baris mismatch
Waktu eksekusi algoritma: 0.999267578125 miliseconds

Testcase 40 baris
6 baris mismatch
Waktu eksekusi algoritma: 1.99755859375 miliseconds

Testcase 60 baris
7 baris mismatch
Waktu eksekusi algoritma: 7.02197265625 miliseconds

Testcase 80 baris
9 baris mismatch
Waktu eksekusi algoritma: 6.97216796875 miliseconds

Testcase 100 baris
11 baris mismatch
Waktu eksekusi algoritma: 13.000244140625 miliseconds
```

- 1000 baris testcase

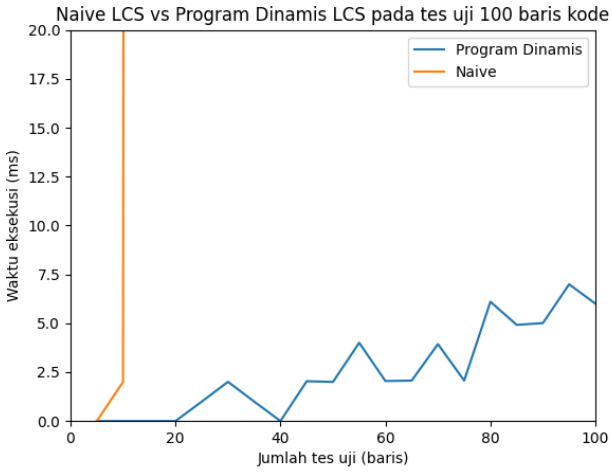
```
(venv) C:\Users\kadek\Documents\Paper-Stima\src>python lcs.py
Testcase 200 baris
0 baris mismatch
Waktu eksekusi algoritma: 21.99951171875 miliseconds

Testcase 400 baris
0 baris mismatch
Waktu eksekusi algoritma: 98.99658203125 miliseconds

Testcase 600 baris
0 baris mismatch
Waktu eksekusi algoritma: 222.004638671875 miliseconds

Testcase 800 baris
0 baris mismatch
Waktu eksekusi algoritma: 411.998291015625 miliseconds

Testcase 1000 baris
4 baris mismatch
Waktu eksekusi algoritma: 556.988037109375 miliseconds
```

Analisis:

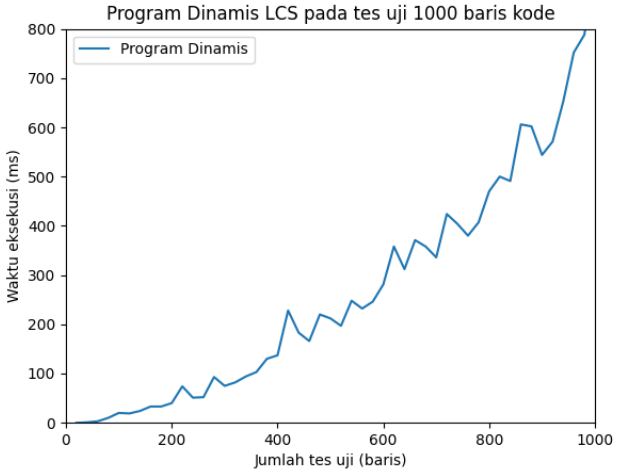
Terlihat bahwa walapun testcase sudah mencapai 1000 baris, algoritma masih berjalan dibawah 1 detik. Jika dalam 1 baris diasumsikan terdapat 40 karakter dan karena algoritma LCS dengan program dinamis berjalan dengan kompleksitas waktu algoritma $O(n^2)$ sehingga akan dilakukan operasi sekitar $1000.40^2 = 16.10^7$ operasi, jumlah operasi tersebut masih kurang dari 10^8, sesuai dengan pendekatan 1 detik ~ 10^8 operasi.

C. *Perbandingan Algoritma LCS Naïve vs LCS dengan Program Dinamis*

Berikut merupakan grafik perbandingan algorima LCS Naïve vs LCS dengan program dinamis untuk tes uji 100 baris kode.



Terlihat pada grafik tes uji bahwa LCS secara naif waktu eksekusinya naik signifikan pada saat jumlah tes uji lebih besar dari 15, sesuai dengan teori kompleksitas waktu untuk LCS naïve. Sedangkan dengan program dinamis, waktu eksekusi terlihat cenderung linear. Namun, apabila kita plot program dinamis LCS dengan tes uji 1000 baris akan didapatkan:



Terlihat bahwa dengan pedekatan program dinamis, semakin besar masukan maka grafik akan cenderung membentuk kurva fungsi kuadratik yang sesuai dengan teori kompleksitas waktu untuk program dinamis LCS yaitu $O(n^2)$.

D. *Implementasi Program Dinamis LCS sebagai Replika Utilitas diff Pada Git*

Untuk bisa mereplikasi utilitas *diff* pada git, selain fungsi panjang LCS yang telah diimplementasikan dengan program dinamis, diperlukan juga prosedur untuk mendapatkan *string* LCS terpanjang tersebut. Selain itu diperlukan juga prosedur untuk penentuan *diff* dengan memanfaatkan tabel program dinamis LCS tersebut.

1) Pseudocode LCS_String

```
function LCS_String(lcsTable, X[0..m-1], Y[0..n-1], m, n)
  stringAns <- ‘’
  while(i != 0 AND j != 0) do
    if(x[i-1] = y[j-1]) then
      i <- i-1
      j <- j-1
      stringAns <- stringAns + x[i]
    else
      if(lcsTable[m-1][n] > lcsTable[m][n-1]) then
        i <- i-1
      else
        j <- j-1
  ➔ Reverse(stringAns)
```

Penjelasan:

Untuk mendapatkan *string* LCS yang terpanjang dari tabel yang berisi panjang LCS untuk setiap prefix X dan Y, dilakukan pemetaan ulang *path* dari index 0,0 index ke m-1,n-1 dari tabel. Pemetaan ini dilakukan dengan mencocokan karakter saat ini dengan nilai panjang LCS prefix saat ini, jika misalkan karakter yang dituju pada index i,j dikedua string sama berarti karakter tersebut masuk ke string solusi, jika tidak, maka kita bandingkan prefix yang memiliki nilai LCS terbesar yaitu antara prefix $X_{(m-1)..n}$ atau $Y_{(m)..(n-1)}$ dan ubah index sekarang ke index prefix tersebut, jika keduanya memiliki panjang yang sama, tidak masalah apapun pilihannya, tapi dalam implementasi ini akan menuju ke j-1.

Kemudian, karena pemetaan ulang dilakukan mulai dari index m-1,n-1 maka string hasil perlu dibalikkan urutannya.

Berikut Ilustrasinya:
Misalkan X = INFORMATIKA dan Y = STIMA, akan didapatkan tabel LCS dan *path* sebagai berikut:

IMA

i : 11 Xi : A
j : 5 Yj : A

Step: Xi='A' equal to Yi='A'
b[11, 5]='↖' and c[11, 5]=2+1=3
See line number 10 and 11

i	j	0	1	2	3	4	5
			y	S	T	I	M	A
0	x	0	0	0	0	0	0
1	I	0	↑ 0	↑ 0	↖ 1	← 1	← 1
2	N	0	↑ 0	↑ 0	↑ 1	↑ 1	↑ 1
3	F	0	↑ 0	↑ 0	↑ 1	↑ 1	↑ 1
4	O	0	↑ 0	↑ 0	↑ 1	↑ 1	↑ 1
5	R	0	↑ 0	↑ 0	↑ 1	↑ 1	↑ 1
6	M	0	↑ 0	↑ 0	↑ 1	↖ 2	← 2
7	A	0	↑ 0	↑ 0	↑ 1	↑ 2	↖ 3
8	T	0	↑ 0	↖ 1	↑ 1	↑ 2	↑ 3
9	I	0	↑ 0	↑ 1	↖ 2	↑ 2	↑ 3
10	K	0	↑ 0	↑ 1	↑ 2	↑ 2	↑ 3
11	A	0	↑ 0	↑ 1	↑ 2	↑ 2	↖ 3

(sumber : Longest Common Subsequence Simulation (LCS) @jon.andika (sourceforge.net))

2) Pseudocode Prosedur Penentuan Diff

```
procedure print_diff(lcsTable, X[0..m-1], Y[0..n-1], i, j)
   listDiff <- []

   while(i >= 0 OR j >=0) do
      if(i < 0) then
         listDiff <- listDiff + ['+ ' + y[j]]
         j <- j-1
      else if(j < 0) then
         listDiff <- listDiff + ['- ' + x[i]]
         i <- i-1
      else if(x[i] = y[j]) then
         listDiff <- listDiff + [' ' + x[i]]
         i <- i-1
         j <- j-1
      else if(lcsTable[i][j-1] >= lcsTable[i-1][j]) then
         listDiff <- listDiff + ['+ ' + y[j]]
         j <- j-1
      else
         listDiff <- listDiff + ['- ' + x[i]]
         i <- i-1

   for diff in Reverse(listDiff):
      write(diff)
```

Penjelasan:

Prinsip penentuan *diff* dengan menggunakan tabel lcsTable, sama seperti penentuan LCS_String yakni penentuan dimulai dari index belakang (m-1, n-1) dan mengikuti aturan yang sama dengan LCS_String dengan beberapa modifikasi yakni '+' saat suatu baris mendapat penambahan *string* dan '-' saat suatu baris mendapat pengurangan *string*.

3) Implementasi dan Eksperimen Fungsi dan Prosedur Terkait

Misalkan terdapat dua file tes uji `10linescase_old.txt` dan `10linecase_new.txt`, `10linecase_new.txt` merupakan file yang akan dimodifikasi beberapa barisnya, misalkan isi filenya:

10linescase_old.txt

```
1  # my_first_calculator.py by AceLewis
2  # TODO: Make it work for all floating point numbers too
3
4  if 3/2 == 1:  # Because Python 2 does not know maths
5      input = raw_input  # Python 2 compatibility
6
7  print('Welcome to this calculator!')
8  print('It can add, subtract, multiply and divide whole numbers from 0 to 50')
9  num1 = int(input('Please choose your first number: '))
10 sign = input('What do you want to do? +, -, /, or *: ')
11
```

Akan dilakukan modifikasi pada baris 2, 4, 10 dan penghapusan pada baris ke 7. Sehingga file yang baru menjadi:

10linescase_new.txt

```
1  # my_first_calculator.py by AceLewis
2  # TODO: Make it work for many floating point numbers too
3
4  if 3/2 == 1:  # Because Python 2 does know maths
5      input = raw_input  # Python 2 compatibility
6
7
8  print('It can add, subtract, multiply and divide whole numbers from 0 to 50')
9  num1 = int(input('Please choose your first number: '))
10 sign = input('What operator do you want to do? +, -, /, or *: ')
11
```

Hasil setelah implementasi dan eksekusi fungsi dan prosedur diatas:

```
(venv) C:\Users\kadek\Documents\Paper-Stima\src>python diff.py ..\10linescase_old.txt
- # my_first_calculator.py by AceLewis
- # TODO: Make it work for all floating point numbers too
+ # my_first_calculator.py by AceLewis
+ # TODO: Make it work for many floating point numbers too
-
- if 3/2 == 1:  # Because Python 2 does not know maths
+
+ if 3/2 == 1:  # Because Python 2 does know maths
      input = raw_input  # Python 2 compatibility

- print('Welcome to this calculator!')
+
  print('It can add, subtract, multiply and divide whole numbers from 0 to 50')
  num1 = int(input('Please choose your first number: '))
- sign = input('What do you want to do? +, -, /, or *: ')
+ sign = input('What operator do you want to do? +, -, /, or *: ')

(venv) C:\Users\kadek\Documents\Paper-Stima\src>
```

Terlihat bahwa fungsionalitas *diff* pada VCS Git sudah dapat tereplikasi, walaupun masih belum sempurna, namun secara internal algoritma yang terpakai adalah sama yaitu LCS dengan program dinamis.

IV. Penutup

A. Kesimpulan

1. Terdapat perbedaan performansi yang signifikan saat menggunakan algoritma LCS dengan pendekatan *naïve* dibandingkan dengan pendekatan program dinamis. Algoritma *naïve* yang secara teori meningkat secara exponensial terbukti tidak efektif ketika masukan meningkat secara pesat.

2. Pada aplikasinya di dunia nyata, khususnya pada aplikasi *Version Control System* Git, implementasi algoritma untuk utilitas *diff*nya diimplementasikan dengan LCS program dinamis karena performanya sudah terbukti sangat efisien bahkan untuk file dengan 1000 baris kode.

B. Saran

1. Pembaca diharapkan membaca *source code* aplikasi VCS Git untuk melihat perbedaan implementasi dan optimasi yang dipakai pada dunia nyata pada referensi [7]

Video Link at Youtube

https://youtu.be/MBsL0nIU_pE

Ucapan Terima Kasih

Penulis mengucapkan terima masih kepada Tuhan Yang Maha Esa karena atas berkat dan rahmatnya, penulis dapat menyelesaikan makalah yang berjudul "Penerapan Program Dinamis pada Utilitas diff di Aplikasi Version Control System Git" .

Penulis juga mengucapkan terima kasih kepada tim dosen pengajar IF2211 Strategi Algoritma yang telah memberikan ilmu-ilmu baik yang berikaitan dalam pembuatan makalah ini.

Penulisa juga berterima kasih kepada kerabat maupun teman-teman penulis yang terlah membantu baik secara langsung maupun moral dalam perkuliahan ini.

References

[1] Zolkipli, Nazatul Nurlisa & Ngah, Amir & Deraman, Aziz. (2018). Version Control System: A Review. Procedia Computer Science. 135. 408-415. 10.1016/j.procs.2018.08.191.

[2] "A Short History of Git". Pro Git (2nd ed.). Apress. 2014. Archived from the original on 25 December 2015.

[3] Sipser, Michael (2006). Introduction to the Theory of Computation. Course Technology Inc. ISBN 0-619-21764-2.

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf.

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf.

[6] http://www.columbia.edu/~cs2035/courses/csor4231.F11/lcs.pdf

[7] https://github.com/git/git

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, 11 Mei 2021

Ttd

Kadek Surya Mahardika 13519165