Perbandingan Kecepatan Algoritma *Flood Fill* yang Diimplementasi Menggunakan DFS Tumpukan Eksplisit dan BFS

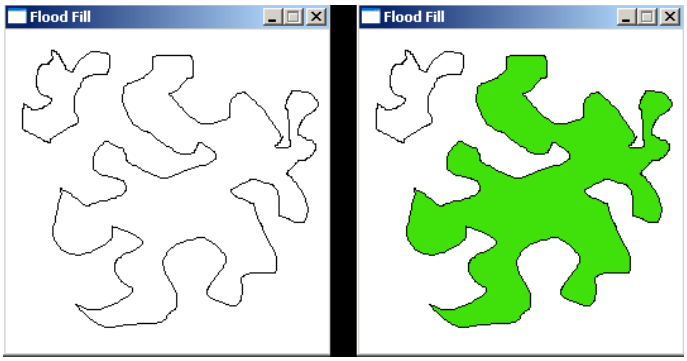
Reinaldo Ignatius Wijaya
Program Studi Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha No.10 Bandung 40132, Indonesia
13515093@std.stei.itb.ac.id
rei_iw@hotmail.com

***Abstrak*—Pada aplikasi-aplikasi pengolah gambar berbasis bitmap/raster kita dapat menemui kakas bernama *paint bucket*. Tool tersebut berupa sebuah ember cat dapat dituangkan oleh pengguna pada kanvasnya. Jika pengguna menggunakan *paint bucket* pada suatu pixel, maka warna yang diberikan pada pixel tersebut akan menyebar ke semua pixel berwarna sama yang bersebelahan dengan pixel tersebut. Untuk mencapai efek tersebut, digunakanlah algoritma *flood fill*.**

Kata Kunci—Flood Fill, DFS, BFS

I. PENDAHULUAN

Flood fill adalah sebuah algoritma yang digunakan untuk menentukan area yang terhubung pada suatu titik pada array berdimensi dua atau lebih. *Flood Fill* juga kadang disebut sebagai *seed fill* atau *boundaryfill* (jika diaplikasikan untuk mengisi suatu area terbatas pada gambar seperti yang dilakukan kakas *paint bucket*).



Gambar 1.1 *Flood Fill*

Sumber: http://lodev.org/cgtutor/floodfill.html

Walaupun aplikasi algoritma *flood fill* kebanyakan terdapat pada grafika komputer, *flood fill* juga digunakan pada beberapa permainan mulai dari permainan abad ke-20 seperti *Minesweeper* sampai permainan tradisional China yang bernama *Go*. Pada permainan *Go* pemain akan bertujuan mengelilingi bidak lawan menggunakan bidak miliknya. Bidak yang dikelilingi akan "dimakan". Bidak yang mengelilingi bidak musuh akan menjadi batas dari *flood fill*, yang kemudian akan dilakukan di dalamnya.

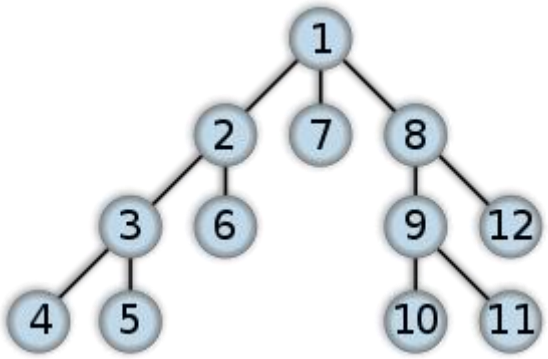
Seperti namanya, algoritma *flood fill* berperilaku seperti banjir. Algoritma ini "membanjiri" area yang terhubung dengan simpul awal yang dipilih. Untuk mengimplementasikan algoritma *flood fill* ini, terdapat dua algortima traversal graf yang dapat digunakan, yakni DFS dan BFS.

Dalam makalah ini, penulis akan membahas algoritma mana yang lebih baik untuk digunakan untuk mengimplementasikan *flood fill*.

II. DASAR TEORI

A. *Depth-First Search*

Depth-first search (DFS) atau pencarian mendalam adalah salah satu algoritma traversal atau pencarian pada struktur data pohon ataupun graf. Pencarian dilakukan dari akar (pada struktur data pohon) ataupun suatu simpul (pada struktur data graf). Pencarian dilakukan sedalam mungkin (pada kasus pohon, sampai *level* tertinggi yang bisa dicapai). Jika solusi tidak ditemukan pada ujung pencarian maka akan dilakukan runut balik secukupnya dan penulusuran akan dilakukan ke cabang lain.



Gambar 2.1 Urutan pencarian DFS

Sumber:
https://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Depth-first-tree.svg/300px-Depth-first-tree.svg.png

Berikut adalah langkah-langkah DFS.

Catatan: pencarian dimulai dari simpul v.

1. Kunjungi simpul v.
2. Kunjungi simpul w yang bertetangga dnegan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelunmnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.[1]

Pada implementasinya DFS memerlukan struktur data stack atau tumpukan yang digunakan untuk menyimpan simpul-simpul yang akan dikunjungi pada jalur yang sedang ditelusuri. Pada awalnya tumpukan tersebut kosong. Pada saat simpul baru dikunjungi maka simpul tersebut akan ditambakahkan (push) ke tumpukan. Sedangkan pada saat dilakukan runut balik, simpul yang dirunut balik akan dikeluarkan dari tumpukan (pop).

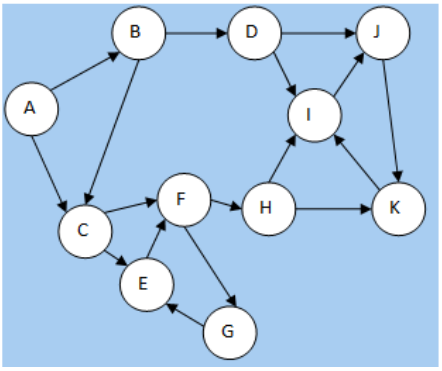
Tumpukan pada DFS dapat bersifat explisit maupun implisit. Secara eksplisit artinya terdapat implementasi struktur data tumpukan pada program. Sedangkan implisit artinya DFS dilakukan secara rekursif yang belaku seperti tumpukan. DFS yang menggunakan tumpukan implisit sering juga disebut DFS rekursif.

Waktu yang diperlukan oleh DFS untuk menelusuri graf secara menyeluruh adalah $\Theta(|V| + |E|)$. Di mana $|V|$ adalah jumlah simpul pada graf, dan $|E|$ adalah jumlah sisi pada graf. Untuk graf implisit, waktu yang diperlukan DFS untuk menelusurinya adalah $O(b^d)$, di mana b adalah *branching factor* dari graf dan d adalah kedalaman yang ditelusuri oleh DFS.

Memori yang diperlukan oleh DFS adalah $O(|V|)$ pada kasus terburuknya pada penulurusan tanpa perulangan (setiap simpul ditelusuri tidak lebih dari satu kali) untuk menyimpan tumpukan dari simpul-simpul yang dikunjungi pada jalur yang sedang ditelusuri.

Contoh kasus:

Diberikan graf sebagai berikut.



Gambar 2.2 Graf Berarah

Sumber: https://i.stack.imgur.com/hxRPl.png

Dengan menggunakan DFS, sebuah jalur akan ditentukan, dari simpul A untuk mencapai simpul K dengan asumsi bahwa jika terdapat lebih dari satu simpul tetangga maka simpul yang akan dipilih adalah simpul dengan abjad terkecil.

Berikut adalah ilustrasi pencarian menggunakan DFS untuk contoh kasus di atas.

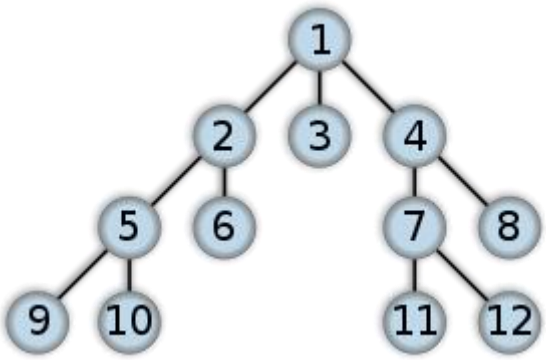
1. DFS(A): visited[A] = true; DFS(B)
2. DFS(B): visited[B] = true; DFS(C)
3. DFS(C): visited[C] = true; DFS(E)
4. DFS(E): visited[E] = true; DFS(F)
5. DFS(F): visited[F] = true; DFS(G)
6. DFS(G): visited[G] = true; Backtrack; DFS(H)
7. DFS(H): visited[H] = true; DFS(I)
8. DFS(I): visited[I] = true; DFS(J)
9. DFS(J): visited[J] = true; DFS(K)
10. DFS(K): visited[K] = true

Dari ilustrasi di atas didapat solusi: A, B, C, E, F, H, I, J, K.

Jika DFS di atas dilakukan tanpa mengingat simpul yang telah dikunjungi maka pencarian akan terjebak pada siklus E, F, G, E dan tidak dapat mencapai simpul K. Oleh karena itu DFS dapat dikatakan bersifat tidak *complete* karena pencarian dapat "terserat" pada graf yang tidak memiliki *goal state* dan tidak pernah keluar.

B. Breadth-First Search

Breadth-first search (BFS) atau pencarian melebar seperti hanya DFS adalah salah satu algoritma traversal atau pencarian pada struktur data pohon atau graf. Pencarian dimulai dari simpul akar (struktur data pohon) atau simpul awal (struktur data graf). Perbedaannya dengan DFS, BFS menelusuri semua tetangga dari simpul terlebih dahulu sebelum melanjutkan pencarian ke kedalaman selanjutnya.



Gambar 2.3 Urutan pencarian BFS

Sumber: https://upload.wikimedia.org/wikipedia/commons/thumb/3/33/Breadth-first-tree.svg/300px-Breadth-first-tree.svg.png

Berikut adalah langkah-langkah BFS.

Catatan: pencarian dimulai dari simpul v.

1. Kunjungi simpul v.
2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang telah dikunjungi, demikian seterusnya.[1]

Pada implementasinya BFS memerlukan struktur data queue atau antrian. Antrian tersebut berisi simpul-simpul mana saja yang harus dikunjungi selanjutnya. Pada awalnya antrian tersebut berisi simpul akar atau awal. Setelah mengunjungi suastu simpul, semua tetangga dari simpul tersebut yang belum pernah dikunjungi akan dimasukkan ke antrian (enqueue). Kemudian simpul tersebut akan dikeluarkan dari antrian (dequeue). Jika antrian kosong, maka artinya graf sudah ditelusuri sepenuhnya.

Waktu yang diperlukan oleh BFS pada kasus terburuk adalah O(|V|+|E|). Di mana |V| adalah jumlah simpul pada graf dan |E| adalah jumlah sisi pada graf. Sama seperti DFS.

Memori yang diperlukan BFS adalah $\Theta(|V| + |E|)$ jika graf direpresentasikan dengan senarai ketetanggaan (*adjacency list*) dan $\Theta(|V|^2)$ jika graf direpresentasikan dengan matriks ketetanggaan (*adjacency matrix*).

Kompleksitas waktu dan memori di biasanya berlaku untuk graf yang dapat disimpan secara eksplisit atau kecil. Sedangkan untuk graf yang ukuran yang tak hingga kompleksitas waktu maupun memori nya dalah $O(b^d)$ di mana b adalah *branching factor* dari graf dan d adalah kedalaman simpul yang dicari dari simpul awal.

Contoh kasus:

Diberikan graf seperti pada gambar 2.2. Tentukan apakah simpul A terhubung dengan simpul K. Asumsi simpul yang akan dimasukkan ke dalam antrian lebih dahulu adalah simpul dengan abjad terkecil.

Berikut adalah ilustrasi pencarian menggunakan BFS untuk contoh kasus di atas.

i	v	Queue	visited										
			A	B	C	D	E	F	G	H	I	J	K
0	A	{A}	T	F	F	F	F	F	F	F	F	F	F
1	A	{B,C}	T	T	F	F	F	F	F	F	F	F	F
2	B	{C,D}	T	F	F	F	F	F	F	F	F	F	F
3	C	{D,E,F}	T	T	T	F	F	F	F	F	F	F	F
4	D	{E,F,I,J}	T	T	T	T	F	F	F	F	F	F	F
5	E	{F,I,J}	T	T	T	T	T	F	F	F	F	F	F
6	F	{I,J,G,H}	T	T	T	T	T	T	F	F	F	F	F
7	I	{J,G,H}	T	T	T	T	T	T	F	F	T	F	F
8	J	{G,H,K}	T	T	T	T	T	T	F	F	T	T	F
9	G	{H,K}	T	T	T	T	T	T	T	F	T	T	F
10	H	{K}	T	T	T	T	T	T	T	T	T	T	F
11	K	{}	T	T	T	T	T	T	T	T	T	T	T

Tabel 2.1 Ilustrasi BFS

Sumber: Dokumen Pribadi

Dari ilustrasi di atas dapat ditarik kesimpulan bahwa K dapat dicapai A.

Berbeda dengan DFS, BFS memiliki sifat *complete*. Artinya jika BFS diaplikasikan pada graf tak hingga, suatu saat solusi akan ditemukan.

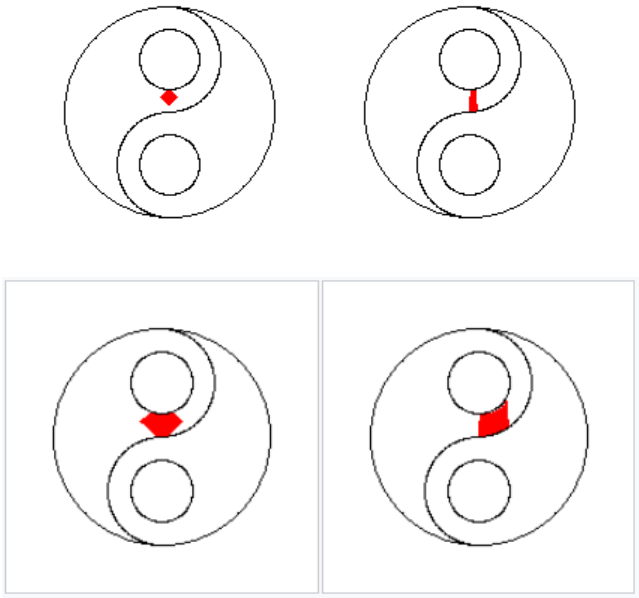
C. Flood Fill

Flood fill adalah algoritma yang digunakan untuk memeriksa keterhubungan suatu area dengan salah satu titik di larik berdimensi dua atau lebih. Implementasi dari *flood fill* dapat memanfaatkan kedua algoritma traversal graf yang telah dijelaskan di atas.

Flood fill pada larik berdimensi dua berdasarkan banyak arah penyebarannya dibagi dua, yakni *Four-way flood fill* dan *eight-way flood fill*. *Four-way flood fill* hanya memungkinkan penyebaran ke arah atas, bawah, kiri, dan kanan. Sedangkan

eight-way flood fill memungkinkan penyebaran ke empat diagonalnya.

Flood fill yang diimplementasikan dengan DFS, jika disimulasikan akan terlihat seperti manusia yang sedang mewarnai suatu gambar dengan pensil warna. Sedangkan jika diimplementasikan dengan BFS, akan terlihat seperti cat dituangkan ke dalam suatu wadah. Walaupun demikian, hasil yang didapat tidak memliki perbedaan.



Gambar 2.4 Ilustrasi *flood fill* dimensi dua dengan BFS (kiri) dan DFS (kanan)

Sumber: https://en.wikipedia.org/wiki/File:Wfm_floodfill_animation_queque.gif dan https://en.wikipedia.org/wiki/File:Wfm_floodfill_animation_stack.gif

III. Perbandingan Kecepatan Algoritma Flood Fill yang Diimplementasi Menggunakan DFS Tumpukan Eksplisit dan BFS

A. *Kasus uji*

Untuk menguji perbedaan waktu yang dibutuhkan algoritma *flood fill* yang diimplementasi dengan DFS dan BFS, akan disediakan kasus uji dengan input berupa larik dua dimensi yang berisi karakter "." dan '+'. Tanda '+' akan menjadi batas dan tanda '.' akan menjadi bagian yang "dibanjiri" oleh *flood fill*.Ukuran input akan divariasikan mulai dari 10*10, 100*100, 1000*1000, 10000*10000, dan 100000*100000. Algoritma *flood fill* yang digunakan adalah jenis *four-way*.

Akan terdapat tiga jenis kasus uji. Yang pertama adalah semua elemen larik merupakan tanda '.'.

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

Code 3.2 Contoh kasus uji tipe 2 dengan n = 10

Sumber: Dokumen pribadi

Kasus uji kedua akan memiliki beberapa tanda '+' dengan pola sebagai berikut.

```
. . . . . . . . . .
. + . . . . . . + .
. . + . . . . + . .
. . . + . . + . . .
. . . + . . + . . .
. . . + . . + . . .
. . . + . . + . . .
. . + . . . . + . .
. + . . . . . . + .
. . . . . . . . . .
```

Code 3.2 Contoh kasus uji tipe 2 dengan n = 10

Sumber: Dokumen pribadi

Kasus uji ketiga menyerupai kasus uji kedua namun dirotasi 90 derajat.

```
. . . . . . . . . .
. + . . . . . . + .
. . + . . . . + . .
. . . + + + + . . .
. . . . . . . . . .
. . . . . . . . . .
. . . + + + + . . .
. . + . . . . + . .
. + . . . . . . + .
. . . . . . . . . .
```

Code 3.3 Contoh kasus uji tipe 3 dengan n = 10

Sumber: Dokumen pribadi

Untuk ketiga kasus uji di atas *flood fill* akan dimulai pada titik n/2-1, n/2-1 dengan n adalah ukuran larik pada salah satu dimensinya. *Flood fill* akan selesai jika semua elemen larik yang berisi tanda '.' sudah dikunjungi tepat sekali.

B. *Flood fill dengan DFS tumpukan eksplisit*

Pada implementasinya *flood fill* DFS akan menggunakan tumpukan eksplisit. Implementasi DFS rekursif tidak memungkinkan karena ukuran data yang relatif besar sehingga tidak dapat disimpan pada tumpukan *runtime*. Jika terdapat

lebih dari satu arah yang dapat ditelusuri, maka prioritas arah yang akan dipilih adalah atas, kiri, kanan, kemudian bawah.

Berikut adalah *pseudocode* dari *flood fill* dengan DFS rekursif.

```
procedure floodFillDFS(input i :
integer, input j : integer)
{Melakukan flood fill dengan DFS pada
titik pada baris i dan kolom j}
Deklarasi:
   s : stack of pair of integer
   temp : pair of integer
Algoritma:
   s.push(i, j)
   kunjungi titik i, j
   while not s.empty() do
      temp = s.pop()
      for semua tetangga temp do
          if tetangga belum dikunjungi
          then
              s.push(tetangga)
              kunjungi tetangga
```

Code 3.4 *Pseudocode flood fill* DFS tumpukan eksplisit

Sumber: Dokumen pribadi

C. Flood fill dengan BFS

Pada *flood fill* dengan BFS, struktur data antrian akan digunakan secara eksplisit. Jika terdapat lebih dari satu arah yang dapat ditelusuri, maka prioritas arah yang akan dipilih adalah atas, kiri, kanan, kemudian bawah.

Berikut adalah *pseudocode* dari *flood fill* dengan BFS.

```
procedure floodFIllBFS(input i :
integer, input j : integer)
{Melakukan flood fill dengan BFS pada
titik pada baris i dan kolom j}
Deklarasi:
   q : queue of pair of integer
   temp : pair of integer
Algoritma:
   q.enqueue(i, j)
   kunjungi titik i, j
   while not q.empty() do
      temp = q.dequeue()
      for semua tetangga temp do
          if tetangga belum dikunjungi
          then
              q.enqueue(tetangga)
              kunjungi tetangga
```

Code 3.5 *Pseudocode flood fill* BFS

Sumber: Dokumen Pribadi

D. Perbandingan waktu algoritma flood fill DFS dan BFS

Berikut adalah tabel yang menunjukkan hasil percobaan dengan kasus uji pertama.

n	DFS	BFS
10	0 ns	0 ns
100	1504300 ns	1500100 ns
1000	119093900 ns	123845200 ns
10000	12317001400 ns	19475013900 ns

Tabel 4.1 Hasil uji coba kasus 1

Sumber: Dokumen pribadi

Berikut adalah tabel yang menunjukkan hasil percobaan dengan kasus uji kedua

n	DFS	BFS
10	0 ns	0 ns
100	1002900 ns	998400 ns
1000	123586900 ns	130590500 ns
10000	13960421700 ns	21724157200 ns

Tabel 4.2 Hasil uji coba kasus 2

Sumber: Dokumen pribadi

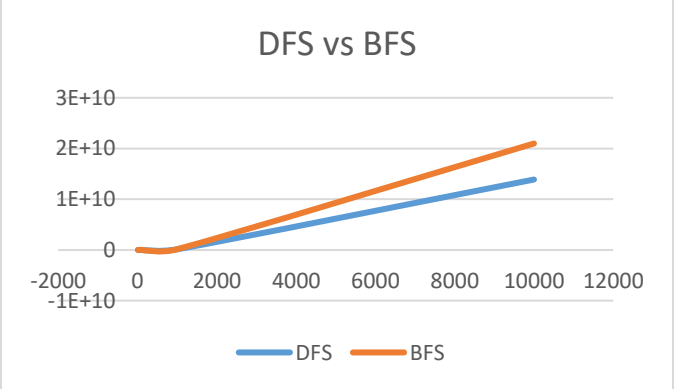
Berikut adalah tabel yang menunjukkan hasil percobaan dengan kasus uji ketiga

n	DFS	BFS
10	0 ns	0 ns
100	1502600 ns	999200 ns
1000	128605400 ns	149604500 ns
10000	15370962000 ns	21745495400 ns

Tabel 4.3 Hasil uji coba kasus 3

Sumber: Dokumen pribadi

Data dari ketiga tabel tersebut kemudian dirata-ratakan untuk menghasilkan grafik berikut.



Grafik 4.1 DFS vs BFS. Sumbu x menunjukkan n dan y menunjukkan waktu yang diperlukan dalam satuan nano detik

Sumber: Dokumen Pribadi

IV. KESIMPULAN

Untuk percobaan dengan data sangat kecil (dalam kasus ini n <= 10), tidak dapat dijadikan pembanding karena waktu yang diperlukan dari kedua algoritma sangat kecil (sepersekian nano detik). Dari bagian D pada bab III kita dapat menyimpulkan bahwa *flood fill* lebih baik diimplementasikan dengan BFS untuk data berukuran kecil (dalam kasus ini n <= 100), sedangkan DFS untuk data yang berukuran besar. Untuk penyebab mengapa *flood fill* dengan BFS lebih cepat pada data dengan ukuran kecil masih diperlukan penelitian lebih lanjut.

UCAPAN TERIMAKASIH

Penulis mengucapkan terimakasih kepada Tuhan Yang Maha Esa karena oleh kuasanya penulis dapat menyelesaikan makalah ini. Selanjutnya penulis mengucapkan terimakasih kepada kedua orangtua penulis yang oleh restu mereka penulis dapat menyelesaikan makalah ini. Penulis juga mengucapkan terimakasih kepada Bapak Rinaldi Munir, Ibu Masayu Leylia Khodra, dan Ibu Nur Ulfa Mauliadevi selaku dosen pengampu mata kuliah Strategi Algoritma karena makalah ini ditulis dengan bekal materi yang telah mereka berikan kepada penulis. Terakhir, penulis ingin berterimakasih kepada pihak-pihak lain yang secara langsung maupun tidak langsung telah membantu penulisan makalah ini

REFERENSI

[1] Diktat Strategi Algoritma Rinaldi Munir

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Mei 2017

Reinaldo Ignatius Wijaya – 13515093