Laporan Tugas Besar III

IF2211 Strategi Algoritma

**Aplikasi String Matching untuk Disposisi Tweets ke Dinas-Dinas dan Instansi di Bawah Pemerintah Kota Bandung**

*Micky Yudi Utama 13514011*

*Atika Azzahra Akbar 13514077*

*Ade Yusuf Rahardian 13514079*

Program Studi Teknik Informatika

Institut Teknologi Bandung

Semester Genap Tahun Ajaran 2015/2016

Daftar Isi

[Cover](#_Toc446878936) 1

[Daftar Isi](#_Toc446878936) 2

Bab I [Deskripsi Tugas 3](#_Toc446878914)

Bab II [Dasar Teori 5](#_Toc446878916)

[A. Algoritma DFS 5](#_Toc446878917)

[B. Algoritma BFS 7](#_Toc446878918)

Bab III [Analisis Pemecahan Masalah 11](#_Toc446878920)

[A. Spesifikasi Program 11](#_Toc446878921)

[B. Struktur Data 11](#_Toc446878922)

[C. Pemecahan Masalah 13](#_Toc446878923)

Bab IV [Implementasi dan Pengujian 14](#_Toc446878925)

1. [Implementasi Struktur Data 14](#_Toc446878926)

[B. Modul Program 14](#_Toc446878927)

[C. Fungsi dan Prosedur 15](#_Toc446878928)

[D. Screenshot Antarmuka Program 16](#_Toc446878929)

[E. Pengujian 17](#_Toc446878930)

[F. Analisis 17](#_Toc446878931)

Bab V [Kesimpulan dan Saran 19](#_Toc446878933)

[A. Kesimpulan 19](#_Toc446878934)

[B. Saran 19](#_Toc446878935)

[Daftar Pustaka 20](#_Toc446878936)

# Bab I

# Deskripsi Tugas

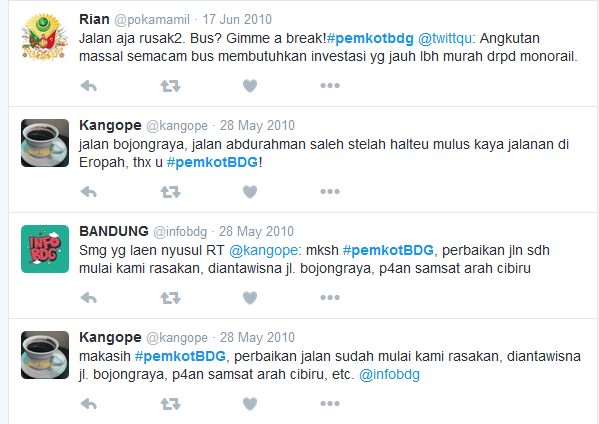
Pada Tugas Besar III kali ini Kami diminta membuat aplikasi sederhana analisis *tweet* berbasis kata kunci. *Tweet* yang berhasil diunduh dari *Twitter* dianalisis untuk selanjutnya didisposisikan ke dinas atau instansi yang terkait. Salah satu fungsi dasar yang digunakan dalam sistem analisis teks tersebut adalah pencocokan string. Algoritma pencocokan *string* (*pattern*) yang digunakan adalah *Knuth-Morris-Pratt* (KMP), dan *Boyer-Moore* (BM).

Pengguna aplikasi analisis *tweet* sederhana ini akan memberikan dua jenis masukan yaitu: (1) *keyword* pencarian *tweet* yang berasal dari tagar (*hashhtag*) atau *mention* #pemkotbdg, @pemkotbdg, #pemkotbandung, @ridwan kamil, @infobdg, dan yang sejenis dengan itu, dan (2) *keyword*. Tabel berikut menunjukkan contoh masukan dari pengguna.

|  |  |  |
| --- | --- | --- |
| *Keyword* Pencarian *Tweet* | *Keyword* | Output ke Dinas/Lembaga |
| #pemkotbdg  @pemkotbdg  @ridwan kamil  @infobdg | Pipa bocor; air ga ngalir; pdam mati; ga ngocor  ... | PDAM Bandung |
| PJU; Penerangan mati; lampu padam; terang; jalan gelap;  … | Dinas Bina Marga |
| sampah; bau busuk;  … | Dinas kebersihan |



**Gambar 1. Contoh tweets @pemkotbdg**



**Gambar 2. Contoh tweets #pemkotbdg**

Berdasarkan *keyword* pencarian *tweet* yang dimasukkan pengguna, aplikasi akan menggunakan *Twitter API* untuk mendapatkan minimal 100 *tweets* terbaru (*recent tweets*). Lalu, aplikasi menentukan kategori dinas/instansi dari setiap *tweet* dengan mengecek apakah *tweet* tersebut mengandung *keyword* yang diberikan untuk kategori dinas/lembaga tersebut.

* Jika *tweet* mengandung lebih dari satu *keyword* dari kategori dinas yang berbeda, *tweet* tersebut masuk dalam kategori dengan *keyword* yang muncul lebih dulu. Sebagai contoh, jika terdapat keyword “sampah” (kategori “dinas kebersihan”) dan “taman kotor” (kategori “dinas pertamanan”), *tweet* masuk kategori dinas kebersihan jika “sampah” muncul lebih dulu daripada kata “taman”.
* Jika tidak mengandung *keyword* apapun, *tweet* tersebut masuk dalam kategori dinas “unknown”.

Pencocokan *string* yang dibuat adalah *exact matching*, jadi *tweet* yang diproses mengandung string yang tepat sama dengan *keyword* yang telah ditentukan oleh pengguna. Di sini Algoritma KMP dan Boyer-Moore dapat digunakan. Untuk *keyword* berupa frase, maka perlu dilakukan penanganan secara khusus untuk menemukan tweet yang mengandung kata-kata di dalam frasa. Pencarian juga tidak bersifat *case sensitive*, jadi huruf besar dan huruf kecil dianggap sama (hal ini dapat dilakukan dengan mengganggap seluruh karakter di dalam *pattern* dan teks sebagai huruf kecil semua atau huruf kapital semua).

Jika *tweet* mengandung nama tempat (dimulai dengan kata depan “di”), maka sebagai **bonus**, Kami dapat menampilkan lokasi dari setiap kategori dengan *GoogleMaps API*.

|  |  |
| --- | --- |
| Tweet | Lokasi Hasil Ekstraksi |
| [**#pemkotbdg**](https://twitter.com/search?q=%23lalinBDG&src=hash) 09.16 : lampu di jln tamansari jika malam padam | tamansari |
| [@**infobdg**](https://twitter.com/infobdg): 08.23 : Macet (lagi) di jln rancaekek - cicalengka maju dikit2 ga terlalu [pic.twitter.com/gyK7PaCTVM](http://t.co/gyK7PaCTVM) | [@**Latifaa\_Aulia**](https://twitter.com/Latifaa_Aulia): | rancaekek - cicalengka |
| [#**suaraBDG**](https://twitter.com/search?q=%23suaraBDG&src=hash) via [@**dionmudjenan**](https://twitter.com/dionmudjenan): Hatihati di jembatan perbatasan komp GBI td ada yg kena begal pelaku mabok bawa golok nyandra yg bawa motor | jembatan perbatasan komp GBI |
| [@**ridwan kamil**](https://twitter.com/quinsymegamira): di terminal leuwipanjang banyak anak jalanan yg ngemis2 sementara ibunya diem di pos polisi sambil bbman X\_X [#**Suarabdg**](https://twitter.com/search?q=%23Suarabdg&src=hash) [@**dinsos\_bdg**](https://twitter.com/Dinsos_BDG) | terminal leuwipanjang |

# Bab II

# Dasar Teori

## A. Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt adalah salah satu algoritma pencarian string, dikembangkan secara terpisah oleh Donald E. Knuth pada tahun 1967 dan James H. Morris bersama Vaughan R. Pratt pada tahun 1966, namun keduanya mempublikasikannya secara bersamaan pada tahun 1977.

Perhitungan penggeseran pada algoritma ini adalah sebagai berikut, bila terjadi ketidakcocokkan pada saat pattern sejajar dengan **teks[i..i+n-1]**, kita bisa menganggap ketidakcocokan pertama terjadi di antara **teks[i+j]** dan **pattern[j]**, dengan **0 < j < n**. Berarti, **teks[i..i+j-1]** = **pattern[0..j-1]** dan **a=teks[i+j]** tidak sama dengan **b=pattern[j]**. Ketika kita menggeser, sangat beralasan bila ada sebuah awalan **v** dari pattern akan sama dengan sebagian akhiran **u** dari sebagian teks. Sehingga kita bisa menggeser pattern agar awalan **v** tersebut sejajar dengan akhiran dari **u**.

Dengan kata lain, pencocokkan string akan berjalan secara efisien bila kita mempunyai tabel yang menentukan berapa panjang kita seharusnya menggeser seandainya terdeteksi ketidakcocokkan di karakter ke-j dari pattern. Tabel itu harus memuat **next[j]** yang merupakan posisi karakter **pattern[j]** setelah digeser, sehingga kita bisa menggeser pattern sebesar **j-next[j]** relatif terhadap teks.

Secara sistematis, langkah-langkah yang dilakukan algoritma Knuth-Morris-Pratt pada saat mencocokkan string:

1. Algoritma Knuth-Morris-Pratt mulai mencocokkan pattern pada awal teks.
2. Dari kiri ke kanan, algoritma ini akan mencocokkan karakter per karakter pattern dengan karakter di teks yang bersesuaian, sampai salah satu kondisi berikut dipenuhi:
3. Karakter di pattern dan di teks yang dibandingkan tidak cocok (*mismatch*).
4. Semua karakter di pattern cocok. Kemudian algoritma akan memberitahukan penemuan di posisi ini.
5. Algoritma kemudian menggeser pattern berdasarkan tabel next, lalu mengulangi langkah 2 sampai pattern berada di ujung teks.

Berikut adalah pseudocode algoritma KMP pada fase pra-pencarian:

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

Dan berikut adalah pseodocode algoritma KMP pada fase pencarian:

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 interger

Algoritma:

preKMP(n, P, kmpNext)

i:=0

while (i<= 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

next:= j - kmpNext[j]

i:= i+next

endwhile

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 alpabet.

## B. Algoritma Boyer-Moore

Algoritma Boyer-Moore adalah salah satu algoritma pencarian string, dipublikasikan oleh Robert S. Boyer, dan J. Strother Moore pada tahun 1977.

Algoritma ini dianggap sebagai algoritma yang paling efisien pada aplikasi umum. Algoritma Boyer-Moore mulai mencocokkan karakter dari sebelah kanan pattern. Ide di balik algoritma ini adalah bahwa dengan memulai pencocokan karakter dari kanan, dan bukan dari kiri, maka akan lebih banyak informasi yang didapat.

Misalnya ada sebuah usaha pencocokan yang terjadi pada **teks[i..i+n-1]**, dan anggap ketidakcocokan pertama terjadi di antara **teks[i+j]** dan **pattern[j]**, dengan **0 < j < n**. Berarti, **teks[i+j+1..i+n-1] = pattern[j+1..n-1]** dan **a=teks[i+j]** tidak sama dengan **b=pattern[j]**. Penggeseran yang mungkin adalah:

Penggeseran bad-character yang terdiri dari menyejajarkan **teks[i+j]** dengan kemunculan paling kanan karakter tersebut di pattern. Bila karakter tersebut tidak ada di pattern, maka pattern akan disejajarkan dengan **teks[i+n+1]**.

Secara sistematis, langkah-langkah yang dilakukan algoritma Boyer-Moore pada saat mencocokkan string adalah:

* 1. Algoritma Boyer-Moore mulai mencocokkan pattern pada awal teks.
  2. Dari kanan ke kiri, algoritma ini akan mencocokkan karakter per karakter pattern dengan karakter di teks yang bersesuaian, sampai salah satu kondisi berikut dipenuhi:

1. Karakter di pattern dan di teks yang dibandingkan tidak cocok (mismatch).
2. Semua karakter di pattern cocok. Kemudian algoritma akan memberitahukan penemuan di posisi ini.
   1. Algoritma kemudian menggeser pattern nilai penggeseran bad-character, lalu mengulangi langkah 2 sampai pattern berada di ujung teks.

Berikut adalah pseudocode algoritma Boyer-Moore pada fase pra-pencarian:

procedure preBmBc(

input P : array[0..n-1] of char,

input n : integer,

input/output bmBc : array[0..n-1] of integer

)

Deklarasi:

i: integer

Algoritma:

for (i := 0 to ASIZE-1)

bmBc[i] := m;

endfor

for (i := 0 to m - 2)

bmBc[P[i]] := m - i - 1;

endfor

Dan berikut adalah pseudocode algoritma Boyer-Moore pada fase pencarian:

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: integer

BmBc : array[0..255] of interger

Algoritma:

preBmBc(n, P, BmBc)

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;

endif

bmBcShift:= BmBc[chartoint(T[i+j])]-n+j+1

shift:= bmBcShift

i:= i+shift

Tabel untuk penggeseran bad-character dapat dihitung dengan kompleksitas waktu dan ruang sebesar O(n + σ) dengan σ adalah besar ruang alfabet. Sedangkan pada fase pencarian, algoritma ini membutuhkan waktu sebesar O(mn), pada kasus terburuk, algoritma ini akan melakukan 3n pencocokkan karakter, namun pada performa terbaiknya algoritma ini hanya akan melakukan O(m/n) pencocokkan.

# Bab III

# Analisis Pemecahan Masalah

## Spesifikasi Program

1. Aplikasi berbasis web analisis *tweets* yang anda buat menerima 2 masukan yaitu: (1) *keyword* pencarian *tweet* yang akan dianalisis berupa *hashtag* atau *mention*; (2) daftar *keyword*.

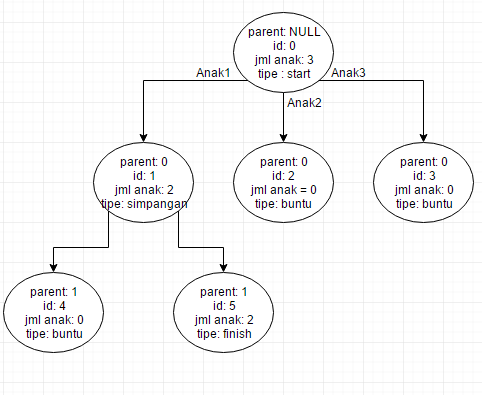
1. Luaran program adalah sebuah laman web baru yang berisi hasil analisis berupa daftar *tweets* untuk setiap setiap dinas terkait, minimal seperti yang ditunjukkan oleh gambar berikut. Pada setiap *tweet* yang dihasilkan, ada pranala ke laman *Twitter* yang berisi *tweet* tersebut.

## Struktur Data

Kami menggunakan dua struktur data khusus untuk memecahkan masalah pencarian jalan keluar ini yaitu pohon dan queue.

1. Pohon

Pohon yang digunakan terdiri dari beberapa node tertentu yang akan membentuk pohon. Node itu sendiri berisi informasi jumlah node anaknya yang berjumlah antara 0 - 3 buah (next node), id dari node itu, serta node yang menjadi parent dari node tersebut (prev node), dan tipe node itu (antara start, finish, buntu, atau persimpangan). Ilustrasinya adalah:

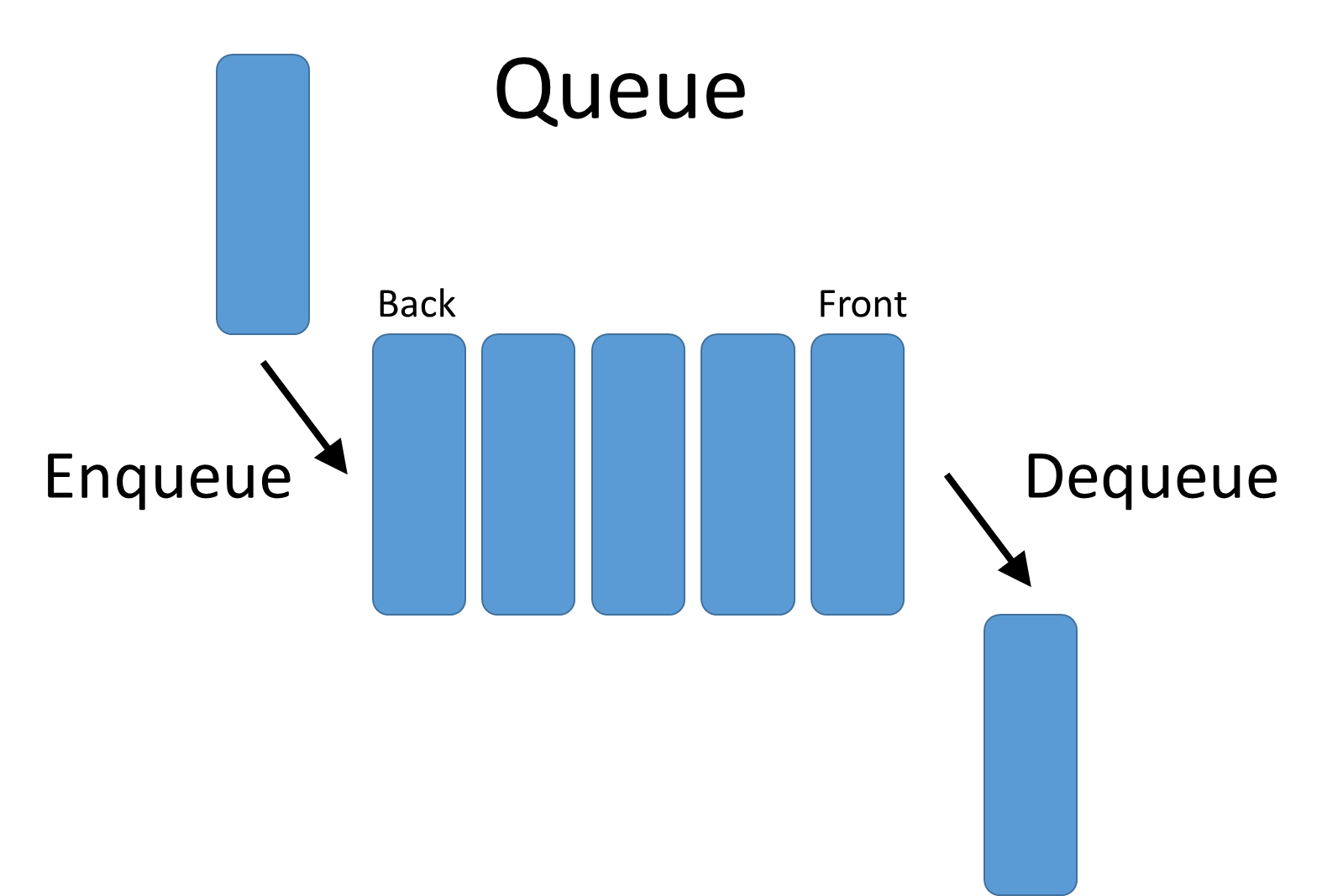


Gambar 6 : Ilustrasi struktur data pohon pada program

Struktur data pohon ini kami gunakan pada algoritma DFS maupun BFS dengan tujuan untuk mencari jalan kembali ke node sebelumnya dan pergi ke node tertentu.

b. Queue

Queue yang digunakan adalah queue biasa/standard (FIFO) dan digunakan untuk menyimpan id dari node pada tree yang akan dikunjungi. Karena itulah queue hanya digunakan pada saat menggunakan algoritma BFS. Node yang sudah dikunjungi pada BFS akan dihapus dari queue. Ilustrasinya sebagai berikut :



Gambar 7 : Ilustrasi struktur data queue pada program

## Pemecahan Masalah

1. DFS

Untuk DFS, pertama kali robot akan berjalan dan mengambil persimpangan paling kanan ketika bertemu dengan setiap persimpangan (warna hijau) yang ada dan kembali ke persimpangan sebelumnya ketika bertemu dengan titik buntu (warna merah). Dengan demikian, dijamin bahwa pada suatu saat robot pasti akan bertemu dengan titik finish (warna biru).

Ketika perjalanan tersebut, robot akan mencatat node-node yang ia lewati dengan bantuan struktur data pohon. Robot akan mencatat berapa cabang yang ada pada setiap persimpangan dengan memutar badan sejauh 360 derajat dan menghitung jumlah garis yang ada. Setelah itu, robot akan berjalan ke persimpangan paling kanan dan menyimpannya di node lain yang telah dibuat. Proses ini akan selalu diulang dan setiap tipe node baik buntu ataupun persimpangan akan terus dicatat sampai robot menemukan node finish.

Setelah mencapai finish, robot akan berputar 180 derajat, berjalan hingga menemukan node persimpangan, lalu mencari jalan untuk mencapai node parent dari node persimpangan ini. Pencarian jalur adalah mencari berapa kali robot harus berputar dan menemukan garis hitam hingga mencapai jalur yang benar, hal ini ditemukan dengan melihat anak ke berapakah cabang itu dari *parent-*nya.

1. BFS

Pertama kali, robot akan berjalan ke persimpangan pertama. Di persimpangan pertama, robot akan berputar, menghitung berapa persimpangan yang ada, lalu menambahkan jalur-jalur tersebut sebagai node baru dalam tree. Jalur-jalur baru tersebut juga akan ditambahkan ke dalam queue, yang menandakan bahwa node tersebut akan dikunjungi nantinya.

Setelah itu, robot akan mengunjungi node yang berada di head queue. Setelah tiba di sana, robot akan mengecek apakah node itu node buntu, persimpangan, atau finish lalu menambahkan node tersebut ke pohon. Jika merupakan node persimpangan, maka robot akan berputar dan mencatat berapa persimpangan yang ada lalu menambahkan setiap jalur yang ada ke queue; jika node adalah node finish maka program pencarian akan berhenti; jika node adalah node buntu maka program tidak akan melakukan apapun. Setelah mencapai dan memeriksa node tujuan, robot akan pergi ke node yang ada di head queue lalu melakukan pengecekan yang sama seperti langkah sebelumnya.

Jika robot telah mencapai node finish, robot akan berhenti sejenak dan mencari jalan untuk “pulang” kembali ke node start. Pencarian jalur dan jalannya robot ke node start ini menggunakan algoritma dan struktur data serta implementasi program yang sama seperti yang digunakan pada algoritma DFS.

# BAB IV

# Implementasi dan Pengujian

## Implementasi Struktur Data

* 1. Pohon

Karena pada IDE ROBOTC ini tidak bisa dilakukan alokasi memori dinamis untuk alamat node pohon, kami menggunakan sebuah array of node sebagai representasi dari struktur data pohon ini. Kami mengimplementasikan setiap node yang ada dalam ROBOTC sebagai berikut:

typedef struct {

int prev;

int next[3];

int nIntersect;

int nPassed;

char tipe;

} Node;

Untuk ID dari setiap node, kami memanfaatkan indeks dari array agar dapat mengakses elemen dengan ID tertentu dengan cepat (tidak perlu proses traversal).

b. Queue

Kami menggunakan modifikasi dari ADT Queue representasi array secara sirkuler seperti yang pernah dibuat pada praktikum mata kuliah Algoritma dan Struktur data. Setiap elemen queue mencerminkan id dari setiap node yang akan dikunjungi.

## Modul Program

Modularitas program dibagi sebagai berikut :

* *boolean.h*

Sebuah file header yang digunakan untuk mendefinisikan tipe data boolean yang tidak ada secara default dalam bahasa pemrograman C. Digunakan baik untuk algoritma DFS maupun DFS.

* *Queue.c*

File yang berisi deklarasi sekaligus implementasi struktur data Queue dan segala operasi yang mungkin dilakukan. Digunakan hanya pada algoritma BFS.

* *RobotMovement.c*

Kami membuat sebuah modul khusus bernama RobotMovement yang mengatur berbagai pergerakan robot dan cara robot mengenali warna melalui sensor warna. Modul ini sendiri digunakan oleh kedua program yang menangani algoritma DFS dan BFS sehingga menghindari redundansi modul.

* *DFS.c*

Modul ini menangani pergerakan robot secara DFS, mulai dari arah/cabang yang dituju robot hingga finish, penyimpanan data dalam pohon, hingga kembalinya robot ke titik start setelah robot menemukan node finish.

* *BFS.c*

Modul ini menangani pergerakan robot secara BFS, mulai dari arah/cabang yang dituju robot hingga finish, penyimpanan data dalam pohon, hingga kembalinya robot ke titik start setelah robot menemukan node finish. Baik untuk algoritma DFS maupun BFS, kami memrogram robot untuk memilih cabang paling kanan terlebih dahulu (robot akan memriotaskan arah kanan pertama untuk setiap persimpangan).

## Fungsi dan Prosedur

Semua fungsi dan prosedur berikut ada pada file RobotMovement.c :

1. GetColor -> untuk menentukan jenis warna yang dideteksi sensor warna

2. turnAndCount -> untuk menghitung jumlah cabang pada persimpangan

3. turnTilBlack -> agar robot berputar hingga mendeteksi warna hitam

4. turnTilWhite -> agar robot berputar hingga mendeteksi warna putih

5. turnTilDegree -> agar robot berputar hingga nilai sensor gyro tertentu

6. lineTrackForTime -> agar robot bergerak secara line follower dengan durasi tertentu

7. findFinish -> untuk mencari node final pada pohon yang terbentuk

8. PrintJalurBenar -> untuk output jalur ke finish yang benar pada EV3 remote screen

9. findJmlBelok -> untuk menentukan berapa kali robot harus berputar

10. goUntilGreen -> agar robot bergerak secara line follower hingga node hijau

11. goUntilBlue -> agar robot bergerak secara line follower hingga node biru

12. goForward -> agar robot bergerak dari suatu node ke salah satu node anaknya

13. goPrev -> agar robot bergerak dari suatu node ke node parentnya

14. findSameParent -> untuk mencari parent yang sama dari dua buah node

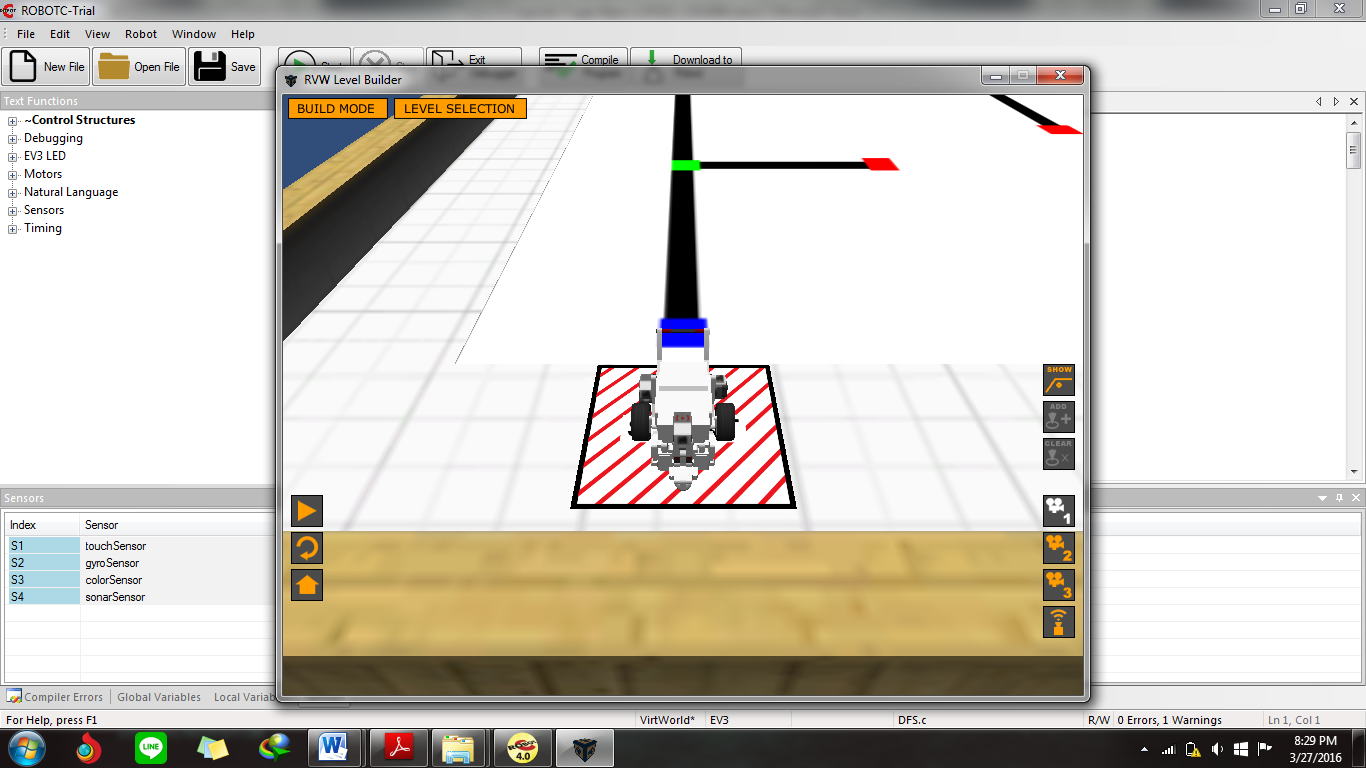
15. findSameParent -> untuk mencari parent yang sama dari dua buah node

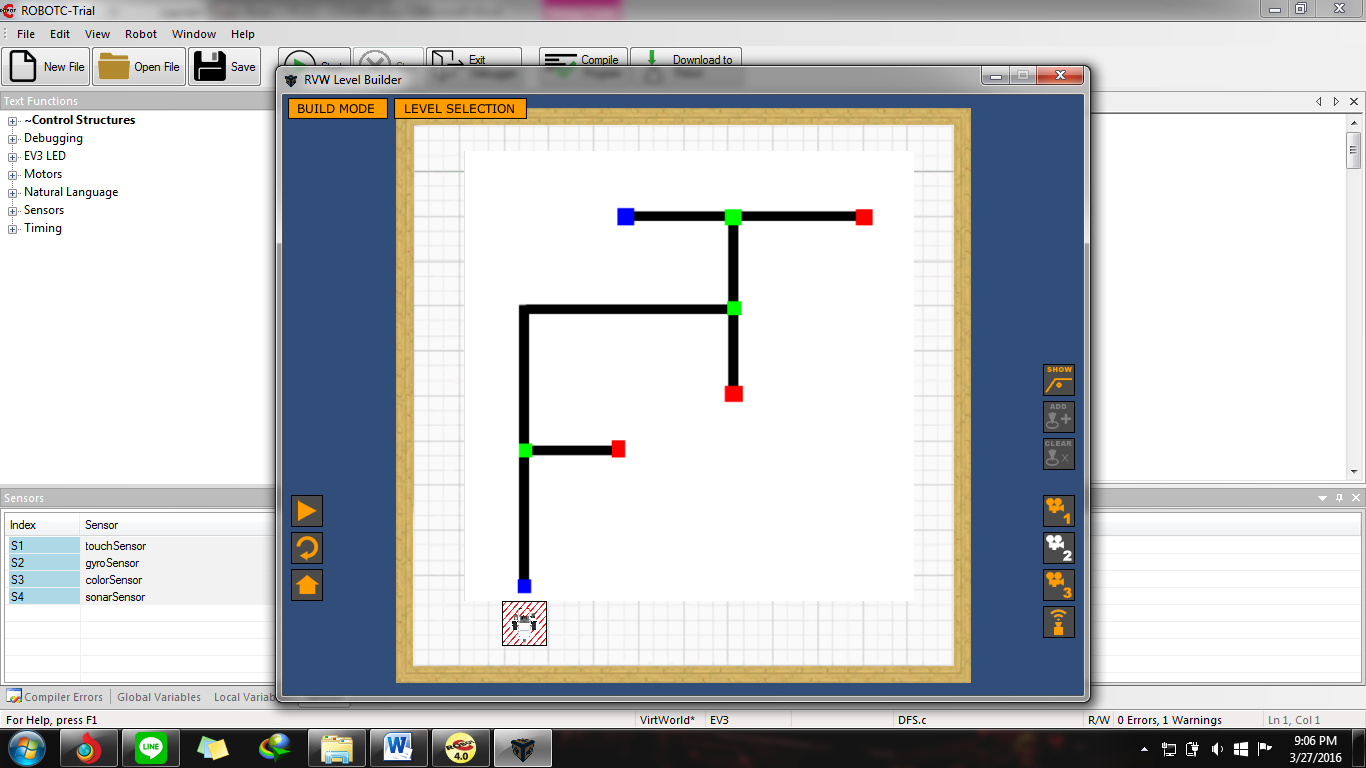
16. findJalanAwal -> agar robot bergerak kembali ke start setelah mencapai finish

17. gotoNode -> agar robot bergerak dari suatu node ke node yang lain

18. PrintJalurDitempuh -> untuk output jalur yang sudah ditempuh robot

## D. Screenshot Antarmuka Program

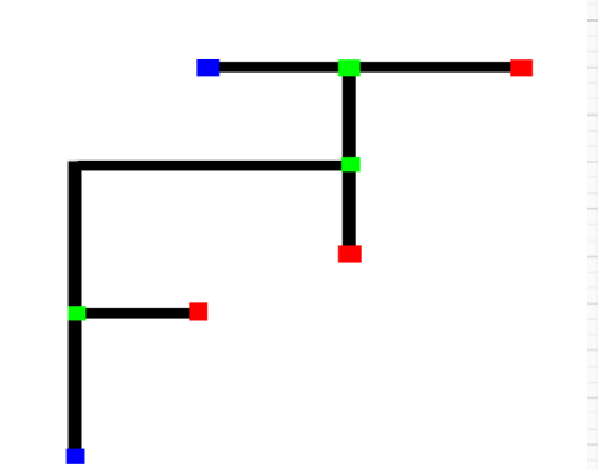




Gambar 8 : Antarmuka program tampak mendatar dan tampak atas

## E. Pengujian

Berikut ditampilkan hasil pengujian terhadap labirin yang diberikan asisten :



7

3

4

5

6

2

0

1

Gambar 9 : Peta labirin yang diuji

Jalur yang dilalui robot untuk masing-masing algoritma adalah sebagai berikut :

1. DFS : 0 – 1 – 2 – 1 – 3 – 4 – 3 – 5 – 6 – 5 – 7

2. BFS : 0 – 1 – 2 – 1 – 3 – 1 – 3 – 4 – 3 – 5 – 3 – 5 – 6 – 5 – 7

## F. Analisis

Algoritma BFS maupun DFS adalah sebuah algoritma yang digunakan untuk mencari solusi dari sebuah permasalahan yang memiliki model struktur data berbentuk graf/pohon. Namun, solusi yang dihasilkan tidak selalu merupakan solusi paling optimal, tidak seperti algoritma pencarian graf yang lain semisal algoritma Dijkstra. Hal ini dikarenakan algoritma BFS dan DFS hanya mencari solusi pertama yang ditemukan setelah menelusuri graf tanpa memperhitungkan *cost* yang harus dibayar. Berbeda dengan algoritma Dijkstra yang mampu menentukan solusi yang meminimalkan *cost*.

Dibalik kekurangan algoritma BFS dan DFS, terdapat pula kelebihan yang dapat dimanfaatkan. Algoritma Dijkstra hanya mampu melakukan traversal pada graf yang sudah terbentuk (graf statis) karena ia memeriksa matriks ketetanggaannya dan tidak mampu melakukan traversal graf pada graf yang belum terbentuk. Sedangkan, algoritma BFS dan DFS mampu melakukan traversal pada graf statis maupun graf dinamis (graf yang baru akan dibentuk ketika program dieksekusi bersamaan dengan proses traversal).

Masalah pencarian jalan keluar labirin ini lebih cocok untuk diselesaikan dengan algoritma BFS dan/atau DFS. Hal ini karena graf belum terbentuk sebelum program dieksekusi (robot tidak mengetahui gambaran labirin secara keseluruhan pada awal program). Pada saat eksekusi program, robot akan mencari-cari jalan sambil membentuk graf dinamis setiap kali menemukan node merah, hijau, atau biru.

Dari hasil eksekusi di atas, kita dapat lihat bahwa DFS menemukan solusi keluar labirin yang lebih cepat dibandingkan algoritma BFS. Secara umum hal ini benar karena dengan menggunakan algoritma BFS, robot perlu melakukan gerakan bolak-balik yang berulang-ulang pada suatu percabangan. Jika labirin diilustrasikan sebagai pohon, gerakan bolak-balik ini bisa dimengerti karena untuk melakukan traversal pada sebuah level pohon, maka traversal dilakukan bolak-balik dari parent node ke node anak, kembali ke node parent, kemudian ke node-node anak yang lainnya (dalam hal ini, node parent adalah node hijau untuk setiap percabangan).

Namun, ada kasus labirin khusus yang akan membuat algoritma DFS tidak lebih efektif dari algoritma BFS. Kasus itu adalah ketika suatu cabang yang ditelusuri memiliki banyak cabang anak dan tidak ada satu pun yang mengarah ke solusi. Algoritma DFS akan terus menelusuri hingga cabang terdalam (jalan buntu) dan sangat menghabiskan waktu. Hal ini sesuai dengan sifat natural algoritma DFS yang tidak selalu menemukan solusi jika jumlah cabang tak hingga dan tak mengarah ke solusi, sedangkan algoritma BFS lebih terukur dalam menentukan solusi walau memang secara umum memakan waktu yang lebih lama karena proses traversal yang bolak-balik antara node parent dan node anak.

# BAB V

# Kesimpulan dan Saran

## Kesimpulan

Algoritma Knuth-Morris-Prath dan/atau Boyer-Moore adalah algoritma yang tepat untuk menyelesaikan masalah pencocokan string. Kedua algoritma ini lebih cepat dibandingkan dengan Brute Force. Algoritma KMP tidak pernah bergerak mundur pada teks input, hal ini menyebabkan algoritma ini bagus untuk memproses file yang sangat besar dan dibaca dari eksternal. Namun algoritma KMP tidak bekerja cukup baik dengan meningkatnya variasi alfabet. Hal ini karena semakin tingginya kemungkinan *mismatch* dan terjadinya *mismatch* cenderung terjadi pada tiap awal pattern. Namun KMP bekerja cepat jika mismatch terjadi di akhir *pattern.* Sebaliknya dengan algoritma Boyer-Moore, pada Boyer-Moore bekerja dengan cepat jika variasi alfabetnya banyak dan bekerja lambat jika variasi alfabetnya sedikit.

## Saran

Secara umum, apabila kita ingin melalukan pencocokan string pada text biasa, sebaiknya menggunakan algoritma Boyer-Moore daripada KMP. Algorima KMP lebih baik untuk pencarian pada string seperti DNA (string dengan variasi alfabet sedikit).

# Daftar Pustaka

Munir, Rinaldi. 2014. Slide Kuliah IF2211 Strategi Algoritma : Pencocokan String (2015). Bandung : Institut Teknologi Bandung.

http://www-igm.univ-mlv.fr/~lecroq/string/node8.html diakses 22 April 2016 pukul 10.00.

http://www-igm.univ-mlv.fr/~lecroq/string/node14.html diakses 22 April 2016 pukul 10.00.