APLIKASI DFS DAN BFS: ROBOT LEGO MINDSTORMS EV3 PENCARI SUMBER API DI DALAM LABIRIN

LAPORAN

Diajukan untuk memenuhi Tugas Besar II IF2211 Strategi Algoritma

oleh

KELOMPOK TOWERBRIDGE:

FELIX LIMANTA / 13515065

RIONALDI CHANDRASETA / 13515077

HOLY LOVENIA / 13515113

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2017

DAFTAR ISI

DAFTAR GAMBAR

BAB I
DESKRIPSI TUGAS

Sebuah robot diberi tugas menemukan api di dalam sebuah ruangan. Robot mempunyai sensor yang dapat mengetahui ada sumber api di dalam sebuah ruangan. Ruangan itu berbentuk lorong-lorong yang berliku-liku laksana sebuah labirin (*maze*). Robot melewati labirin sampai berhasil menemukan api (dan kalau bisa memadamkannya). [1]

Pada tugas kali ini, mahasiswa diminta membuat program simulasi robot (menggunakan robot *Lego*) yang dapat menemukan sumber api di dalam labirin yang telah disiapkan, dengan memanfaatkan algoritma DFS dan BFS. Berikut akan dijelaskan tugas yang dikerjakan secara detail.

Robot Lego yang digunakan adalah *Lego Mindstorms EV3*. Karena ketersediaan Robot *Lego Mindstorms EV3* terbatas, maka uji coba algoritma dilakukan terlebih dahulu pada program simulasi *Robot Virtual World* dengan IDE Pemrograman ROBOTC. Setelah berhasil menjalankan algoritma pada robot virtual, maka selanjutnya (opsional) anda dapat mengujinya pada robot *Lego Mindstorms* yang dapat dipinjam di Lab Grafika dan Intelegensia Buatan (GAIB) secara bergiliran.

Labirin yang akan digunakan dalam permainan memiliki spesifikasi sebagai berikut:

1. Labirin terdiri dari jalur yang yang berupa garis berwarna hitam. Robot hanya diperbolehkan berada di atas garis berwarna hitam.
2. Labirin memiliki 1 portal masuk dan 1 portal keluar.
3. Terdapat 4 warna penanda pada jalur yang memiliki arti masing-masing, yaitu:
    a. Warna biru adalah penanda pintu masuk dan pintu keluar.
    b. Warna hijau digunakan sebagai penanda sebuah persimpangan (dianggap sebagai sebuah *node* pada *tree*). Setiap persimpangan terdiri dari 2-3 cabang.
    c. Warna merah digunakan sebagai penanda jalan buntu.
    d. Lokasi api (jika ada) ditandai pada sebuah titik yang berwarna kuning

BAB II
DASAR TEORI

2.1 Gambaran Umum

Algoritma traversal dalam graf adalah algoritma yang mengunjungi simpul-simpul yang ada dalam graf secara sistematik. Pencarian dilakukan dari simpul sumber hingga menemukan simpul target. Algoritma traversal graf menggunakan suatu struktur data untuk menampung simpul-simpul yang belum pernah dieksplorasi. Pada setiap iterasi, sebuah simpul dari struktur data tersebut diproses, lalu tetangga-tetangga dari simpul tersebut akan ditambahkan ke dalam struktur data. Ada beberapa jenis algoritma traversal dalam graf, seperti *Depth-First Search* (DFS) dan *Breadth-First Search* (BFS). [2]

Struktur pencarian solusi berupa pembentukan pohon dinamis, yaitu pohon ruang status yang dibentuk saat proses pencarian dilakukan. Algoritma memeriksa apakah solusi sudah tercapai atau tidak pada setiap simpul. Jika simpul tersebut merupakan simpul solusi, pencarian dapat berakhir (satu solusi) atau diteruskan untuk mencari solusi lain (semua solusi). Berikut ini adalah representasi dari pohon dinamis. [3]

1. Status persoalan (*problem state*): simpul-simpul di dalam pohon dinamis yang memenuhi kendala (*constraints*) sehingga layak untuk membentuk solusi, akar dari pohon merupakan *initial state*.
2. Status solusi (*solution state*): satu atau lebih status yang menyatakan solusi persoalan, yaitu setiap daun pada pohon tersebut.
3. Status tujuan (*goal state*): simpul daun yang dituju.
4. Ruang solusi (*solution space*): himpunan atas semua status solusi.
5. Ruang status (*state space*): seluruh simpul pada pohon dinamis, pohonnya dinamakan pohon ruang status (*state space tree*).

2.2 *Depth-First Search* (DFS)

Algoritma DFS mencari secara mendalam dengan mengunjungi setiap simpul yang dapat dicapai dari simpul target sesuai dengan urutan kedalamannya.

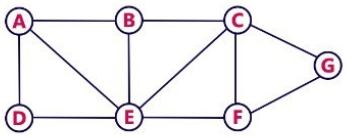
Apabila penelusuran dimulai dari simpul *v*, maka algoritma DFS iteratif adalah sebagai berikut.

1. Mendefinisikan sebuah *stack* dengan ukuran sebesar jumlah dari semua simpul dalam graf.
2. Mengunjungi simpul *v* dan memasukkannya ke dalam *stack*.
3. Mengunjungi salah satu tetangga yang belum dikunjungi dari simpul yang berada pada puncak *stack*, lalu memasukkannya ke dalam *stack*.
4. Mengulangi langkah 3 hingga tidak ada lagi tetangga yang belum dikunjungi dari simpul yang berada pada puncak *stack*.
5. Ketika pencarian sudah mencapai simpul *u* (simpul yang semua tetangganya sudah pernah dikunjungi), pencarian

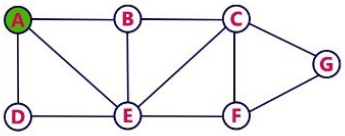
akan dirunut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mencari simpul tetangga lain yang belum dikunjungi dengan cara mem-*pop* simpul dari *stack*.

6. Mengulangi langkah 3, 4, dan 5 hingga *stack* kosong.
7. Ketika *stack* sudah kosong, algoritma akan menghasilkan pohon merentang dengan cara menghapus sisi yang tidak terpakai pada graf. Pohon merentang tersebut adalah jalur yang terbentuk menggunakan DFS.

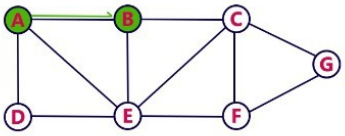
Consider the following example graph to perform DFS traversal



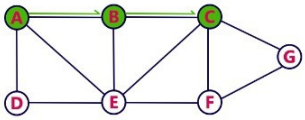
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



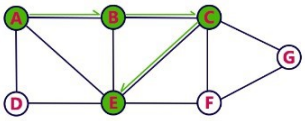
Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



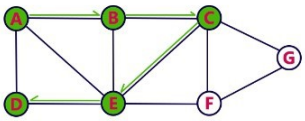
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



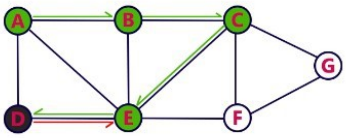
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

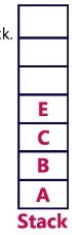


Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

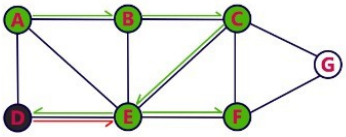


Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

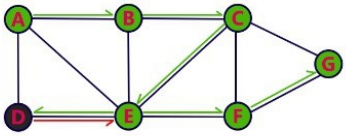




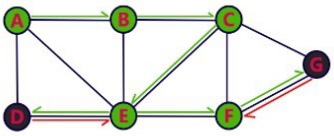
Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



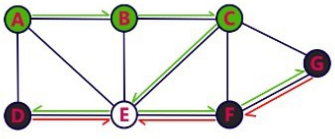
Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

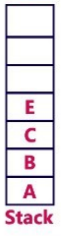


Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

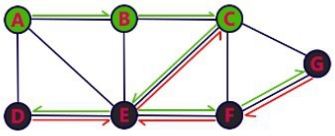


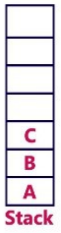
Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

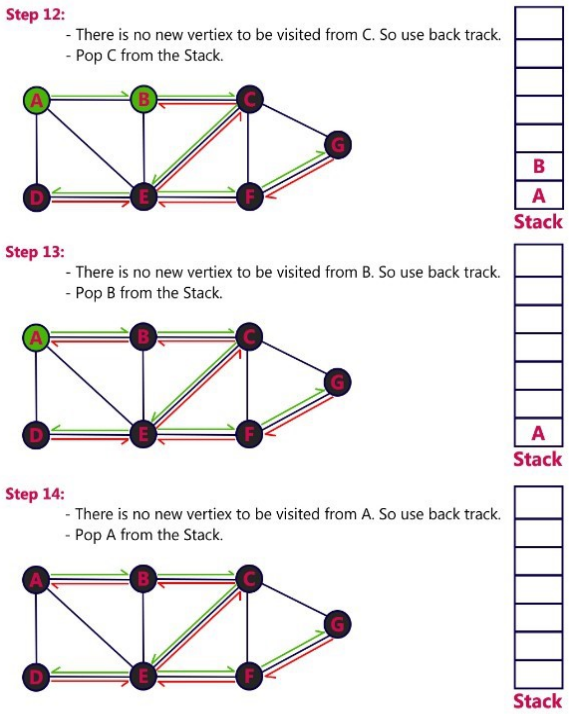


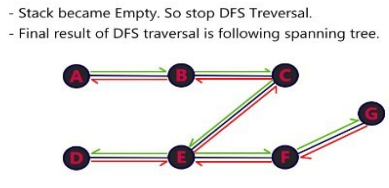


Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.









Gambar 1 Algoritma DFS [7]

Berikut ini adalah *pseudocode* dari algoritma rekursif DFS.

```
procedure DFS(input v: integer)
{ Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS
Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi ditulis ke layar
}

Deklarasi
w: integer

Algoritma
write(v)
dikunjungi[v] ← true
for (w ← 1 to n) do
        if (A[v,w] = 1) then { simpul v dan simpul w bertetangga }
                if (not dikunjungi[w]) then
                        DFS(w)
                endif
        endif
endfor
```

2.3 *Breadth-First Search* (BFS)

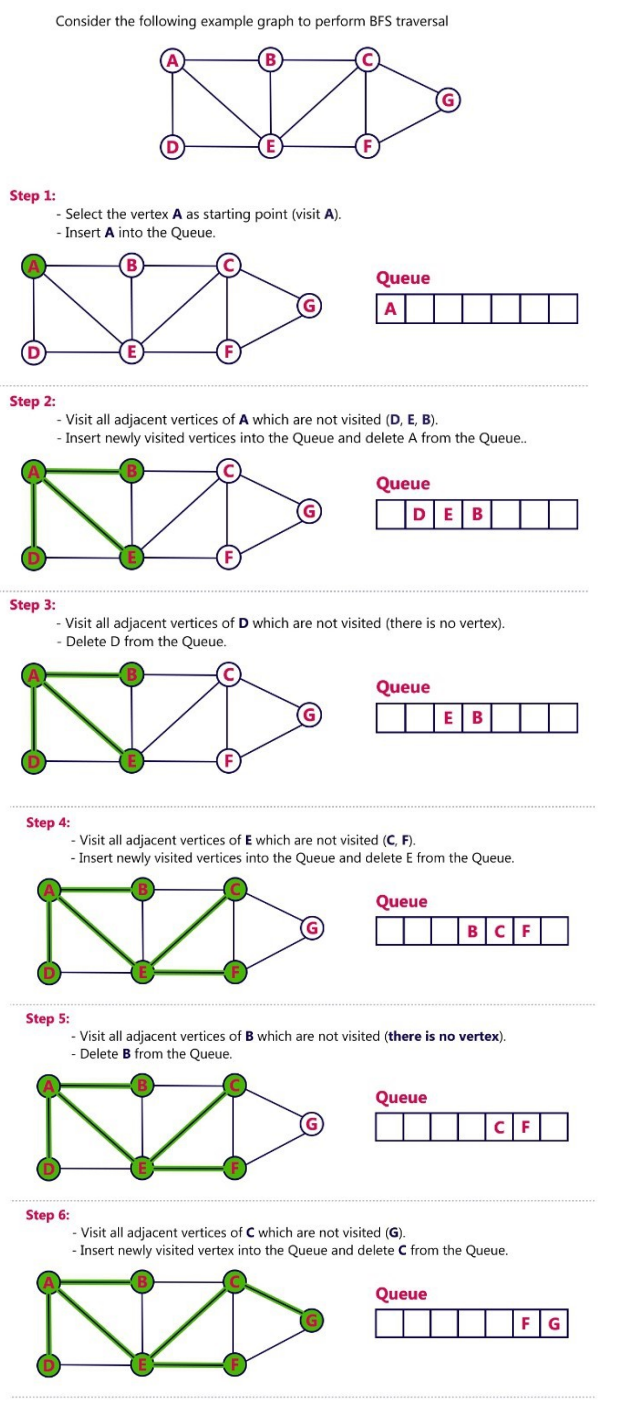
Algoritma BFS mencari secara melebar. Algoritma ini bertujuan melakukan penelusuran sedekat mungkin dengan simpul target. Oleh karena itu, simpul-simpul yang diproses terlebih dahulu adalah yang bertetangga dengan simpul target.

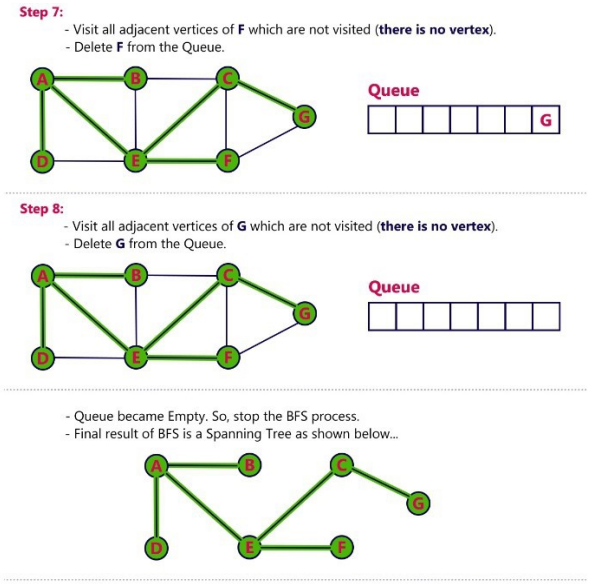
Apabila penelusuran dimulai dari simpul *v*, maka algoritma BFS adalah sebagai berikut.

1. Mendefinisikan sebuah *queue* dengan ukuran sebesar jumlah dari semua simpul dalam graf.
2. Mengunjungi simpul *v* dan memasukkannya ke dalam *queue*.
3. Mengunjungi salah satu tetangga yang belum dikunjungi dari simpul yang berada pada kepala *queue*, lalu

memasukkannya ke dalam *queue*.

4. Ketika tidak ada tetangga yang dapat dikunjungi lagi dari simpul yang berada pada kepala *queue*, simpul tersebut akan dihapus dari *queue*.
5. Mengulangi langkah 3 dan 4 hingga *queue* kosong.
6. Ketika *queue* sudah kosong, algoritma akan menghasilkan pohon merentang dengan cara menghapus sisi yang tidak terpakai pada graf. Pohon merentang tersebut adalah jalur yang terbentuk menggunakan BFS.





Gambar 2 Algoritma BFS [7]

Berikut ini adalah *pseudocode* dari algoritma BFS.

```
procedure BFS(input v: integer)
{ Traversal graf dengan algoritma pencarian BFS.
Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi dicetak ke layar
}

Deklarasi
w: integer
q: antrian
procedure BuatAntrian(input/output q: antrian) { membuat antrian kosong, kepala(q) diisi 0 }
procedure MasukAntrian(input/output q: antrian, input v: integer) { memasukkan v ke dalam
antrian q pada posisi belakang }
procedure HapusAntrian(input/output q: antrian, output v: integer) { menghapus v dari kepala
antrian q }
function AntrianKosong(input q: antrian) → boolean { true jika antrian q kosong, false jika
sebaliknya }

Algoritma
BuatAntrian(q) { buat antrian kosong }
write(v) { cetak simpul awal yang dikunjungi }
dikunjungi[v] ← true { simpul v telah dikunjungi, tandai dengan true }
MasukAntrian(q,v) { masukkan simpul awal kunjungan ke dalam antrian}

{ kunjungi semua simpul graf selama antrian belum kosong }
while (not AntrianKosong(q)) do
        HapusAntrian(q,v) { simpul v telah dikunjungi, hapus dari antrian }
        for (w ← 1 to n) do
                if (A[v,w] = 1) then { v dan w bertetangga }
                        if (not dikunjungi[w]) then
                                write(w) {cetak simpul yang dikunjungi}
                                MasukAntrian(q,w)
                                dikunjungi[w] ← true
                        endif
                endif
        endfor
endwhile
{ AntrianKosong(q) }
```

BAB III
ANALISIS PEMECAHAN MASALAH

3.1 Langkah Pemecahan Masalah

Pada permasalahan ini, algoritma DFS dan BFS digunakan untuk menelusuri labirin dari petak masuk untuk mencari sumber api (petak berwarna kuning). Setiap persimpangan (petak berwarna hijau) pada labirin dianggap sebagai cabang dan arah-arah yang berada di persimpangan tersebut sebagai anak dari simpul arah datang (tempat robot berada saat ini) yang ada pada pohon ruang status. Petak berwarna merah dianggap sebagai simpul daun, yaitu tanda berakhirnya lintasan tersebut. Jalur yang bisa dilalui robot ditandai oleh garis hitam.

Ketika menggunakan algoritma DFS, robot akan melakukan pencarian dengan cara mencoba setiap kemungkinan jalan yang ada hingga menemukan jalan buntu (petak merah), lalu melanjutkan pencarian dengan menelusuri jalan lain yang tersedia pada persimpangan terakhir yang dilewati. Pencarian selesai apabila robot telah berhasil menemukan api (petak kuning), atau robot telah menelusuri semua kemungkinan jalan yang ada dan sudah kembali ke titik awal (kasus tanpa api).

Pada algoritma BFS, robot akan melakukan pencarian dengan cara mencoba setiap kemungkinan jalan yang ada hingga menemukan persimpangan lainnya (petak hijau) atau jalan buntu (petak merah), lalu kembali menelusuri kemungkinan jalur lainnya hingga semua jalur pada persimpangan tersebut sudah dicoba. Selanjutnya, robot akan mencoba jalur-jalur yang dapat dilewati pada persimpangan berikutnya. Pencarian selesai apabila robot telah berhasil menemukan api (petak kuning), atau robot telah menelusuri semua kemungkinan jalan yang ada dan sudah kembali ke titik awal (kasus tanpa api).

Dalam pencarian, urutan penelusuran jalur yang dilakukan robot selalu dimulai dari jalur yang berada paling kiri (berdasarkan sudut pandang robot) ke jalur terkanan. Selama pencarian, robot akan menyimpan jalur yang digunakan untuk mencapai api ke dalam sebuah *stack*. Hal ini dilakukan untuk mempermudah robot untuk menemukan jalan kembali ke titik awal, yaitu mem-*pop* aksi-aksi yang dilakukan robot dari stack tersebut.

3.2 Struktur Data

Struktur data yang digunakan dalam implementasi adalah *stack* dan *queue*. *Queue* merupakan tipe bentukan yang memiliki metode FIFO (*First In First Out*) untuk keluar-masuk data. Struktur data ini dipakai untuk mendukung implementasi dari algoritma BFS. Beberapa fungsi yang berada dalam struktur data *queue* adalah *enqueue* (menambahkan elemen ke urutan paling belakang) dan *dequeue* (mengambil elemen yang berada di posisi paling awal).

Stack adalah tipe bentukan yang memiliki metode LIFO (*Last In First Out*) untuk keluar-masuk data. Struktur data ini digunakan dalam implementasi algoritma DFS dan proses penelusuran jalan pulang ke titik awal. Beberapa fungsi yang berada dalam struktur data *stack* adalah *push* (menambahkan elemen ke *top stack*) dan *pop* (mengambil elemen yang berada di *top stack*).

3.3 Spesifikasi Program

Program dibuat menggunakan Lego Mindstorm Simulator. Bahasa yang digunakan adalah C. Pengerjaan dilakukan secara paralel dan tatap muka memakai laptop masing-masing. Algoritma yang digunakan untuk membentuk kecerdasan buatan milik robot dimasukkan ke file bernama dfs.c dan bfs.c.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi

4.1.1 Program

4.1.1.1 *Depth-First Search* (DFS)

Berkas program yang digunakan untuk implementasi algoritma DFS adalah stackt.h dan dfs.c. Berikut ini adalah beberapa cuplikan layar dari antarmuka program.

```
stackt.h | dfs.c
1  #include "stackt.h"
2
3  #pragma config(Sensor, S1,     touchSensor,    sensorEV3_Touch)
4  #pragma config(Sensor, S2,     gyroSensor,     sensorEV3_Gyro)
5  #pragma config(Sensor, S3,     colorSensor,    sensorEV3_Color, modeEV3Color_Color)
6  #pragma config(Sensor, S4,     sonarSensor,    sensorEV3_Ultrasonic)
7  #pragma config(Motor,  motorA,          armMotor,       tmotorEV3_Large, PIDControl, encoder)
8  #pragma config(Motor,  motorB,          leftMotor,      tmotorEV3_Large, PIDControl, driveLeft, encoder)
9  #pragma config(Motor,  motorC,          rightMotor,     tmotorEV3_Large, PIDControl, driveRight, encoder)
10 //*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
11
12 #define TOLERANCE 15
13
14 Stack path;
15
16 int r, g, b;
17 int begin_degree;
18
19 int getDegrees(short gyro_sensor) {
20   int gyroReading = getGyroDegrees(gyro_sensor);
21   if (gyroReading > 360) {
22     gyroReading %= 360;
23   } else if (gyroReading < 0) {
24     gyroReading %= 360;
25     gyroReading += 360;
```

Gambar 3 File stackt.h

```
stackt.h | dfs.c
1  /* File : stackt.h */
2  /* deklarasi stack yang diimplementasi dengan tabel kontigu dan ukuran sama */
3  /* TOP adalah alamat elemen puncak */
4  /* Implementasi dalam bahasa C dengan alokasi statik */
5  #ifndef stackt_H
6  #define stackt_H
7
8  #define Nil 0
9  #define MaxEl 20
10 /* Nil adalah stack dengan elemen kosong . */
11 /* Karena indeks dalam bhs C dimulai 0 maka tabel dg indeks 0 tidak dipakai */
12
13 typedef struct {
14   int num;
15   int degree;
16 } node;
17 typedef int address;   /* indeks tabel */
18
19 /* Contoh deklarasi variabel bertype stack dengan ciri TOP : */
20 /* Versi I : dengan menyimpan tabel dan alamat top secara eksplisit*/
21 typedef struct {
22   node T[MaxEl+1]; /* tabel penyimpan elemen */
23   address TOP;  /* alamat TOP: elemen puncak */
24 } Stack;
25 /* Definisi stack S kosong : S.TOP = Nil */
26 /* Elemen yang dipakai menyimpan nilai Stack T[1]..T[MaxEl] */
27 /* Jika S adalah Stack maka akses elemen : */
28    /* S.T[(S.TOP)] untuk mengakses elemen TOP */
29    /* S.TOP adalah alamat elemen TOP */
30
31 /* Definisi akses dengan Selektor : Set dan Get */
```

Gambar 4 File dfs.c

4.1.1.2 *Breadth-First Search* (BFS)

Berkas program yang digunakan untuk implementasi algoritma BFS adalah stackt.h dan bfs.c. Berikut ini adalah beberapa cuplikan layar dari antarmuka program.

```
bfs.c | queue.h
1  /* File : queue.h */
2  /* Definisi ADT Queue dengan representasi array secara eksplisit dan alokasi dinamik */
3  /* Model Implementasi Versi III dengan circular buffer */
4
5  #ifndef queue_H
6  #define queue_H
7
8  #define Nil_Q 0
9  #define MaxEl_Q 20
10 /* Konstanta untuk mendefinisikan address tak terdefinisi */
11
12 /* Definisi elemen dan address */
13 typedef struct {
14   int num;
15   int degree;
16 } node_q;
17
18 typedef int address;   /* indeks tabel */
19 /* Contoh deklarasi variabel bertype Queue : */
20 /* Versi I : tabel dinamik, Head dan Tail eksplisit, ukuran disimpan */
21 typedef struct {
22   node T[MaxEl+1]; /* tabel penyimpan elemen */
23   address HEAD;  /* alamat TOP: elemen puncak */
24   address TAIL;
25 } Queue;
26 /* Definisi Queue kosong: HEAD=Nil; TAIL=Nil. */
```

Gambar 5 File queue.h

```
bfs.c | queue.h
1  #include "stackt.h"
2  #include "queue.h"
3
4  #pragma config(Sensor, S1,     touchSensor,    sensorEV3_Touch)
5  #pragma config(Sensor, S2,     gyroSensor,     sensorEV3_Gyro)
6  #pragma config(Sensor, S3,     colorSensor,    sensorEV3_Color, modeEV3Color_Color)
7  #pragma config(Sensor, S4,     sonarSensor,    sensorEV3_Ultrasonic)
8  #pragma config(Motor,  motorA,          armMotor,       tmotorEV3_Large, PIDControl, encoder)
9  #pragma config(Motor,  motorB,          leftMotor,      tmotorEV3_Large, PIDControl, driveLeft, encoder)
10 #pragma config(Motor,  motorC,          rightMotor,     tmotorEV3_Large, PIDControl, driveRight, encoder)
11 //*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
12
13 #define TOLERANCE 15
14
15 Stack path;
16 Queue path_q;
17
18 int r, g, b;
19 int begin_degree;
20
21 int getDegrees(short gyro_sensor) {
22   int gyroReading = getGyroDegrees(gyro_sensor);
23   if (gyroReading > 360) {
24     gyroReading %= 360;
25   } else if (gyroReading < 0) {
26     gyroReading %= 360;
```

Gambar 6 File bfs.c

4.1.2 Kakas

Kakas yang digunakan untuk implementasi program adalah ROBOTC beserta *Robot Virtual Worlds*. Paket perangkat lunak ini terdiri dari 3 buah bagian:

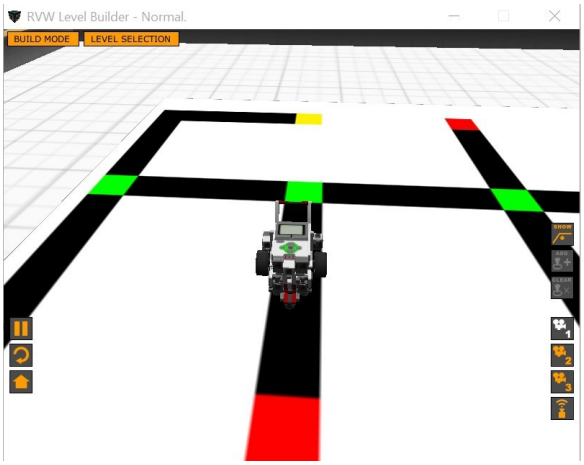
1. ROBOTC digunakan untuk memrogram dan mensimulasikan *Lego Mindstorms EV3*.
2. *RVW Level Editor* digunakan untuk mengubah dan membuat level (arena) untuk memberikan tantangan baru untuk robot.
3. *Virtual Brick* adalah *software* simulasi *Lego Mindstorms EV3* (hanya simulator, tidak termasuk IDE) yang digunakan untuk keperluan *testing*.

4.2 Pengujian

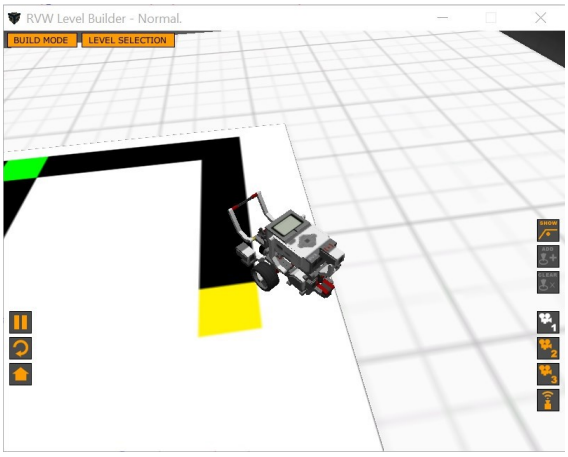
Secara umum, program dapat berjalan dengan baik (robot dapat menemukan api maupun kembali ke titik awal) menggunakan algoritma DFS ataupun BFS. Dari keseluruhan pengujian, beberapa kali kegagalan terjadi. Hal ini dikarenakan sensor tidak berhasil membaca warna yang ada pada petak.

4.2.1 Algoritma DFS

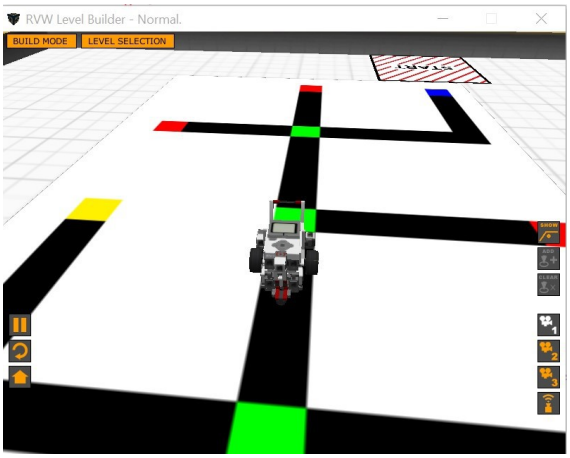
1. Rute normal



Gambar 7 Algoritma DFS: Rute Normal (Awal)

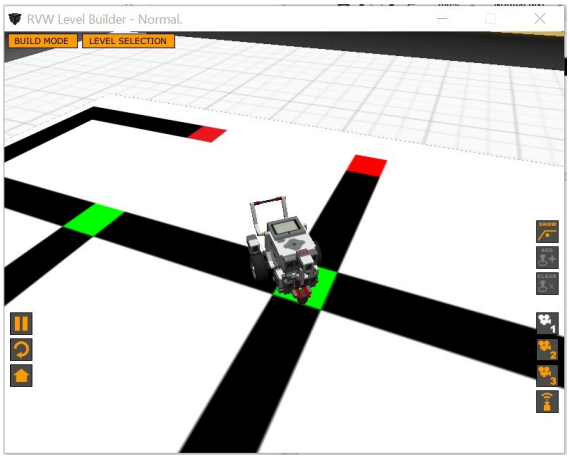


Gambar 8 Algoritma DFS: Rute Normal (Tengah)

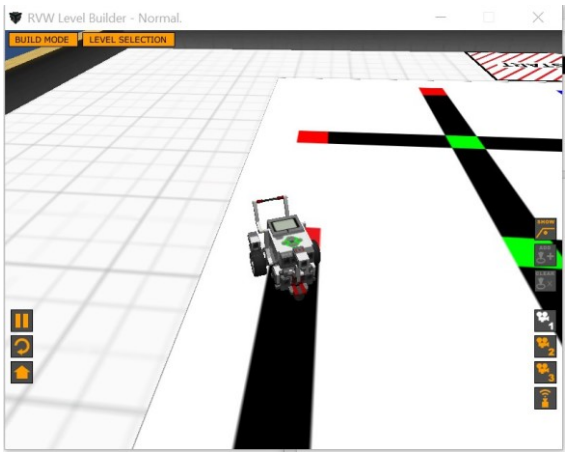


Gambar 9 Algoritma DFS: Rute Normal (Tengah)

2. Rute tanpa api

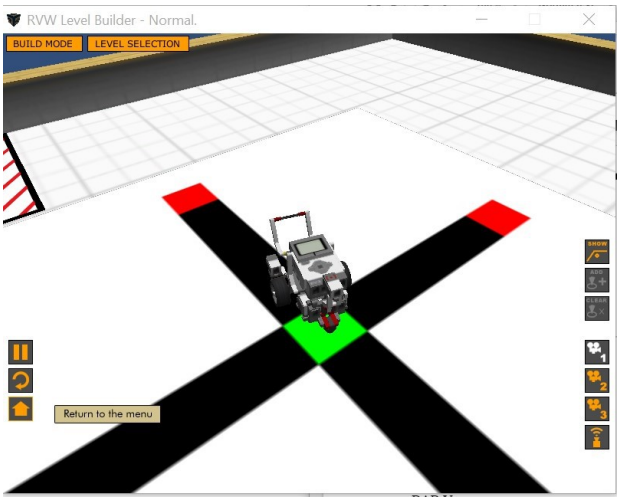


Gambar 10 Algoritma DFS: Rute Tanpa Api (Awal)

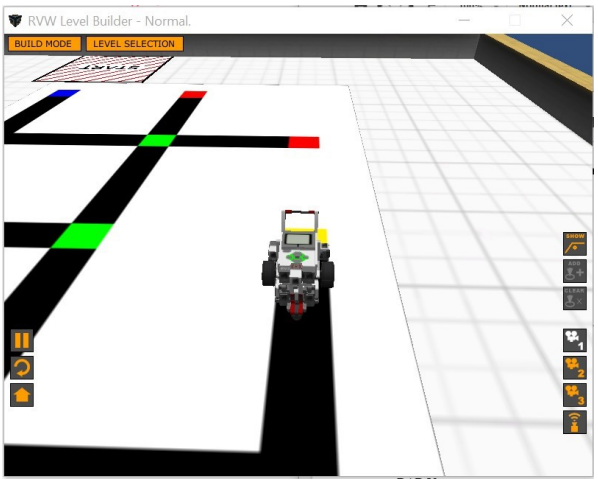


Gambar 11 Algoritma DFS: Rute Tanpa Api (Tengah)

3. Rute yang dicerminkan



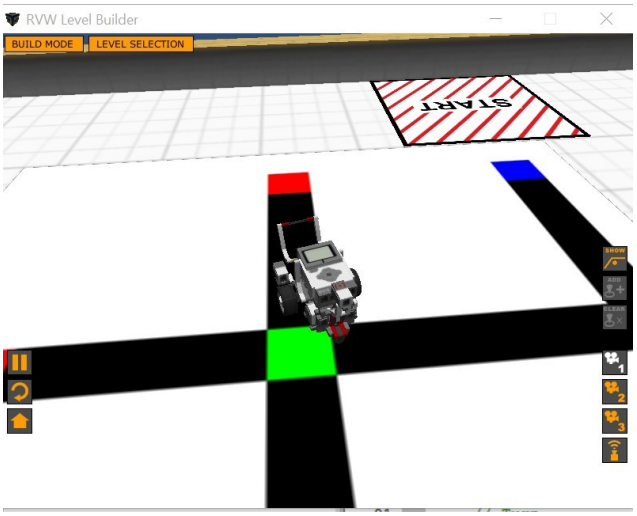
Gambar 12 Algoritma DFS: Rute yang Dicerminkan (Awal)



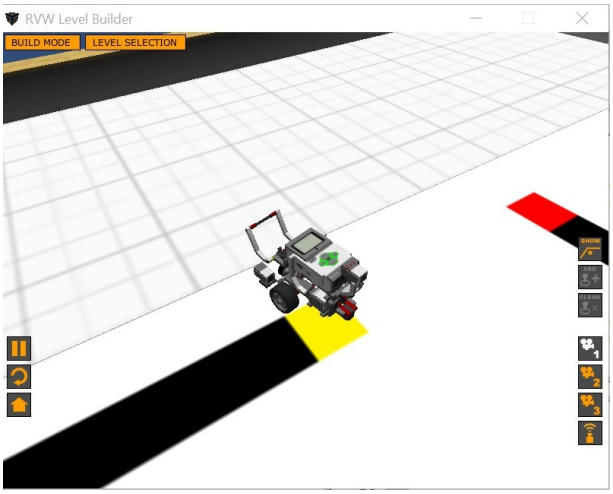
Gambar 13 Algoritma DFS: Rute yang Dicerminkan (Tengah)

4.2.2 Algoritma BFS

1. Rute normal

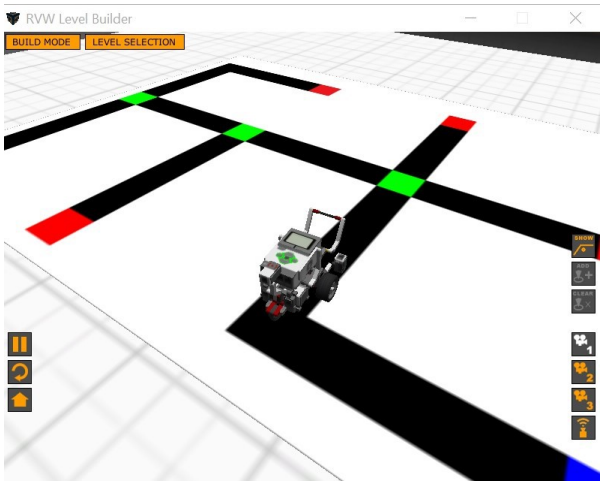


Gambar 14 Algoritma BFS: Rute Normal (Awal)

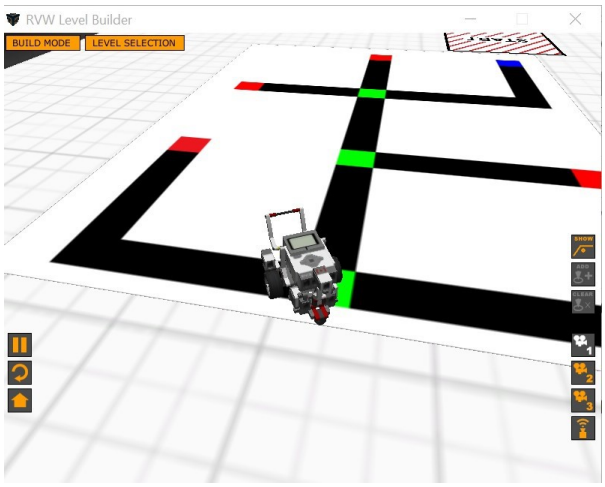


Gambar 15 Algoritma BFS: Rute Normal (Tengah)

2. Rute tanpa api

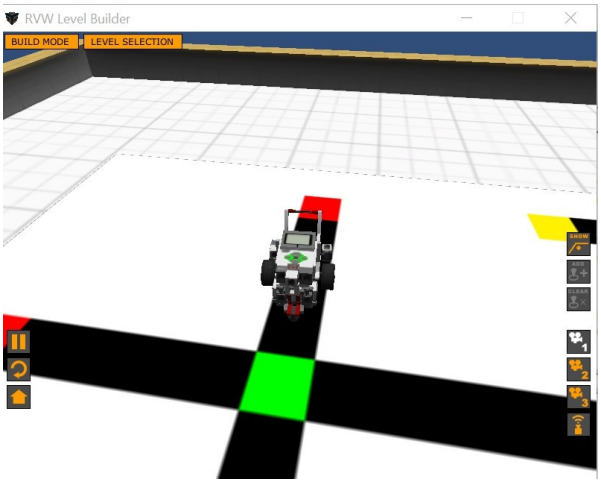


Gambar 16 Algoritma BFS: Rute Tanpa Api (Awal)

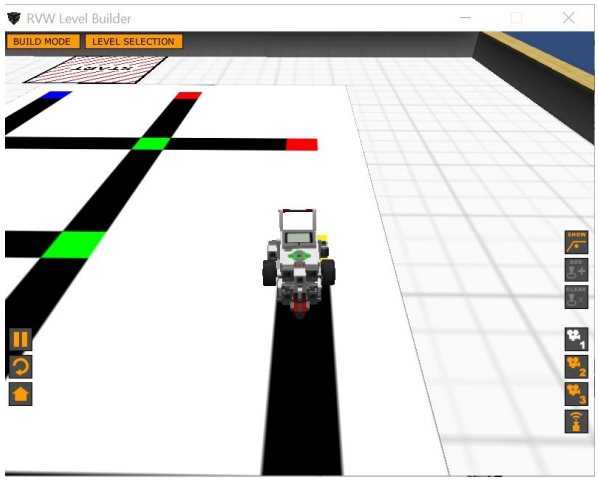


Gambar 17 Algoritma BFS: Rute Tanpa Api (Tengah)

3. Rute yang dicerminkan



Gambar 18 Algoritma BFS: Rute yang Dicerminkan (Awal)



Gambar 19 Algoritma BFS: Rute yang Dicerminkan (Tengah)

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

1. Pada permasalahan ini, DFS memakan waktu lebih sedikit dalam pencarian api dibandingkan BFS. Hal ini dikarenakan BFS perlu melakukan *backtracking* setiap kali pengecekan simpul, sehingga banyak menghabiskan waktu untuk pencarian jejak.
2. Secara umum, algoritma BFS terbukti lebih aman dan unggul, karena sistem pencariannya menyeluruh dan pasti menemukan solusi terbaik. Kekurangan BFS adalah penggunaan memorinya besar, sedangkan algoritma DFS memiliki kompleksitas memori yang lebih rendah.
3. DFS unggul dalam persoalan yang memiliki banyak solusi, karena DFS dapat menemukan solusi setelah mengeksplorasi hanya sebagian kecil dari seluruh ruang status. Kasus yang tidak dapat ditangani DFS adalah ketika kedalaman dari pohon ruang statusnya tak terbatas atau membentuk siklik, karena tidak akan pernah mencapai simpul daun agar bisa melakukan runut-balik.

5.2 Saran

1. Lebih baik tugas video yang merupakan bonus dispesifikasi lebih lanjut agar tidak terjadi miskonsepsi tentang konten video.
2. Sebaiknya pengumpulan menggunakan CD/DVD diganti dengan pengumpulan *online* melalui *uploader* untuk menghindari penggunaan barang sekali pakai.

DAFTAR PUSTAKA

[1] Spesifikasi Tugas Besar II IF2211 Strategi Algoritma: Aplikasi Algoritma DFS dan BFS: Robot Lego Mindstorms Pencari Sumber Api di dalam Labirin.
[2] R. Munir, “Bahan Kuliah ke-7”, dalam Diktat Kuliah IF2251 Strategi Algoritmik.
[3] R. Munir, “Bahan Kuliah ke-8”, dalam Diktat Kuliah IF2251 Strategi Algoritmik.
[4] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. Waktu akses: 23 Maret 2017 pukul 22.18.
[5] cs.stanford.edu/people/abisee/gs.pdf. Waktu akses: 23 Maret 2017 pukul 22.18.
[6] www.cs.tut.fi/~elomaa/teach/AI-2012-2.pdf. Waktu akses: 23 Maret 2017 pukul 22.18.
[7] http://btechsmartclass.com/DS/U3_T11.html. Waktu akses: 23 Maret 2017 pukul 22.18.