Aplikasi Graf *Breadth-First Search* Pada Solver Rubik's Cube

Felix Terahadi - 13510039
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
felix.terahadi@student.itb.ac.id, fterahadi@yahoo.com

***Abstract*— Pada makalah ini dibahas sebuah topik mengenai aplikasi graf *Breadth-First Search (BFS) pada solver rubik's cube*. Pada makalah ini dibahas bahwa permainan rubik's cube dapat diselesaikan dengan menggunakan graf, yaitu melakukan enumerasi seluruh gerakan yang mungkin sampai ditemukan solusi.**

***Kata Kunci*—graf, solver, rubik**

I. PENDAHULUAN

Rubik's cube adalah sebuah permainan yang diciptakan oleh seorang kelahiran asal Hungarian bernama Erno Rubik. Rubik belajar mengenai seni pahat pada masa studinya di universitas. Setelah Rubik lulus, dia kembali belajar mengenai arsitektur pada sebuah universitas bernama Academy of Applied Arts and Design. Rubik menciptakan Rubik's cube pada musim semi tahun 1974. Namun pada saat itu Rubik tidak bisa mengembalikan kembali rubik's cube ke posisi awal sebelum diacak. Namun dalam waktu kurang lebih 1 bulan ia memikirkan dan berhasil memecahkan *puzzle* tersebut. Pada bulan Januari 1975 Rubik mengajukan paten terhadap *puzzle* ciptaannya, yang kemudian paten tersebut disetujui pada awal tahun 1977 dan diakhir tahun yang sama muncullah rubik's cube yang pertama.

II. DASAR TEORI

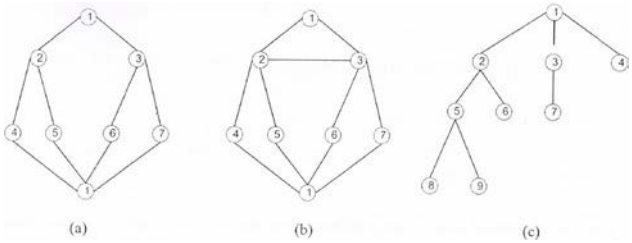
2.1 Algoritma Pencarian Melebar (BFS)

Misalkan kita mempunyai graf G yang mempunyai n buah simpul. Kita akan melakukan traversal di dalam graf, dan misalkan traversal dimulai dari simpul v. Algoritma BFS adaah sebagai berikut: Kunjungi simpul v, kemudian semua simpul yang bertetangga dengan simpul v dikunjungi terlebih dahulu. Selanjutnya, simpul yang belum diknjungi dan bertetangga dengan simpul – simpul tadi dikunjungi, demikian seterusnya. Jika graf berbentuk pohor berakar, maka semua simpul pada arah d dikunjungi lebih dahulu sebelum simpul – simpul pada aras d + 1.

Tinjau graf pada Gambar 1(a). Bila graf dikunjungi mulai dari simpul 1, maka urutan simpul yang dikunjungi adalah 1, 2, 3, 4, 5, 6, 7, 8.

Untuk graf pada Gambar 1(b) – yaitu graf (a) ditambah dengan sisi (2, 3) – urutan simpul yang dikunjungi akan tetap sama seperti di atas (1, 2, 3, 4, 5, 6, 7, 8)

Untuk pohon berakar pada gambar 1(c), urutan simpul yang dikunjung adalah 1, 2, 3, 4, 5, 6, 7, 8, 9.



Gambar 1

Di bawah ini diberikan algoritma mengunjungi setiap simpul di dalam graf dengan algoritma BFS. Asumsi dibuat sebelum proses kunjungan adalah bahwa graf yang dikunjungi adalah graf terhubung.

Algoritma BFS memerlukan sebuah antrian q untuk menyimpan simpul yang telah dikunjungi. Simpul – simpul yang telah dikunjungi suatu saat diperlukan sebagi acuan untuk mengunjungi simpul – simpul yang bertetangga dengannya. Tiap simpul ang telah dikunjungi masuk ked lam antrian hanya satu kali.

Algoritma BFS memerlukan table Boolean yang bernama dikunjungi untuk menyimpan simpul – simpul yang telah dikunjungi (ini dimaksudkan agar tidak ada simpul yang dikunjungi lebih dari satu kali). Deklarasi table tersebut di dalam kamus adalah

Deklarasi
```
Dikunjungi:array [1..n] of boolean
```

Elemen tabel
```
Dikunjungi[i]
```

Bernilai *true* jika simpul *I* di dalam graf sudah dikunjungi, sebaliknya bernilai *false* jika simpul *I* belum dikunjungi.

Pada mulanya, seluruh elemen tabel `dikunjungi` diinisialisasi dengan nilai *false*, yang berarti bahwa belum ada simpul – simpul graf yang dikunjungi,

```
For i ← 1 to n do
      Dikunjungi[i] ← false
Endfor
```

Jika simpul *i* sudah dikunjungi, maka `dikunjungi`, diubah menjadi *true*,

```
Dikunjungi[i] ← true
```

Kita mengasumsikan bahwa graf direpresentasikan dengan matriks ketetanggan $A = [a_{ij}]$ yang berukuran n x n, yang dalam hal ini

aij = 1, jika simpul *i* dan simpul *j* bertetangga,
aij = 0, jika simpul *i* dan simpul *j* tidak bertetangga.

Algoritma BFS untuk traversal graf selengkapnya adalah :

```
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 pois 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) { masukan 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
                 write(w) {cetak
                   simpul yang
                   dikunjungi}
                 MasukAntrian(q,w)
                 Dikunjungi[w] ←
                   true
            endif
       endif
endif
endwhile
{ AntrianKosong(q) }
```

2.1 Algoritma Umum Pemecahan Rubik's Cube

Berikut ini adalah algoritma yang digunakan untuk memecahkan rubik's cube. Notasi yang digunakan adalah sebagai berikut

- F : Depan (Front)
- B : Belakang
- R : Kanan (Right)
- L : Kiri (Left)
- U : Atas (Up)
- D : Bawah (Down)

Seluruh notasi diatas diartikan sebagai 1x putaran pada bagian tersebut searah jarum jam. Jika terdapat huruf i kecil seperti Fi, Ri, Li, Ui, Bi, maka berarti bagian tersebut diputar berlawanan arah jarum jam.

Ada tiga tahapan utama pada algoritma ini, yaitu

1. Penyelesaian layer 1 dengan benar dengan sisi sampingnya membentuk huruf T seperti pada gambar 2

gambar 2

Pada tahap ini, perlu ditetapkan sisi yang akan

diselesaikan terlebih dahulu. Penyelesaian dari layer 1 relatif mudah sehingga beberapa orang tidak perlu menggunakan algoritma untuk melakukan tahapan awal ini. Namun yang membuat tahapan 1 agak sulit adalah ketika ingin menyelesaikan salah satu sisinya dan mendukung untuk sisi yang berada di sekitarnya.

2. Penyelesaian layer 2 yaitu menyelesaikan seluruh layer 2 seperti pada gambar 3

gambar 3

Penyelesaian tahap 2 cukup sulit untuk dapat dipecahkan tanpa menggunakan algoritma. Untuk menyelesaikan layer 2, terdapat 2 buah algoritma yaitu untuk versi pemindahan ke kiri dan pemindahan ke kanan. Yang akan dijelaskan disini adalah versi kiri seperti pada gambar 4

gambar 4

Dengan berdasarkan pada gambar, yang perlu dilakukan adalah memindahkan bagian yang ditunjuk oleh panah berwarna putih, ke posisi yang ditunjuk oleh panah berwarna kuning. Algoritmanya adalah sebagai berikut

Ui Li U L U F Ui Fi

Begitu juga dengan versi yang kanan, yaitu menggunakan algoritma sebagai berikut

U R Ui Ri Ui Fi U F

Dengan berbekal dua buah algoritma tersebut, layer 2 dapat diselesaikan dengan mudah.

3. Penyelesaian layer 3, merupakan tahapan yang paling sulit karena akan ada variasi kasus yang muncul.

Gambar 5

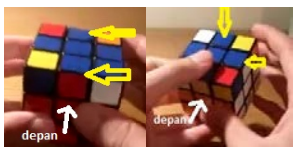
Setelah tahap 2 selesai, lihat sisi yang sama sekali belum dipecahkan. Akan terdapat 4 kasus yang mungkin ditemui seperti pada gambar 6 dibawah ini.

gambar 6

Tujuan yang harus dicapai adalah *state* yang kanan yaitu berbentuk (+). Pencapaian state tersebut tidak memperdulikan kebenaran sisi – sisi sekitarnya. Lakukan algoritma dibawah dengan *state* dan posisi – posisi seperti di atas berulang kali hingga mencapai bentuk (+)

F R U Ri Ui Fi

Setelah terbentuk (+), dilakukan cek apakah masing – masing warna dari bagian (+) tersebut cocok dengan sisi sampingnya. Namun bagaimanapun *state*nya, pasti ada setidaknya 2 bagian yang sisinya sudah benar pada posisinya. Akan ada dua kasus utama yang perlu diperhatikan yaitu seperti pada gambar 7



(A) *(B)*

Gambar 7

Algoritma yang digunakan untuk menyelesaikan tahapan ini adalah

R U Ri U R U U Ri

Jika kasus yang dihadapi adalah kasus (A) maka lakukan algoritma diatas dengan posisi tersebut sebanyak satu kali, kemudian *state* akan berubah menjadi kasus (B), lakukan algoritma yang sama dengan posisi seperti pada gambar sebanyak satu kali dan seluruh bagian pada (+) akan benar terhadap sisi – sisi lainnya. Jika sejak awal ditemui kasus (B) maka cukup lakukan algoritma

diatas sebanyak satu kali.

Langkah berikutnya adalah membenarkan letak setiap bagian sudut pada tempat yang sebenarnya namun tidak memperdulikan ketepatan posisi warna seperti pada gambar 7 dibawah ini

Gambar 7

Pada gambar 7, letak dari bagian sudut yang diberi panah sudah benar yaitu terletak diantara warna merah, putih, dan biru walaupun posisinya masih belum benar.

Lakukan algoritma berikut ini dengan meletakan posisi sudut yang benar pada kanan bawah (jika ada) secara berulang – ulang hingga seluruh sudut berada pada letak yang benar.

U R Ui Li U Ri Ui L

Langkah berikutnya yang juga adalah langkah terakhir yaitu membenarkan posisi bagian sudut tersebut. Untuk membenarkannya digunakan algoritma

Ri Di R D

Dengan ketentuan dan kondisi sebagai berikut

1. Lakukan algoritma diatas terus – menerus hingga posisi pada sudut kanan bawah menjadi benar.
2. Jika sudut tersebu sudah benar, lakukan Ui sebanyak satu kali, sehingga sudut kanan bawah terdapat bagian yang posisinya masih salah.
3. Lakukan langkah 1 dan langkah 2 berulang – ulang hingga rubik's cube terpecahkan. Saat melakukan langkah 1 dan langkah 2 dengan berulang – ulang, terlihat sepertinya seluruh rubik's cube menjadi berantakan, namun pada akhirnya akan dengan sendirinya kembali ke posisi yang benar.

III. ANALISIS DAN IMPLEMENTASI

3.1 Solusi Menggunakan Algoritma Umum

Penggunaan algoritma umum seperti pada upa-bab dasar teori mencakup seluruh *state* yang mungkin terjadi. Oleh karena itu jumlah gerakan yang dilakukan untuk menyelesaikannya akan relatif sama untuk setiap kasus *state* rubik's cube yang mungkin. Relatif sama diartikan sebagai jumlah gerakan yang diperlukan untuk mencapai solusi pada setiap kasus tidak berbeda jauh hingga berpuluh – puluh kalinya atau bahkan lebih dibandingkan alternatif solusi yang lain.

Algoritma ini memiliki kelebihan yaitu mudah dimengerti sebab memiliki pola – pola, kecenderungan, dan algoritma yang tetap. Sehingga setiap orang dapat mempelajari bagaimana cara memecahkan *puzzle* rubik's cube. Namun algoritma ini memiliki kelemahan yaitu jumlah gerakan yang harus dilakukan tidaklah optimum global. Selain itu jika algoritma umum digunakan sebagai *solver*, maka diperlukan algoritma tambahan dalam memilih warna apa yang sebaiknya dipilih untuk diselesaikan terlebih dahulu berdasarkan *state* yang ada. Pemilihan warna yang kurang tepat akan mengakibatkan keoptimuman dari *solver* berkurang.

3.2 Penerapan BFS Pada Solver

Implementasi BFS pada solver rubik's cube pertama – tama dilakukan adalah pendefinisian struktur data. Struktur data yang akan saya gunakan adalah sebagai berikut :

```
Type Sisi : array [0..2] of array
[0..2] of integer
Type Rubik : <    Kiri        : Sisi,
                  Kanan       : Sisi,
                  Atas        : Sisi,
                  Bawah       : Sisi,
                  Depan       : Sisi,
                  Belakang    : Sisi
             >

Type DaftarDikunjungi : List of Rubik

Type Simpul : < Info  : Rubik
                Parent: Simpul
                Child : List of Simpul
              >

Type Graf : < Akar : Simpul >
```

Kemudian buat fungsi – fungsi yang disesuaikan dengan algoritma umum yaitu F B L R U D dengan masukan n yaitu berapa kali sisi tersebut diputar dan mengembalikan boolean yang hasil pergerakan jika *state* belum ada di dalam daftar dikunjungi dan *NULL* jika *state* sudah ada di dalam daftar dikunjungi. Terdapat beberapa fungsi dan prosedur untuk mendukung *solver :*

1. Prosedur initRubik(input R: Rubik) yang berguna untuk menginisiasi rubik dengan *state* tertentu sehingga siap untuk dilakukan solve
2. Fungsi isContain(R: Rubik) berguna untuk mencek apakah Rubik R sudah ada didalam daftar dikunjungi atau belum. Mengembalikan nilai *true* jika R sudah terdapat di dalam daftar dikunjungi dan mengembalikan nilai *false* jika R

sudah terdapat di dalam daftar dikunjungi. isContain digunakan di dalam fungsi F, B, L, R, U, D.

3. Prosedur addVisited(R: Rubik) berguna untuk menambahkan suatu *state* R ke dalam daftar dikunjungi.
4. Prosedur addChild(input/output Simpul: CurNode, R: Rubik) adalah prosedur yang menambahkan anak pada simpul tersebut
5. Prosedur initGraph(input R: Rubik) inisialisasi graph dengan akarnya adalah Rubik R yang belum dilakukan pergerakan
6. Fungsi solusi adalah fungsi yang mengembalikan true jika sudah terpecahkan

```
function F(n: integer[1..3]) → Rubik
function B(n: integer[1..3]) → Rubik
function L(n: integer[1..3]) → Rubik
function R(n: integer[1..3]) → Rubik
function U(n: integer[1..3]) → Rubik
function D(n: integer[1..3]) → Rubik
function isContain(R: Rubik) → Rubik
procedure initRubik(input R: Rubik)
procedure addVisited(input/output L:
DaftarDikunjungi, input S: Simpul)
procedure addChild(input/output:
Simpul: CurNode, input R: Rubik)
procedure initGraph(input R: Rubik)
function solusi(R: Rubik) → boolean
```

Kemudian tentukan urutan pencobaan seluruh kemungkinan yang ada. Urutan yang saya tetapkan adalah Sisi depan, belakang, atas, bawah, kiri dan kanan.

Pada dasarnya, program akan melakukan pencobaan terhadap seluruh kemungkinan yang ada dengan keterurutan seperti yang telah ditentukan diatas. Saat mencoba, dilakukan juga pengecekan terhadap hasil cobaan tersebut dengan *state* yang terdapat di dalam daftar dikunjungi, jika belum terdapat di dalam daftar dikunjungi, maka tambahkan *state* tersebut di dalam daftar dikunjungi, tambahkan simpul tersebut sebagai anak dari *current state*. Setelah itu, dilakukan cek terhadap simpul tersebut, apakah simpul tersebut sudah merupakan solusi atau belum, jika sudah merupakan solusi, maka proses pengulangan dihentikan sebab telah ditemukan solusi yang juga merupakan solusi optimum.

Untuk detailnya dapat dilihat pada *pseudo-code* di bawah ini:

```
function solveRubik(R: Rubik) → Simpul

Deklarasi
  curState : Simpul;
  Puzzle : Rubik;
  G : Graf;
  q : antrian;
  LV : DaftarDikunjungi;

  procedure BuatAntrian(input/output q
  : antrian;
  {membuat antrian kosong, kepala (q)
  diisi 0}

  procedure MasukAntrian(input/output q
  : antrian, input R: Rubik)
  { memasukkan R ke dalam antrian q
  pada pois belakang}

  Procedure HapusAntrian(input/output q
  : antrian, output R: Rubik)
  { 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}
  initGraph() {inisialisasi graph}
  initRubik(Puzzle) {inisialisasi
  rubik}

  MasukAntrian(q,G.akar);
  {Tambahkan G.akar dalam antrian}

  HapusAntrian(q,curState); {Ambil}
  addVisited(LV,CurState);
  {Tambahkan dalam ListDikunjungi}

  While not AntrianKosong(q) do
    HapusAntrian(q, curState) {
    simpul curState telah dikunjungi,
    hapus dari antrian }

    For a ← 1 to 3 do
      Rubik CR ← F(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}

    For a ← 1 to 3 do
      Rubik CR ← B(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}

    For a ← 1 to 3 do
      Rubik CR ← U(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}
```

```
    For a ← 1 to 3 do
      Rubik CR ← D(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}

    For a ← 1 to 3 do
      Rubik CR ← L(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}

    For a ← 1 to 3 do
      Rubik CR ← R(a);
      if (CR <> NULL) then
        MasukAntrian(q,CR);
        addChild(curState,CR);
        addVisited(LV,CR);
      if (solusi(CR))
        break; {berhenti mencari}

  endwhile
```

V. KESIMPULAN

Aplikasi BFS sangat banyak, salah satu diantaranya adalah *solver* Rubik's Cube. BFS memberikan solusi yang optimum global pada setiap kasus dari *puzzle* Rubik's Cube. Namun pencarian solusi optimum tersebut sangat sulit dimengerti secara logika bagi orang – orang pada umumnya. Berbeda halnya dengan algoritma umum yang memberikan pola – pola, kecenderungan, dan algoritma yang digunakan untuk menangani kasus – kasus tersebut. Dalam pengaplikasiannya, *solver* Rubik dengan BFS bisa dikatakan boros karena seluruh state yang mungkin harus ditampung di dalam daftar list yang dikunjungi dan di dalam graf sebagai simpul. Dan dapat berpotensi terjadinya *out of memory*. Modifikasi untuk rubik's cube dengan ukuran yang lebih besar dapat dilakukan namun akan memperbesar kemungkinan terjadinya *out of memory* sebab selain jumlah kemungkinan *state*nya yang membesar, kedalaman dari grafpun cenderung bertambah dibandingkan dengan ukuran rubik's cube yang lebih kecil.

REFERENSI

[1] Ir. Rinaldi Munir, M.T., Diktat Kuliah IF3051 Strategi Algoritma, Teknik Informatika ITB, 2009.

[2] http://www.rubiks.com/world/cube_facts.php (Waktu akses : 2 Desember 2012 22:48 WIB)

[3] http://inventors.about.com/od/rstartinventions/a/Rubik_Cube.htm (Waktu akses : 2 Desember 2012 23:11 WIB)

[4] http://www.youtube.com/watch?v=HsQIoPyfQzM (Waktu akses : 15 Desember 2012 23:39 WIB)

[5] http://www.youtube.com/watch?v=IW_BBp3FPMQ (Waktu akses : 15 Desember 2012 23:39 WIB)

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, 16 Desember 2012

Felix Terahadi (13510039)