Penerapan Graf dalam validasi *path* di Permainan Saboteureun menggunakan DFS

Hendro Triokta Brianto / 13512081
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
hendro.tb@students.itb.ac.id

***Abstrak*—Saboteur adalah salah satu permainan *card game*. Permainan ini dapat diangkat menjadi permainan computer. Salah satunya bernama Saboteureun. Dalam permainan ini ada suatu permasalahan dalam prosesnya, yaitu validasi jalur. Menempatkan *path* tidak bisa sembarangan harus bertahap dengan menyambungkannya pada *path* sebelumnya. Dibuatlah suatu algoritma untuk menyelesaikan permasalahan ini. Algoritma ini mengggunakan teori graf dengan metode DFS(*Depth-first search*), yaitu dengan cara mencari sampai kedalaman dari pohon maupun graf itu sendiri.**

***Kata Kunci*—Algoritma, DFS, Graf, *path*, Saboteureun.**

I. PENDAHULUAN

Saboteur adalah salah satu permainan yang berjenis *card game* yang didesain oleh Frederic Moyersoen. Permainan ini bertujuan untuk mencari emas. Pemain dapat bekerja sama dengan pemain lainnya. Ada dua karakter dalam permainan ini, yaitu *Saboteur* dan *Gold Miner*. Permainan akan berakhir ketika emas sudah ditemukan atau semua kartu sudah habis.

Permainan Saboteur ini menjadi inspirasi para asisten praktikum IF2110 Algoritma dan Struktur Data dan menjadikan permainan ini sebagai tugas besar yang diberikan kepada mahasiswa Teknik Informatika ITB di semester ganjil tahun 2013. Namun, tidak semua fitur yang ada di permainan aslinya menjadi spek wajib tubes tersebut. Oleh karena itu, tubes tersebut diberi nama 'Saboteureun' atau akronim dari 'Saboteur *Meureun'* yang dalam bahasa Sunda berarti 'Saboteur Sepertinya'.

Berbagai permainan dalam jenis apapun baik *card game, board game* maupun jenis lainnya dapat dibuat versi komputernya. Namun, tidak selalu mirip dengan aslinya.

Penulis mengambil judul tersebut karena ada salah satu algoritma yang menurut penulis cukup menarik. Algoritma tersebut digunakan untuk mencari jalur yang menghubungkan *path* satu dengan *path* yang lainnya. Banyak ditemukan berbagai macam algoritma yang menarik dalam setiap aplikasi yang dibuat oleh pemrogram.

II. TEORI DASAR

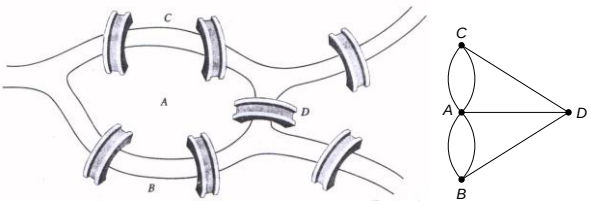
A. Graf

Graf digunakan untuk merepresentasikan objek-objek diskrt dan hubungan antara objek-objek tersebut. Graf direpresentasikan dalam bentuk *node* atau bulatan dan dihubungkan dengan garis yang disebut *edge*. Graf dimodelkan untuk menyelasaikan suatu permasalahan.

Definisi [2] Graf G (V,E) adalah koleksi atau pasangan dua himpunan, yang dalam hal ini :

a) V = himpunan tidak kosong dari simpul-simpul (*vertices*);
b) E = himpunan sisi (*edges*) yang menghubungkan sepasang simpul.

Kelahiran Teori Graf, dimulai ketika muncul suatu persoalan yang dikenal dengan masalah jembatan Königsberg (tahun 1736):



Gambar 2.1 Masalah Jembatan Königsberg.[1]

Graf yang merepresentasikan jembatan Königsberg:

- Simpul (*Vertex*) → menyatakan daratan
- Sisi (*edge*) → menyatakan jembatan

Perjalanan Euler, adalah perjalanan dari suatu simpul kembali ke simpul tersebut dengan melalui setiap ruas tepat satu kali. Perjalanan Euler akan terjadi jika graf terhubung dan banyaknya ruas yang datang pada setiap simpul adalah genap.

Selain itu ada juga **Lintasan Hamilton. Lintasan Hamilton** adalah lintasan yang melalui simpul di dalam

[1] Sumber Gambar: Slide Matematika Diskrit Graf (bagian 1)

graf tepat satu kali. Jika lintasan tersebut kembali ke titik awal dan membentuk sirkuit, lintasan tertutup tersebut disebut **Sirkuit Hamilton**.

Berikut adalah beberapa Terminologi dalam graf:

1. **Ketetanggaan (*Adjacent*),** dua buah simpul dikatakan bertetangga bila keduanya terhubung secara langsung pada sebuah sisi dalam sebuah graf.
2. **Bersisian (*Incidency*),** untuk sembarang sisi e = (v_j, v_k), sisi e dikatakan bersisian dengan simpul v_j atau dengan simpul v_k.
3. **Simpul Terpencil (*Isolated Vertex*),** simpul terpencil ialah simpul yang tidak mempunyai sisi yang bersisian dengannya.
4. **Graf Kosong (*null graph* atau *empty graph*),** graf yang himpunan sisinya merupakan himpunan kosong (N_n).
5. **Derajat (*Degree*),** derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut.
6. **Lintasan (*Path*),** lintasan yang panjangnya *n* dari simpul awal v_0 ke simpul tujuan v_n di dalam graf ialah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk v_0, e_1, v_1, e_2, v_2,... , v_{n-1}, e_n, v_n sedemikian sehingga e_1 = (v_0, v_1), e_2 = (v_1, v_2), ... , e_n = (v_{n-1}, v_n) adalah sisi-sisi dari graf *G*.
7. **Siklus (*Cycle*) atau Sirkuit (*Circuit*),** lintasan yang berawal dan berakhir pada simpul yang sama.
8. **Terhubung (*Connected*),** dua buah simpul v_1 dan simpul v_2 disebut terhubung jika terdapat lintasan dari v_1 ke v_2.
9. **Upagraf (*Subgraph*) dan Komplemen Upagraf,** misalkan $G = (V,E)$ adalah sebuah graf. $G1 = (V_1,E_1)$ adalah upagraf dari G jika $V_1 \subseteq V$ dan $E_1 \subseteq E$. **Komplemen** dari upagraf G_1 terhadap graf G adalah graf $G_2 = (V_2, E_2)$ sedemikian sehingga $E_2 = E - E_1$ dan V_2 adalah himpunan simpul yang anggota-anggota E_2 bersisian dengannya.
10. **Komponen Graf (*connected component*),** adalah jumlah maksimum upagraf terhubung dalam graf *G*.
11. **Upagraf Rentang (*Spanning Subgraph*),** upagraf $G_1 = (V_1, E_1)$ dari $G = (V, E)$ dikatakan **upagraf rentang** jika $V_1 = V$ (yaitu G_1 mengandung semua simpul dari *G*).
12. ***Cut-Set*,** *cut-set* dari graf terhubung *G* adalah himpunan sisi yang bila dibuang dari *G* menyebabkan *G* tidak terhubung. Jadi, *cut-set* selalu menghasilkan dua buah komponen.
13. **Graf Berbobot (*Weighted Graph*),** adalah graf yang setiap sisinya diberi sebuah harga (bobot).

B. DFS (Depth-First Search)

DFS adalah sebuah algoritma pencarian pada struktur data pohon atau graf. Cara kerjanya ialah mencari lebih dalam graf selama masih *possible*. DFS mengeksplorasi simpul terluar dari simpul *v* yang telah terbuka. Setelah semua tepi *v* dieksplorasi, pencarian ini akan melakukan *backtracking*[2]. Proses ini berlanjut sampai ditemukannya semua simpul yang bisa dijangkau dari akar.

Jika ada simpul yang belum ditemukan tetap, DFS akan memilih simpul tersebut sebagai sumber yang baru dan mengulangi pencarian dimulai dari sumber tersebut. Algoritma DFS ini akan mengulangi seluruh proses sampai semua *vertex* ditemukan [3].

Berikut adalah contoh algoritma DFS.[3]

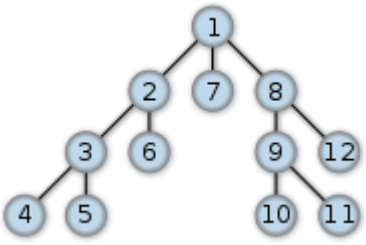
- DFS(G)

```
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-VISIT(G,u)
```

- DFS-VISIT(*G*,*u*)

```
time = time + 1 // white vertex u has just been discovered
u.d = time
u:color = GRAY
for each  v ∈ G.Adj[u] // explore edge (u,v)
        if v.color == WHITE
                v.π = u
                DFS-VISIT(G,v)
u.color = BLACK         // blacken u; it is finished
time = time + 1
u.f = time
```

Berikut adalah ilustrasi gambar dari DFS.



Gambar 2.2 *Depth-first tree*[4]

C. Saboteur

Saboteur adalah sebuah permainan berjenis *card game* yaitu permainan yang hanya memakai kartu sebagai alat bermainnya. Sehingga tidak membutuhkan papan khusus untuk bermainnya hanya memerlukan arena seluas 5x9 satuan kartu.

Permainan ini bertemakan penambangan. *Game* ini didesain oleh Frederic Moyersoen dan diperkenalkan pada tahun 2004[1]. Pada permainan ini, ada dua peran, yaitu *gold miner* dan *saboteur*. Setiap pemain mendapat salah satu dari peran tersebut. Masing-masing pemain memegang kartu sejumlah lima buah kartu berupa *path card* atau *action card*. Permainan ini dilakukan dengan

[2] *Backtracking* ialah menelusuri kembali bagian yang belum sempat ditelusuri.

[3] Algoritma diambil dari [3]

[4] Sumber gambar : http://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Depth-first-tree.svg/200px-Depth-first-tree.svg.png

metode *turn-based* (bergiliran). Pada setiap giliran, seorang pemain harus mengeluarkansatu kartu di tangan (*path card* atau *action card*) dan mengambil satu kartu dari *draw pile* jika masih tersedia.

Gold Miner mempunyai tujuan untuk membentuk sebuah *path* yang menuju ke tujuan. Tujuan yang dimaksud adalah tiga buah kartu yang masih tertutup yang disebut dengan *goal card*. Terdapat satu buah kartu emas dan dua buah kartu batu. *Gold miner* dikatakan menang jika mampu membuat jalur dari titik awal sampai kartu emas terbuka.

Saboteur memiliki tujuan yang berbanding terbalik dengan *gold miner*. *Saboteur* bertugas untuk menghalang-halangi *gold miner* untuk membangun jalan menuju kartu emas dengan berbagai macam cara. *Saboteur* dikatakan menang jika seluruh kartu di tangan pemain dan *draw pile* sudah habis dan kartu emas tidak terbuka. Setelah permainan berakhir, tim pemenang berhak mendapatkan hadiah.

Gambar 2.3 Permainan Saboteur[5]

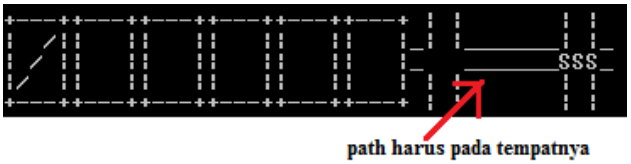
Gambar 2.4 *Goal card* emas[6]

III. VALIDASI JALUR PADA GAME SABOTEUREUN

Graf dapat juga diimplementasikan ke dalam bentuk apalikasi. Salah satunya pada permainan Saboteureun yang diadopsi dari permainan *Card Game* Saboteur. Salah satu hal yang mengharuskan kita untuk menerapkan metode ini adalah ketika kita memasang *path* pada arena permainan. Karena memasang *path* tidak bisa sembarang. Memasang *path* harus terhubung dengan *path* yang lain.

Hal ini adalah salah satu kunci sukses dalam membuat permainan dengan validasi jalur. Karena dengan validasi ini, kita dapat bermain secara bertahap sesuai dengan aslinya.

Gambar 3.1 *board field*



Gambar 3.2 memasang *path*

Oleh karena itu dibuatlah validasi untuk mencocokkan *path* dengan papan yang disediakan. Selain itu, validasi harus bisa melihat apakah jalur tersebut terhubung dengan *path* sebelumnya atau tidak. Jika ya maka *path* dapat dipasang jika tidak, *path* tidak bisa dipasang. Algoritma ini memakai metode DFS (*Depth-first search*), yaitu dengan menelusuri semua jalur apakah jalur tersebut terhubung dengan *start card*.

Langkah-langkah dalam memvalidasi jalur:

1. Memeriksa apakah *path* tersebut sudah dilalui atau belum, jika sudah proses langsung ke nomor 4;
2. Melihat jenis *path*, karena *path* ada yang satu, dua, tiga, dan empat jalur;
3. Memeriksa *Path card* yang bisa dilalui atau *path* yang buntu;
4. Memeriksa empat titik, yaitu atas, bawah, kanan, dan kiri;
5. Jika ada *path* disebelahnya, maka pemeriksaan akan kembali dari nomor 1, begitu seterusnya;
6. Jika DFS menemukan *start card*, *path* tersebut dapat dipasang, jika tidak, *path* tersebut tidak dapat dipasang.

Berikut cuplikan *source code* yang sudah diimplementasikan ke dalam bahasa C:

- **Void POB**

```
void POB(Koordinat C, Map M, Map *Lewat, boolean
*V)
/* I.S: Terdefinisi
   F.S: Mengembalikan nilai boolean V yang
berarti validasi kartu dengan kartu
disekitarnya,
   Proses: Memakai rekursif dalam pencarian
kartu disekitarnya, dan memberikan nilai true
atau false*/
```

[5] Sumber Gambar : http://1.bp.blogspot.com/_NhSli-DKA9U/TEMd86gUh0I/AAAAAAAAAKI/lT3_3biQ_-8/s1600/pic746722.jpg

[6] Sumber Gambar : http://www.annarbor.com/assets_c/2010/11/IMG_1047b-thumb-300x203-62728.jpg

```
  {
        /* Kamus Lokal */
Koordinat Cs;

/* Algoritma */
Cs.i = 3;
Cs.j = 9;
  if(!(C.i==3 && C.j==9)){
   if(IsTengah(GetKartu(C,M)))
   {
    (*Lewat).Field[C.i][C.j].Type = 0;
    if(IsAtas(GetKartu(C,M))                          &&
((*Lewat).Field[C.i-1][C.j].Type==1)){
     POB((Koordinat){C.i-1,C.j},M,Lewat,V);
    }
    if(IsBawah(GetKartu(C,M))                         &&
((*Lewat).Field[C.i+1][C.j].Type==1)){
     POB((Koordinat){C.i+1,C.j},M,Lewat,V);
    }
    if(IsKanan(GetKartu(C,M))                         &&
((*Lewat).Field[C.i][C.j+1].Type==1)){
     POB((Koordinat){C.i,C.j+1},M,Lewat,V);
    }
    if(IsKiri(GetKartu(C,M))                          &&
((*Lewat).Field[C.i][C.j-1].Type==1)){
     POB((Koordinat){C.i,C.j-1},M,Lewat,V);
    }
   }
  }
  else{
   (*V) = true;
  }
  }
```

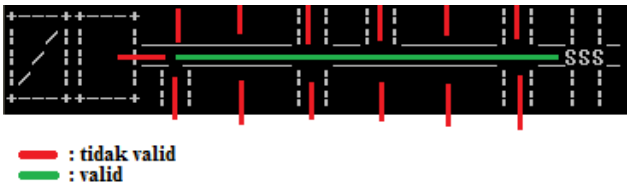
- **Boolean IsPOBValidasi**

```
boolean IsPOBValidasi(Koordinat C, Map M)
/* Mengecek apakah kartu yang disimpan di board
terhubung dengan startcard */
{
        int i,j;
  boolean V;
  Map Lewat;

  V = false;

  for(i=1;i<=5;i++){
    for(j=1;j<=9;j++){
       if(M.Field[C.i][C.j].Type!=6){
         Lewat.Field[i][j].Type = 1;
       }
       else{
         Lewat.Field[i][j].Type = 0;
       }
    }
  }
  Lewat.Field[C.i][C.j].Type = 0;
  if(IsAtas(GetKartu(C,M))  &&  (Lewat.Field[C.i-
1][C.j].Type==1)){
        POB((Koordinat){C.i-1,C.j},M,&Lewat,&V);
  }
  if(IsBawah(GetKartu(C,M))                           &&
(Lewat.Field[C.i+1][C.j].Type==1)){
        POB((Koordinat){C.i+1,C.j},M,&Lewat,&V);
  }
  if(IsKanan(GetKartu(C,M))                           &&
(Lewat.Field[C.i][C.j+1].Type==1)){
        POB((Koordinat){C.i,C.j+1},M,&Lewat,&V);
  }
  if(IsKiri(GetKartu(C,M))                            &&
(Lewat.Field[C.i][C.j-1].Type==1)){
        POB((Koordinat){C.i,C.j-1},M,&Lewat,&V);
  }

  return V;

  }
```

Pada awalnya algoritma ini menggunakan metode graf standard, yaitu dengan mengecek satu per satu tiap titik. Namun, dengan metode DFS salah satu metode yang dapat digunakan pada graf, pencarian jalur pun lebih cepat karena algoritma tersebut dapat langsung mencari jalur dari awal sampai akhir sampai ditemukan ujungnya sehingga beberapa *source code* pun dapat dihilangkan.

Selain untuk memvalidasi jalur, algoritma ini juga dapat memeriksa jalur yang sudah dilalui. Jika sudah *path* tersebut dapat dikatakan tidak valid, hal ini bertujuan agar menghemat waktu proses, walaupun tidak terlalu berpengaruh pada aplikasi sederhana seperti saboteureun.

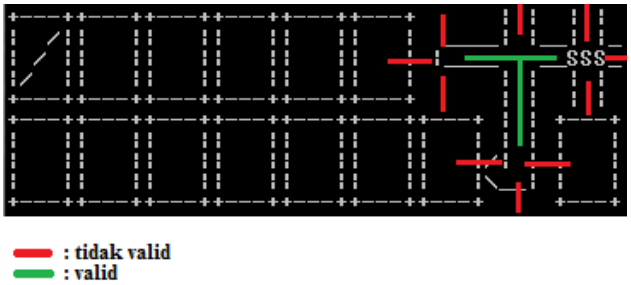
Berikut ilustrasi gambarnya:



Gambar 3.3 Validasi jalur sampai *start card*

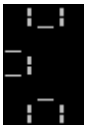
Terlihat pada gambar walaupun *path* hanya memiliki dua jalur, tetap diperiksa keempat titiknya. Hal ini karena algoritma yang dipakai bekerja seperti itu. Walaupun begitu, hal ini tidak menjadi masalah karena semua prosesnya telah teratasi. Sama halnya ketika kita memasang kartu yang mempunyai jalan buntu.

Berikut ilustrasi gambarnya :



Gambar 3.4 Validasi jalur pada jalan buntu

Dari gambar tersebut, kita melihat walaupun yang dipasang adalah kartu buntu, pemeriksaan tetap dilakukan keempat titiknya. Pada jalan buntu yang bercabang pun kita tidak dapat memascang kartu di cabang yang lain, karena jalur di tengah pada *path* tersebut tidak terhubung.



Gambar 3.5 *Path* bercabang dengan titik tengah tidak terhubung.

Path pada gambar 3.5 bisa dipakai di ketiga titiknya.

Tetapi jika ada satu titik yang sudah terhubung, kita tidak dapat memasang kartu di titik yang lainnya.

Walaupun seperti itu, hal ini lagi-lagi tidak menjadi masalah dan algoritma pun dapat berjalan sesuai keinginan.

IV. KESIMPULAN

Teori Graf adalah sebuah teori yang sudah ada sejak cukup lama. Teori ini dapat diterapkan dalam berbagai bidang. Untuk bidang Informatika sendiri, teori graf dapat digunakan dalam menyusun suatu persoalan dan menyelesaikannya. Salah satunya penerapan dalam algoritma pada pemrograman. Algoritma adalah suatu tahapan dalam menyelesaikan masalah. Oleh karena itu, teori graf dapat diterapkan pada algoritma. Tidak selalu teori murni, penerapan graf saat ini sudah sangat berkembang seperti DFS (*Depth-first Search*).

Permainan Saboteureun adalah salah satunya. Dengan penerapan ini, kita dapat memvalidasi jalur maupun *path* pada arena permainan agar sesuai dengan aslinya. Algoritma ini sangat cocok dengan permainan Saboteureun ini karena bisa memvalidasi dengan baik.

V. ACKNOWLEDGMENT

Terima kasih saya ucapkan kepada Allah SWT. karena dengan rahmatnya penulis bisa menyelesaikan tulisan ini. Terima kasih juga kepada kedua orang tua penulis yang selalu berdoa dan bersabar. Terima kasih kepada Dr. Ir. Rinaldi Munir dan Ibu Harlili yang telah membimbing dan mendukung penulis. Terima kasih juga kepada teman-teman yang senantiasa membantu, dan kepada semua orang yang telah membantu dalam menyelesaikan tulisan ini yang tidak dapat disebutkan satu per satu.

DAFTAR PUSTAKA

[1] _____, Saboteur .http://boardgamegeek.com/boardgame/9220/saboteur. Tanggal Akses: 16 Desember 2013 pukul 08.00.

[2] Munir, Rinaldi, Diktat Kuliah IF 2120, Matematika Diskrit, Edisi Keempat, Program Studi Teknik Informatika, STEI, ITB. Bandung.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press and McGraw-Hill, 2001, pp. 603.

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 Desember 2013

Hendro Triokta Brianto
13512081