Penerapan Algoritma *Depth-first search* dan *Backtracking* dalam Program Pembentuk *Maze*

Prisyafandiafif Charifa (13509081)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
prisyafandiafif.charifa@gmail.com

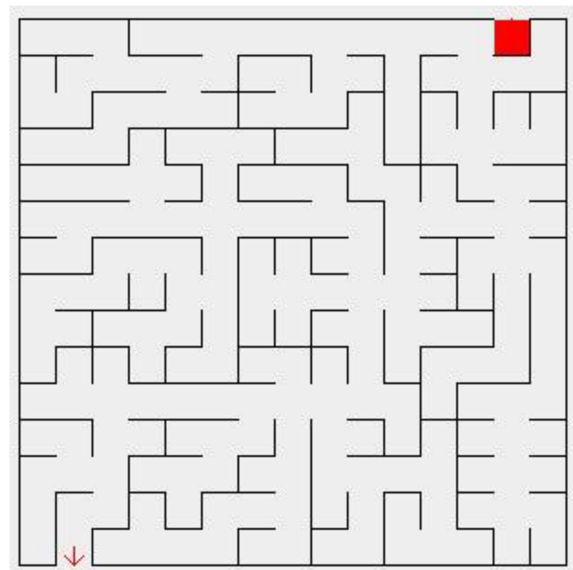
ABSTRAK

Algoritma* Depth-first search *(DFS) maupun* Backtracking *sudah sering digunakan dalam penelusuran masalah pohon ataupun pencarian solusi masalah komputasi. Algoritma DFS dan* Backtracking *juga semakin berkembang seiring berjalannya waktu. Masalah yang dapat dipecahkan tidak hanya masalah sederhana saja, namun sudah mulai mencakup masalah yang lebih rumit seperti* Artificial Intelligence *(Kecerdasan Buatan) ataupun* Search Engine *(Mesin Pencari.). Sehubungan dengan hal tersebut, penulis akan membahas tentang salah satu penerapan kedua algoritma ini pada suatu program pembentuk maze otomatis.* Maze *atau dalam bahasa Indonesia artinya 'membingungkan', adalah sejenis permainan yang mensimulasikan tempat yang penuh dengan lorong-lorong atau biasa disebut dengan labirin. Pada permainan ini, pemain berusaha untuk mencapai titik tujuan dari sebuah titik awal dengan mengandalkan kejelian dan ketelitian.* Maze *ini sendiri sebetulnya dapat berbentuk permainan dua dimensi atau tiga dimensi, namun penulis membatasi masalah hanya dalam bentuk* Maze *dua dimensi saja. Dengan algoritma* Depth-first search *(DFS) dan* Backtracking*, akan dicoba pembentukan suatu* Maze *secara otomatis dengan masukan ukuran* Maze *saja.

***Kata Kunci—DFS, backtracking*, Maze, graf.**

I. PENDAHULUAN

Dewasa ini, banyak sekali permainan yang diciptakan tidak sekadar untuk menghibur, tapi juga untuk melatih kemampuan indera para pemainnya. Permainan yang seperti ini dapat berupa permainan *video* ataupun permainan fisik. Di balik semua itu, tidak banyak yang menyadari bahwa permainan-permainan itu sebagian besar diciptakan dengan memanfaatkan algoritma-algoritma yang ada dalam ilmu komputasi, seperti algoritma *Depth-first search* (DFS) dan *Backtracking*. Salah satu permainan yang melatih kemampuan indera dan menggunakan algoritma tersebut adalah permainan *Maze* ini. Permainan *Maze* ini turut melibatkan ketajaman mata dan konsentrasi pikiran untuk menyelesaikannya, terutama jika *Maze* yang ada sudah berukuran cukup besar dan rumit. Pada permainan *Maze* ini, pemain akan dihadapkan pada suatu gambar labirin yang mempunyai titik awal sebagai titik munculnya pemain dan titik tujuan sebagai titik yang harus dicapai pemain dengan melewati jalan bercabang yang tersedia. Gambar 1 di bawah ini merupakan contoh gambar labirin dalam permainan *Maze* :



Gambar 1. Labirin ukuran kecil

Pada gambar di atas, kotak merah di bagian kanan atas merupakan titik awal dan tanda panah di bagian kiri bawah merupakan titik tujuan. Garis hitam merepresentasikan tembok yang berarti pemain tidak dapat melintasi atau menembus tembok tersebut. Pemain dituntut untuk menggerakkan kotak merah untuk melintasi jalan yang tersedia menuju titik tujuan dengan waktu yang secepat mungkin. Waktu inilah yang akan dijadikan perbandingan dalam menentukan pemenang permainan *Maze* ini.

II. METODE

Metode penulisan makalah yang digunakan adalah metode studi pustaka dan analisis kasus. Setelah penulis menjelaskan pemakain algoritma *Depth-first search*

(DFS) atau *Backtracking* pada program pembentuk *Maze*, penulis mencoba untuk menulis kode program untuk mengimplementasikan program tersebut.

III. DASAR TEORI

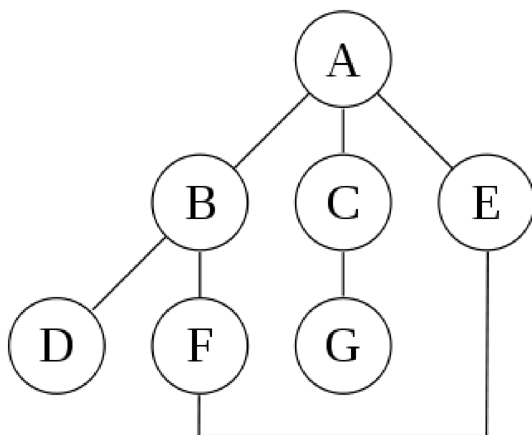
Ada beberapa hal yang perlu diketahui sebelum membahas program pembentuk *Maze* lebih lanjut. Hal tersebut adalah beberapa teori tentang pengertian dari algoritma *Depth-first search* (DFS) dan *Backtracking*.

III.1 Algoritma Depth-first search (DFS)

Depth-first search atau DFS adalah sebuah algoritma untuk melakukan penelusuran atau pencarian pada pohon, struktur pohon, atau graf. Penelusuran atau pencarian ini dimulai pada simpul akar dan menelusuri sejauh mungkin sepanjang cabang yang ada. *Depth-first search* ini pertama kali ditemukan pada abad ke-19 oleh ahli matematika dari Prancis bernama Charles Pierre Trémaux.

Secara formal, DFS adalah sebuah pencarian kurang terinformasi (*uninformed search*) yang berkembang dengan penelusuran yang dimulai dari simpul anak pertama sebuah pohon dan terus masuk hingga simpul yang diinginkan ditemukan, atau hingga mencapai simpul yang tidak mempunyai anak. Setelah itu, terjadi *backtracking* sehingga penelusurannya kembali ke simpul terdekat yang belum dikunjungi. Biasanya, pada implementasi yang tidak rekursif, semua simpul yang telah ditelusuri dengan DFS ini dimasukkan ke dalam sebuah *stack*.

Berikut adalah contoh dari sebuah graf yang akan ditelusuri secara DFS :



Gambar 2. Graf

Dengan DFS, penelusuran dimulai dari simpul A yang merupakan simpul akar. Diasumsikan bahwa simpul di tepi kiri selalu dipilih sebelum simpul di tepi kanan. Diasumsikan pula bahwa dalam penelusurannya, setiap simpul yang telah dikunjungi selalu dicatat sehingga tidak akan ada penelusuran ganda pada simpul tersebut. Dari simpul A, penelusuran akan berlanjut ke simpul B, simpul D, simpul F, simpul E, simpul C, dan terakhir simpul G. Dengan demikian penelusuran akan mengunjungi semua simpul yang ada dalam graf tepat sekali. Seandainya simpul yang pernah dikunjungi tidak dicatat, maka penelusuran akan berjalan dari simpul A ke simpul B, simpul D, simpul F, simpul E, simpul A, simpul B, simpul D, simpul F, simpul E, dan begitu seterusnya berulang-ulang tanpa pernah mengunjungi simpul C dan G.

III.2 Backtracking

Backtracking adalah algoritma penyelesaian masalah umum komputasi(*General Problem Solving*) yang bertujuan untuk mencari seluruh atau sebagian solusinya, menemukan kandidat solusinya, dan membuang sebagian kandidat solusi yang tidak memberikan hasil yang tepat. Tahap inilah yang disebut *backtrack*, yaitu tahap di mana seolah-olah penelusuran kembali bergerak mundur untuk membuang kandidat solusi yang tidak memberikan hasil tepat. Dasar dari *backtracking* ini adalah algoritma pencarian. *Backtracking* ini pertama kali diperkenalkan oleh D.H. Lehmer pada tahun 1950 dan ditulis oleh R.J. Walker pada tahun 1960.

Salah satu contoh penerapan *backtracking* adalah pada *eight queens puzzle*, yaitu permainan yang menuntut pemainnya untuk menyusun delapan buah bidak ratu di atas sebuah papan catur sehingga bidak ratu tersebut tidak bisa saling 'memakan'. *Backtracking* hanya dapat diterapkan untuk permasalahan yang memuat konsep "*partial candidate solution*". Agar *backtracking* ini dapat diterapkan pada suatu permasalahan khusus, dibutuhkan data P yang merepresentasikan permasalahannya, dan enam parameter prosedural yaitu *root, reject, accept, first, next,* dan *output* sehingga menjadi seperti berikut :

1. *root*("P") : mengembalikan kandidat parsial pada simpul akar suatu pohon
2. *reject*(P,c) : mengembalikan nilai *true* hanya jika kandidat parsial c tidak memberikan solusi tepat
3. *accept*(P,c) : mengembalikan nilai *true* jika c adalah solusi dari P, dan sebaliknya
4. *first*(P,c) : menghasilkan ekstensi pertama dari kandidat c
5. *next*(P,s) : menghasilkan ekstensi alternatif dari sebuah kandidat, setelah ekstensi s
6. *output*(P,c) : menggunakan solusi c dari P.

Dengan c adalah kandidat parsial dan s adalah ekstensi yang dihasilkan dari kandidat solusi c.

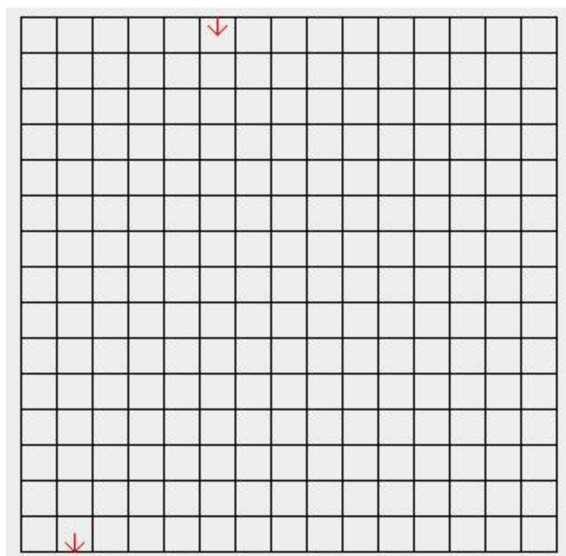
IV. PEMBAHASAN

Setelah membahas tentang algoritma *Depth-first search (DFS)* dan *Backtracking*, berikut ini akan dibahas tentang penerapan algoritma tersebut pada program pembentuk *Maze* dengan lebih lanjut. Program pembentuk *Maze* yang

membentuk labirin secara otomatis ini sebenarnya tidak hanya berbentuk dua dimensi saja, melainkan ada juga yang berbentuk tiga dimensi. Namun dalam makalah ini, penulis membatasi bagian pembahasan ini dengan program pembentuk *Maze* dua dimensi saja.

IV.1 Algoritma Pembentuk Maze

Untuk membentuk sebuah *Maze,* pertama-tama dibutuhkan algoritma pembentu *Maze*. Algoritma pembentuk *Maze* adalah algoritma yang berfungsi untuk membentuk *Maze* secara otomatis. Dalam algoritma ini, *Maze* biasanya dibentuk dari sekumpulan kotak-kotak persegi seperti gambar di bawah ini :



Gambar 3. Kotak-kotak persegi sebagai awal dari sebuah *Maze*

Algoritma pembentuk *maze* akan sedikit demi sedikit menghapus garis-garis hitam atau tembok yang membatasi antar kotak sehingga nantinya akan terbentuk jalan dari titik awal di bagian atas hingga ke titik tujuan di bagian bawah *Maze*. *Maze* ini digambarkan dalam sebuah graf terhubung dengan sisi-sisi graf sebagai representasi dari tembok-temboknya dan simpul-simpul graf sebagai representasi dari kotak-kotak perseginya. Dengan begini, dapat diketahui bahwa tujuan dari algoritma pembentuk *Maze* ini sebenarnya adalah untuk membuat subgraf yang menentukan jalur penghubung antara titik awal dengan titik tujuan.

Jika subgraf tersebut tidak terhubung, maka akan ada daerah atau *region* pada graf yang sia-sia karena daerah tersebut tidak termasuk dalam lingkup penelusuran graf. Sedangkan jika graf-nya mengandung *loop*, maka akan ada banyak jalur penghubung antar titik awal dengan titik tujuan, padahal yang dibutuhkan hanya satu. Karena itulah, pembentukan *Maze* ini sering dikatakan sebagai pembentukan pohon merentang acak atau *random spanning tree*.

Dalam penerapannya, algoritma pembentuk *Maze* menggunakan beberapa algoritma yang umum seperti *DFS* dan *backtracking*. Algoritma DFS yang digunakan dalam pembentukan *Maze* ini sebenarnya adalah sebuah versi acak dari algoritma aslinya. Sering kali diimpelementasikan dengan sebuah *stack*, cara ini adalah salah satu cara tersingkat untuk membentuk *Maze* menggunakan komputer. Bayangkan awal dari *Maze* ini adalah sekumpulan kotak-kotak persegi berukuran besar seperti pada gambar 3 sebelumnya dengan setiap kotak mempunyai empat sisi atau tembok. Dimulai dari sebuah kotak persegi yang dipilih secara acak, komputer kemudian memilih sebuah kotak persegi tetangga yang belum dikunjungi sebelumnya. Kotak persegi tetangga maksudnya adalah kotak persegi yang bersisian langsung dengan kotak persegi sebelumnya. Komputer lalu menghilangkan atau menghapus tembok yang membatasi kedua kotak persegi tersebut dan menambahkan kotak persegi baru ke dalam *stack*. Komputer melanjutkan proses ini hingga menemukan kotak persegi yang mana kotak-kotak persegi tetangganya sudah pernah dikunjungi semua. Ketika menemui kondisi ini, komputer melakukan *backtrack* sepanjang jalur yang pernah dibuatnya hingga mencapai sebuah kotak persegi dengan kotak persegi tetangganya yang belum pernah dikunjungi. Dari sini, pembentukan *Maze* dilanjutkan dengan mengunjungi kotak persegi tetangga tersebut (membentuk persimpangan jalur baru). Proses ini terus berlanjut hingga setiap kotak persegi yang ada telah dikunjungi semua, yang berarti komputer akan melakukan *backtrack* menelusuri jalur yang telah dibuat hingga sampai ke titik awal. Dengan cara ini, dapat dipastikan bahwa kotak persegi yang ada telah dikunjungi semua.

Seperti yang telah dijelaskan sebelumnya, penggunaan algoritma DFS adalah cara yang paling sederhana untuk membentuk *Maze* dan tidak menghasilkan *Maze* yang terlalu kompleks dan rumit. Untuk membuat *Maze* yang lebih kompleks, dibutuhkan perbaikan dan tambahan-tambahan pada algoritma DFS sebagai berikut :

1. Mulai dari suatu kotak persegi yang dipilih secara acak dan beri nama "*exit*"
2. Tandai kotak persegi tersebut bahwa sudah dikunjungi, lalu dapatkan daftar kotak-kotak persegi tetangganya. Untuk setiap tetangganya, berlakukan aturan berikut :

    i. Jika kotak tetangganya belum pernah dikunjungi, hapus tembok atau garis yang membatasi antara kotak persegi tersebut dengan tetangga-tetangganya, lalu lakukan langkah ini lagi secara rekursif pada tetangga-tetangganya tersebut.

Berdasarkan pada penjelasan di atas, algoritma ini melibatkan proses rekursif yang cukup dalam sehingga dapat menyebabkan *stack overflow* pada beberapa

arsitektur komputer. Algoritma ini dapat disusun ulang menjadi sebuah *loop* dengan menyimpan informasi *backtracking* dari penelusuran *Maze* itu sendiri. Hal ini juga merupakan cara cepat untuk menampilkan sebuah solusi, yang mana dimulai dari sebuah titik awal dan terus dilakukan *backtracking* hingga mencapai titik tujuan *Maze*.

Maze atau labirin yang dibentuk dengan algoritma DFS ini mempunyai kemungkinan terciptanya jalur bercabang yang rendah dan sebagai akibatnya akan tercipta banyak jalur yang cenderung lurus. Hal ini lah yang menyebabkan algoritma DFS bagus untuk digunakan dalam *video game*. Menghapus secara acak tembok atau garis yang membatasi antara kotak persegi secar acak setelah membuat *Maze* akan menjadikan jalur-jalur yang cenderung lurus itu semakin sedikit sehingga sesuai untuk situasi di mana tingkat kesulitan penyelesaian *Maze* tidak terlalu penting seperti dalam *video game*. Dalam *Maze* yang dibentuk dengan algoritma DFS ini, akan cukup susah untuk mencari jalur menuju titik tujuan dari titik awal *Maze*, namun akan cukup mudah untuk mencari jalur menuju kotak persegi yang pertama kali dipilih pada awal algoritma dari titik tujuan.

Algoritma DFS untuk pembentuk *Maze* sering kali dterapkan dengan menggunakan *backtracking*. Berikut tahap *backtracking* yang dimaksud :

1. Jadikan kotak persegi yang dipilih di awal sebagai "*current*" dan tandai bahwa kotak persegi itu sudah dikunjungi
2. Ketika ada kotak-kotak persegi yang belum dikunjungi :

   i. Jika kotak persegi "*current*" saat itu mempunyai kotak-kotak tetangga yang belum dikunjungi :

      a) Pilih secara acak satu dari kotak-kotak tetangga yang belum dikunjungi
      b) Masukkan kotak yang dipilih tersebut ke dalam *stack*
      c) Hapus garis batas atau tembok antara kotak persegi "*current*" dengan kotak yang dipilih sebelumnya
      d) Jadikan kotak yang dipilih tersebut "*current*" dan tandai bahwa kotak tersebut sudah dikunjungi.

   ii. Jika kotak-kotak tetangganya sudah dikunjungi semua :

      a) Ambil kotak yang teratas dari *stack*
      b) Jadikan kotak itu "*current*"

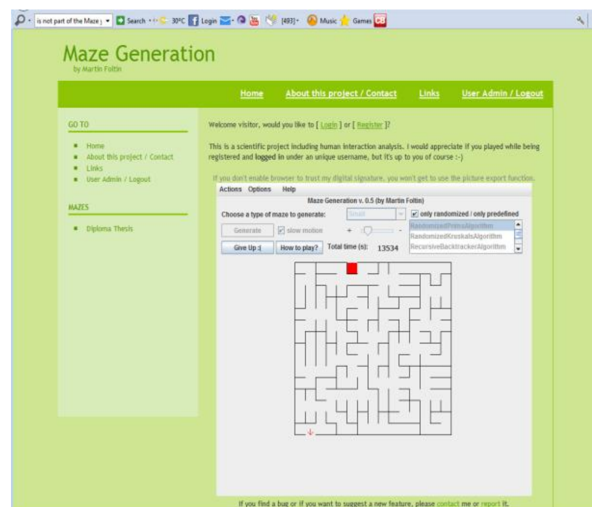
Selain algoritma DFS dan *backtracking*, terdapat beberapa algoritma lain untuk membentuk *Maze*, antara lain algoritma Kruskal acak (*Randomized Kruskal's algorithm*) dan algoritma Prim acak (*Randomized Prim's algorithm*). Algoritma-algoritma ini hanya akan dibahas secara singkat dan tidak akan dibahas secara rinci pada makalah ini karena yang difokuskan pada makalah ini hanya penerapan dengan algoritma DFS dan *backtracking* saja.

Algorima Kruskal acak adalah algoritma yang cukup menarik untuk pembentukan *Maze* karena *Maze* tidak dikembangkan dengan model seperti graf, melainkan lebih seperti 'mengukir' sekumpulan kotak-kotak persegi yang ada secara acak namun tetap menghasilkan *Maze* yang sempurna pada akhirnya. Algoritma ini membutuhkan variabel penyimpanan yang sesuai dengan ukuran *Maze* dan juga kemampuan untuk memproses setiap garis batas atau tembok di antara kotak-kotak persegi di dalam *Maze* secara acak.

Algoritma Prim acak juga hampir serupa dengan algoritma Kruskal acak yaitu membutuhkan variabel penyimpan yang sesuai dengan ukuran *Maze*. Dalam algoritma ini, setiap kotak persegi pasti termasuk dalam tiga tipe berikut :

1. "*In*" : Kotak persegi ini adalah bagian dari *Maze* dan sudah pernah ditelusuri sebelumnya
2. "*Frontier*" : Kotak persegi ini belum merupakan bagian dari *Maze* atau labirin, dan belum ditelusuri sebelumnya, namun berada di sebelah kotak persegi dengan tipe "*In*".
3. "*Out*" : Kotak persegi ini belum merupakan bagian dari *Maze*, dan kotak-kotak tetangganya tidak ada yang bertipe "*In*".

Gambar di bawah ini adalah contoh dari program pembentuk *Maze* yang berbasis *web* :



Gambar 4. *Maze Generator* berbasis *web*

IV.1 Analisis dengan Pemrograman

Setelah membahas tentang algoritma pembentuk *Maze* dengan *Depth-first search* dan *backtracking*, maka berikutnya akan dibahas tentang penerapan algoritma tersebut ke dalam bahasa pemrograman. Di sini, penulis akan menuliskan *pseudo-code* dahulu dan setelah itu sintaks pemrogramannya. Di bawah ini adalah *pseudo-code* dari algoritma pembentuk *Maze* dengan *Depth-first search* :

```
buat sebuah CellStack (LIFO) untuk menampung
 serangkaian lokasi kotak-kotak persegi
atur TotalCells = jumlah kotak-kotak persegi
 yang ada
pilih sebuah kotak persegi secara acak dan
 jadikan sebagai CurrentCell
atur VisitedCells = 1

while VisitedCells < TotalCells
    cari semua kotak tetangga dari
     CurrentCell dengan garis batas atau
     tembok yang lengkap
    if satu atau lebih kotak ditemukan
        pilih satu kotak secara acak
        hapus garis batas atau tembok antara
          kotak tersebut dengan CurrentCell
        masukkan lokasi CurrentCell ke dalam
          CellStack
        jadikan kotak yang baru tadi sebagai
          CurrentCell
        tambahkan 1 pada VisitedCells
     else
        ambil kotak persegi yang paling atas
          pada CellStack
        jadikan kotak tersebut CurrentCell
     endIf
endWhile
```

Pada *pseudo-code* di atas, *cell* berarti kotak persegi yang ada di dalam *Maze*. *Currentcell* berarti kotak persegi yang diproses saat itu. *VisitedCells* berarti kotak-kotak persegi yang sudah dikunjungi. *Pseudo-code* di atas akan diterapkan ke dalam bahasa pemrograman C++. Di bawah ini adalah kode sumber dari penerapan algoritma yang dimaksud :

```
ERR_ENUM cAlgorithmDFS::generate() {
    if (height<=1 || width<=1 || maze==0)
return FAILURE;

    size_t totalCells = (width-1)*(height-1);
    size_t visitedCells = 1;

    curCell.x = 1 + (int)(getRnd() * (width-
1));
    curCell.y = 1 + (int)(getRnd() * (height-
1));

    while (visitedCells < totalCells) {
        bool neighbours[4];
        if (curCell.x < width-1) {
            if
((*maze)[curCell.x+1][curCell.y].getWall(ALL_SI
DES)) {
                neighbours[0]=true;
            } else {
                neighbours[0]=false;
            }
        } else neighbours[0]=false;
        if (curCell.y < height-1) {
            if
((*maze)[curCell.x][curCell.y+1].getWall(ALL_SI
DES)) {
                neighbours[1]=true;
            } else {
                neighbours[1]=false;
            }
        } else neighbours[1]=false;
        if (curCell.x > 1) {
            if         ((*maze)[curCell.x-
1][curCell.y].getWall(ALL_SIDES)) {
                neighbours[2]=true;
            } else {
                neighbours[2]=false;
            }
        } else neighbours[2]=false;
        if (curCell.y > 1) {
            if ((*maze)[curCell.x][curCell.y-
1].getWall(ALL_SIDES)) {
                neighbours[3]=true;
            } else {
                neighbours[3]=false;
            }
        } else neighbours[3]=false;
        if (neighbours[0] || neighbours[1] ||
neighbours[2] || neighbours[3]) {
            int rand = (int)(getRnd() * 4);
            if (!neighbours[rand]) {
                if (neighbours[(rand + 1) %
4]) {

                    rand = (rand + 1) % 4;
                } else if (neighbours[(rand +
2) % 4]) {

                    rand = (rand + 2) % 4;
                } else if (neighbours[(rand +
3) % 4]) {

                    rand = (rand + 3) % 4;
                }
            }
            if (rand==0) {

(*maze)[curCell.x][curCell.y].crushWall(EAST);

(*maze)[curCell.x+1][curCell.y].crushWall(WEST)
;
                cellStack.push(curCell);
                curCell.x++;
            } else if (rand==1) {

(*maze)[curCell.x][curCell.y].crushWall(SOUTH);

(*maze)[curCell.x][curCell.y+1].crushWall(NORTH
);
                cellStack.push(curCell);
                curCell.y++;
            } else if (rand==2) {

(*maze)[curCell.x][curCell.y].crushWall(WEST);
                (*maze)[curCell.x-
1][curCell.y].crushWall(EAST);
                cellStack.push(curCell);
                curCell.x--;
            } else if (rand==3) {

(*maze)[curCell.x][curCell.y].crushWall(NORTH);
                (*maze)[curCell.x][curCell.y-
1].crushWall(SOUTH);
                cellStack.push(curCell);
                curCell.y--;
            }
            visitedCells++;
        } else {
            curCell = cellStack.top();
            cellStack.pop();
        }
    }
    return SUCCESS;
}
```

V. SIMPULAN

Dari berbagai pembahasan di atas mengenai penerapan algoritma DFS dan *backtracking* dalam program pembentuk *Maze*, dapat diambil beberapa simpulan sebagai berikut :

1. Permainan *Maze* tidak diciptakan sekadar untuk menghibur, namun juga untuk melatih kemampuan otak dalam ketelitian dan konsentrasi serta ketajaman indera penglihatan
2. Terdapat beberapa algoritma yang dapat digunakan untuk membentuk *Maze* antar lain DFS, *backtracking*, Kruskal, dan Prim
3. Penggunaan algoritma DFS dalam pembentukan *Maze* adalah cara yang paling sederhana, namun masih memerlukan beberapa tambahan dan perbaikan agar dapat membentuk *Maze* yang lebih rumit.

VI. DAFTAR PUSTAKA

[1] Munir, Rinaldi, “Strategi Algoritma”. Bandung, Indonesia, Januari 2009. hal. 133-138.
[2] http://www.astrolog.org/labyrnth/algrithm.htm. Tanggal Akses : 5 Desember 2011
[3] http://www.martinfoltin.sk/mazes/. Tanggal Akses : 5 Desember 2011
[4] http://www.math.com/students/puzzles/mazegen/mazegen.html. Tanggal Akses : 6 Desember 2011
[5] http://xefer.com/2007/07/maze.. Tanggal Akses : 6 Desember 2010
[6] http://en.wikipedia.org/wiki/Kruskal%27s_algorithm. Tanggal Akses : 6 Desember 2011
[7] http://en.wikipedia.org/wiki/Prim%27s_algorithm Tanggal Akses : 6 Desember 2011

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, 9 Desember 2011

Prisyafandiafif Charifa
(13509081)