# Penerapan Algoritma Breadth-First Search dalam Pencarian Jalur Terpendek dalam Permainan *Tower Defense*

I Dewa Putu Deny Krisna Amrita - 13514096
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jalan Ganesa 10 Bandung 40132*
*idpdka@yahoo.com*

***Abstraksi*—Saat ini, banyak *video game* dalam berbagai platform dan teknologi yang spektakuler diciptakan untuk memberikan hiburan kepada seluruh orang di dunia. *Video game* yang diciptakan hadir dalam berbagai *genre*, seperti petualangan, balapan, olahraga, *real-time strategy*, *tower defense*, dan lain sebagainya. Salah satu *genre* yang digemari saat ini adalah *tower defense*. Untuk menyelesaikan permainan *tower defense*, sebuah objek harus "mengalahkan" suatu objek *target*, dimana untuk mencapai *target* tersebut perlu melewati berbagai halang-rintang. Untuk mencari jalur terbaik yang dapat ditempuh menuju objek *target*, dibutuhkan algoritma *Breadth-First Search* (BFS) yang dapat digunakan untuk mencari jalur yang paling optimal dengan jarak yang terpendek. Algoritma ini dapat menyelesaikan permasalahan penentuan pergerakan objek yang bergerak untuk mencari jalur (*pathfinding*).**

***Keywords—Breadth-First Search, BFS, tower defense, jalur terpendek, pathfinding***

## I. PENDAHULUAN

Saat ini, seluruh orang di dunia sudah mulai mengenal *video game* yang bervariasi *genre*nya. *Genre video game* dalam hal ini adalah petualangan, balapan, olahraga, *real-time strategy*, *tower defense*, dan lain sebagainya. Saat ini *video game* pun sudah diimplementasikan dalam berbagai *platform*, seperti PC, *console*, dan juga dalam *platform mobile*, seperti Android dan iOS.

Dalam permainan ber-*genre tower defense*, terdapat objek-objek, yang biasa disebut "musuh", yang akan bergerak menuju sebuah suatu objek yang lain, yang dalam hal ini disebut "*tower*". Dalam permainan ini juga sudah ditentukan jalan yang akan dilewati. Jalan-jalan ini pada umumnya akan dibatasi oleh berbagai halang-rintang, seperti pohon, batu, dan objek-objek haling-rintang lainnya. Cara untuk memenangkan permainan ini adalah menggerakkan objek musuh sehingga dapat mencapai *tower* dan menghancurkan *tower* tersebut.

**Gambar 1.1 Clash Royale, salah satu game ber*genre tower defense* (sumber : koleksi pribadi)**

Untuk mencapai *target*, objek-objek musuh dapat diberikan algoritma yang digunakan untuk mencari jarak terdekat menuju *target*. Cara untuk mencari jalur ini biasa disebut sebagai "*pathfinding*". Dalam proses *pathfinding*, ada banyak sekali algoritma yang dapat digunakan. Algoritma-algoritma ini menggunakan graf untuk mencari jalur terpendek. Proses *pathfinding* terbagi menjadi 3 yaitu : *one source, one destination*; *one source, all destinations* atau *all sources, one destination*; dan *all sources, all destinations.* Dalam permainan *target defense*, kita menggunakan proses *all sources, one destination*, mengingat posisi objek musuh dapat lebih dari satu, dan hanya ada satu posisi *target*.

Dalam proses *all sources, one destination*, algoritma yang dapat digunakan adalah algoritma *Breadth First Search (BFS)*, algoritma *Djikstra*, dan algoritma *Bellman-Ford*. Pada makalah ini, kita akan membahas implementasi dari algoritma BFS untuk mencari jalur terpendek objek musuh menuju sebuah objek *target.*

## II. BREADTH FIRST SEARCH (BFS)

### A. Definisi

*Breadth-first Search*, dapat juga disebut BFS, merupakan suatu algoritma untuk pencarian pada struktur data pohon atau graf. Pencarian dimulai dari "akar" pohon atau suatu simpul (*node*) yang telah ditentukan sebelumnya. Selanjutnya, dicari simpul tetangganya, sebelum melakukan pencarian ke *level* selanjutnya. Kompleksitas algoritma dari BFS adalah :

$$O(|V| + |E|)$$

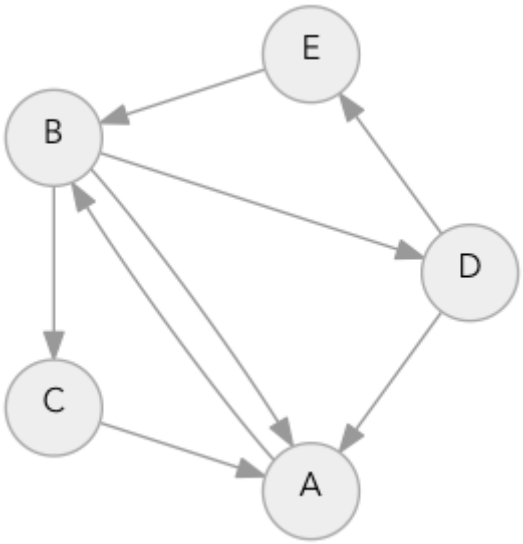
Dimana V merupakan vertex (simpul) dan E merupakan edges (sisi).

### B. Struktur Data

Struktur data yang digunakan dalam algoritma *Breadth-First Search* (BFS) adalah :

- Graf

  Graf yang digunakan adalah graf berarah. Graf berarah digunakan untuk memudahkan penentuan simpul tetangga. Berikut merupakan contoh graf yang digunakan :




**Gambar 2.1 Representasi graf berarah (sumber : Google)**

```
example_graph = SimpleGraph()
example_graph.edges = {
    'A': ['B'],
    'B': ['A', 'C', 'D'],
    'C': ['A'],
    'D': ['E', 'A'],
    'E': ['B']
}
```

**Gambar 2.2 Struktur data graf berarah (sumber : Google)**

- Lokasi

  Lokasi merupakan sebuah variable (dapat berupa integer, string, tuple, dan lain sebagainya) yang dapat menentukan lokasi pada graf.

- *Search*

  *Search* merupakan algoritma yang menggunakan graf dan lokasi (simpul) awal yang digunakan untuk menentukan suatu informasi. Informasi yang didapat dapat berupa simpul yang telah dikunjungi, simpul sebelumnya, atau ongkos jarak ke suatu simpul. Parameter dari algoritma *search* juga dapat ditambah dengan lokasi tujuan.

- *Queue*

  *Queue* merupakan struktur data yang digunakan oleh algoritma *search* untuk menentukan urutan pemrosesan lokasi selanjutnya

```
class Queue:
    def __init__(self):
        self.elements =
collections.deque()

    def empty(self):
        return len(self.elements)
== 0

    def put(self, x):
        self.elements.append(x)

    def get(self):
        return
self.elements.popleft()
```

**Gambar 2.3 Struktur data *queue*(sumber : Google)**

### C. Pseudocode

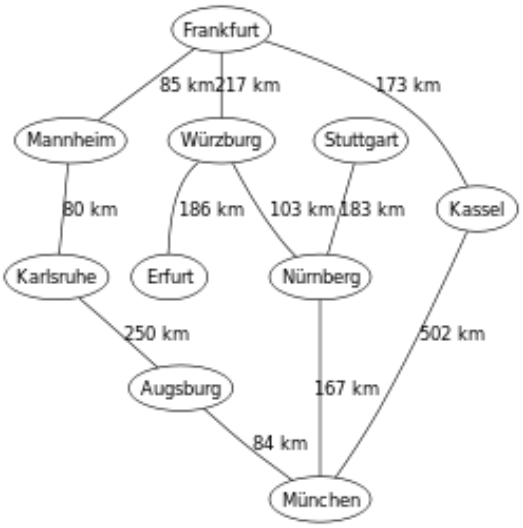
Masukan dari algoritma BFS adalah sebuah graf, dan simpul awal (akar) dari graf tersebut. Harapannya, akan didapat sebuah keluaran berupa seluruh simpul yang telah dihidupkan. *Pseudocode* algoritma BFS ini dapat dijabarkan sebagai berikut :

```
BFS(graf, akar)
  for setiap node n pada graf
    n.jarak = takhinggga;
    n.parent = NIL;
  end for

  createEmptyQueue Q;

  akar.distance = 0;
  Q.add(akar);

  while Q tidak kosong
    curr = Q.delete();
    for setiap node n yang
bertetangga dengan current
      if n.jarak = takhingga
        curr.jarak = n.jarak+1;
        n.parent = curr;
        Q.add(n);
      end if;
    end for;
  end while;
```

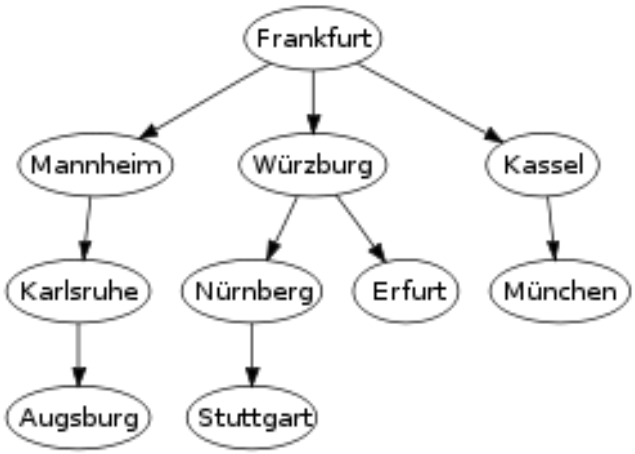
*Pseudocode* yang digunakan di atas memiliki kesamaan implementasi dengan algoritma Depth-First Search (DFS). Perbedaannya adalah BFS menggunakan struktur data *queue* (antrian), sedangkan DFS menggunakan struktur data *stack* (tumpukan). Selain itu, BFS melakukan pengecekan simpul sebelum memasukan simpul ke dalam *queue*, daripada memproses simpul tersebut di dalam *queue* sampai keluar terlebih dahulu.

### *D. Implementasi*

Berikut merupakan contoh dari penggunaan BFS di dalam pencarian kota :




**Gambar 2. Gambar masukan berupa graf kota di Jerman (sumber : wikipedia.org)**




**Gambar 2. Gambar keluaran berupa pohon BFS yang berisi simpul graf masukan yang telah dihidupkan (sumber : wikipedia.org)**

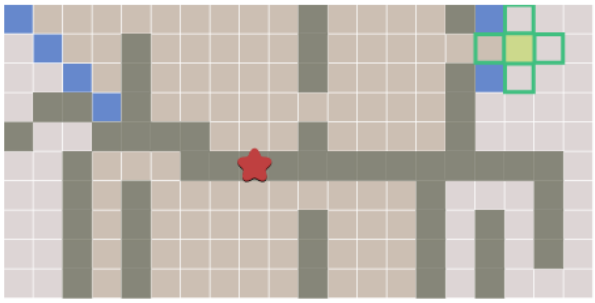
Pada kasus ini, dilakukan pencarian awal di kota Frankfurt, yang akan dianggap sebagai akar dari pohon BFS. Setelah itu, dicari kota-kota yang bertetangga dengan kota Frankfurt, yaitu kota Mannheim, kota Würzburg, dan kota Kassel. Ketiga kota tersebut dimasukkan kedalam *level* di bawah kota Frankfurt. Selanjutnya dilakukan pencarian pada kota-kota tetangga kota Mannheim, kota Würzburg, dan kota Kassel dan dimasukkan ke dalam *level* selanjutnya. Hal tersebut dilakukan sampai seluruh simpul di graf berhasil dijelajahi.

## III. PEMBAHASAN

### *A. Flood Fill dengan menggunakan BFS*

Penggunaan BFS dalam pencarian jarak terpendek di dalam permainan *tower defense* digunakan dalam melakukan proses *Flood Fill.* Untuk melakukan proses Flood Fill, graf masukan tidak direpresentasikan dengan simpul dan sisi. Graf masukan akan direpresentasikan dengan kotak-kotak persegi. Kotak-kotak tersebut dianggap sebagai simpul, dan garis yang membatasi setiap kotak dianggap sebagai simpul.

Proses *Flood Fill* dimulai dengan melakukan BFS yang dimulai dengan menentukan simpul pertama (contohnya dimulai dari kotak ujung kiri atas). Selanjutnya dilakukan proses ekspansi (pembangkitan) dari simpul awal, dilanjutkan ke tetangga selanjutnya, sampai simpul terakhir yang dapat diekspansi. Dalam proses ekspansi, terdapat suatu konsep yang disebut sebagai "*frontier*". *Frontier* merupakan sebuah batas yang membatasi daerah simpul yang sudah diekspansi dengan yang belum diekspansi.



**Gambar 3.1 Representasi graf masukan dalam proses Flood Fill (sumber : redblobgames.com)**

Pada gambar 3.1, terlihat terdapat simpul dengan berbagai macam warna. Pada gambar tersebut, simpul awal direpresentasikan dengan bentuk bintang merah, simpul yang sudah dikunjungi direpresentasikan dengan kotak warna cokelat muda, simpul frontier direpresentasikan dengan kotak berwarna biru, simpul yang belum dikunjungi direpresentasikan dengan kotak berwarna abu-abu, dan simpul halang rintang direpresentasikan dengan kotak berwarna cokelat tua.

Untuk melakukan ekspansi, dibuat sebuah *queue* untuk menyimpan *frontier*. *Queue frontier* nantinya akan berisi *list* atau *array* untuk menyimpan simpul yang ingin diekspansi. *Queue* ini diinisialisasi dengan simpul awal dan nilai *false* untuk simpul sekitarnya, untuk menandakan apakah simpul tersebut sudah dikunjungi atau belum. Selama proses ekspansi, simpul yang sudah dikunjungi akan berubah nilainya menjadi *true*.

Cara penggunaan *queue* frontier adalah dengan mengambil elemen dari *queue* tersebut, lalu dianggap sebagai simpul "current". Lalu, cek simpul tetangga dari current, yang disebut "next". Jika simpul next tersebut belum pernah dikunjungi, masukkan ke dalam *queue* frontier. Lewati jika simpul tersebut sudah dikunjungi. Berikut merupakan *pseudocode* untuk prosedur *Flood Fill* :

```
FloodFill()
 frontier = Queue;
 frontier.add(start);
 visited = array of boolean;
 visited = null;
 visited[start] = True;

 while frontier tidak kosong
   current = frontier.get();
   for n simpul tetangga current
     if n belum dikunjungi
       frontier.add(n);
       visited[n] = True;
     end if;
   end for;
 end while;
```

Untuk simpul halang-rintang tidak akan dikunjungi. Untuk simpul yang tidak dapat diraih oleh *frontier* karena terjebak dengan halang-rintang, simpul tersebut juga tidak akan dikunjungi.

Penggunaan BFS dalam proses *Flood Fill* terletak dalam penentuan simpul next. Pada gambar 3.1, simpul next (ditandai dengan kotak abu-abu berwarna hijau) didapat dari algoritma BFS dimana program mencari simpul pada level selanjutnya yang saling bertetangga (adjacent).

*B. Penentuan arah*

Setelah seluruh simpul telah melewati proses *Flood Fill*, kita dapat mencari arah yang telah dilewati dimulai dari simpul awal sampai simpul akhir yang dikunjungi. Arah tersebut berguna untuk menentukan jalur yang akan dilewati oleh objek musuh untuk menuju ke simpul awal atau simpul objek *target*.

**Gambar 3.2 Proses penentuan arah untuk mencapai simpul awal (sumber : redblobgames.com)**

Untuk menentukan arah dapat dilakukan sejalan dengan proses *Flood Fill*. *Pseudocode* untuk penentuan arah adalah sebagai berikut :

```
FloodFill()
 frontier = Queue;
 frontier.add(start);
 visited = array of boolean;
 visited = null;
 visited[start] = True;
 came_from = array of node;
 came_from = null;
 came_from[start] = null;

 while frontier tidak kosong
   current = frontier.get();
   for n simpul tetangga current
     if n belum dikunjungi
       frontier.add(n);
       visited[n] = True;
       came_from[n] = current;
     end if;
   end for;
 end while;
```

Dengan diberikan algoritma tersebut di atas, kita dapat menentukan jalur paling optimal untuk pergerakan objek musuh menuju objek *target*.

*C. Penentuan ongkos jarak*

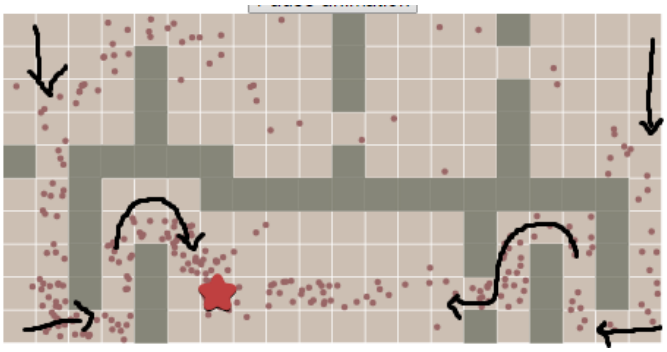
Penentuan ongkos untuk jarak suatu objek menuju simpul objek *target* dapat dicari dengan menentukan jarak dari simpul awal (simpul *target*) sampai seluruh node dikunjungi. Penentuan ongkos jarak ini juga sejalan dengan algoritma *Flood Fill* dan penentuan arah. *Pseudocode* untuk algoritma penentuan ongkos jarak adalah sebagai berikut :

```
FloodFill()
 frontier = Queue;
 frontier.add(start);
 visited = array of boolean;
 visited = null;
 visited[start] = True;
 came_from = array of node;
 came_from = null;
 came_from[start] = null;
 distance = array of integer;
 distance = null;
 distance[start] = 0;

while frontier tidak kosong
   current = frontier.get();
   for n simpul tetangga current
     if n belum dikunjungi
       frontier.add(n);
       visited[n] = True;
       came_from[n] = current;
       distance[n] =
distance[current] + 1;
     end if;
   end for;
 end while;
```

Setelah algoritma tersebut dijalankan, dapat terlihat ongkos jarak pada setiap simpul yang dapat dilhat pada gambar 3.3 sebagai berikut :

**Gambar 3.3 Penentuan ongkos jarak dari simpul awal sampai simpul terakhir dikunjungi (sumber : redblobgames.com)**

Jika representasi graf tersebut diberikan objek musuh yang bergerak, akan terlihat objek-objek musuh bergerak menuju ke arah simpul objek *target*. Ilustrasinya dapat dilihat pada gambar 3.4 di bawah :



**Gambar 3.4 Objek yang bergerak (titik-titik merah) menuju simpul objek *target (bintang merah)* (sumber : redblobgames.com)**

## IV. Kesimpulan

Algoritma BFS dapat digunakan untuk menentukan jalur suatu objek menuju suatu objek yang dikehendaki dengan menganggap bahwa objek yang dituju merupakan akar dari seluruh simpul graf masukan. Algoritma ini dapat mencakup seluruh seluruh simpul yang tidak dibatasi dengan halang-rintang.

Algoritma BFS sangat cocok digunakan untuk membuat permainan yang ber-*genre tower defense* dan *real-time strategy*, seperti Clash Royale, DotA 2, Starcraft, dan sebagainya. Hal ini disebabkan permainan-permainan tersebut merupakan game yang menggunakan proses *all sources, one destination*, dimana dapat menggunakan fitur pergerakan objek yang cukup banyak untuk bergerak ke suatu lokasi yang dikehendaki.

Untuk pergerakan menuju simpul objek *target* dengan BFS belum halus, dimana objeknya masih bergerak ke arah atas, bawah, kiri, dan kanan. Oleh karena itu, untuk mendapatkan pergerakan yang lebih halus, dapat menggunakan algoritma yang menggunakan proses *one source, one destination*, seperti A*. Namun untuk penggunaan A* harus dipikirkan lagi algoritma lain untuk mendukung penggunaan objek yang lebih dari satu.

## V. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas karena berkat dan karunia-Nya penulis dapat menyelesaikan makalah ini dengan lancer dan tepat waktu. Penulis juga berterima kasih kepada Bapak Dr. Rinaldi Munir, M.T. dan Ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc selaku dosen kuliah IF221 Strategi Algoritma yang telah mendedikasikan waktunya untuk memberikan ilmunya kepada penulis serta bimbingan yang telah diberikan. Penulis juga berterima kasih kepada orang tua dan keluarga penulis yang telah memberikan dukungan, baik moril dan materiil kepada penulis.

## Daftar Pustaka

[1] Munir, Rinaldi. 2007. *Diktat kuliah IF2251 Strategi Algoritmik*, Teknik Informatika ITB.

[2] Patel, Amit. 2014. *Pathfinding for Tower Defense*. http://www.redblobgames.com/pathfinding/tower-defense/ diakses pada tanggal 7 Mei 2016.

[3] Skiena, Steven. 2008. *The Algorithm Design Manual*. Springer. p. 480. doi:10.1007/978-1-84800-070-4_4.

[4] Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. 2009. *"Engineering route planning algorithms". Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Springer. pp. 117–139. doi:10.1007/978-3-642-02094-0_7

## 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, 8 Mei 2016

I Dewa Putu Deny Krisna Amrita
13514096