Penerapan Teori Graf untuk Pathfinding pada Permainan Elektronik

Suhendi - 13516048
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516048@std.stei.itb.ac.id

Abstract*—Pembuatan aritificial intelligence NPC pada video games memerlukan algoritma pathfinding sebagai salah satu sarana yang diperlukan untuk melengkapi artificial intelligence tersebut. Dunia permainan sebuah video game pada umumnya di rancang sedemikian rupa agar dapat direpresentasikan sebagai graf yang kemudian oleh graf tersebut dapat diberlakukan algoritma pathfinding. *Pathfinding pada video games umumnya menggunakan A* algorithm untuk menentukan jalur optimum antar dua titik yang akan ditempuh oleh sebuah karakter.

Keywords—Graf, Pathfinding, Algoritma, Dijkstra, A*, Video Games.

I. PENDAHULUAN

Video games atau permainan elektronik yang seperti namanya, merupakan permainan berbentuk elektronik yang tidak nyata (berada di dunia maya). *Video games* banyak digemari orang karena *video games* memperbolehkan pemain untuk melakukan hal-hal tidak dapat dilakukan oleh sang pemain didunia nyata. Salah satu contohnya adalah pada video games, pemain dapat mengendalikan pesawat tempur di luar angkasa dan menyerang markas alien. Hal tersebut tentu saja tidak dapat dilakukan pada dunia nyata (pernyataan ini benar ketika makalah ini ditulis).

Akses terhadap *video games* semakin mudah seiring perkembangan zaman seperti dengan *smartphone*, *computer* beserta *dedicated game console*. Akibatnya, orang dari segala rentang usia baik dari anak-anak, muda dan dewasa yang memiliki akses ke salah satu gawai tersebut telah pernah bermain *video games* baik secara sadar maupun tidak. Tentu saja mahasiswa Informatika ITB termasuk dalam rentang tersebut.

Video games merupakan salah satu hal terfavorit mahasiswa Informatika ITB. Tentu saja selain bermain video games, seorang mahasiswa Informatika ITB juga perlu mengetahui cara kerja beserta algoritma apa saja yang diaplikasikan oleh sebuah *video game*. Pada kesempatan ini, penulis akan membahas secara umum apa itu *Pathfinding* dan algoritma umum *pathfinding* pada *video games*.

II. GRAF

A. Definisi

Graf didefinisikan sebagai G = (V, E), dimana V adalah kumpulan simpul (nodes/vertices) dan E adalah kumpulan sisi (edges). Sisi atau edge adalah suatu garis yang menghubungkan 2 nodes. Dalam kata lain, graf adalah kumpulan simpul yang terhubung oleh sekumpulan sisi.

B. Terminologi Umum

1. Bersisian

   Suatu simpul dan sisi disebut bersisian bila sisi tersebu menghubungkan simpul tersebut dengan suatu simpul lainnya.

2. Bertetanggaan

   2 simpul disebut bertetanggaan bila ada satu atau lebih sisi yang menghubungkan kedua simpul tersebut.

3. Derajat

   Derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut.

4. Lintasan

   Lintasan adalah himpunan sisi E = $\{e_1, e_2, e_3, ...\}$ sehingga untuk himpunan simpul V = $\{v_1, v_2, v_3, ...\}$, e_i menghubungkan v_1 dan v_2, e_2 menghubungkan v_2 dan v_3 dan seterusnya.

5. Sirkuit

   Sirkuit adalah lintasan yang dimana sisi e_1 dan e_n bersisian dengan simpul v_1.

6. Graf Berbobot

   Graf berbobot adalah graf yang sisinya memiliki suatu nilai yang disebut bobot dari sisi tersebut.

7. Graf Berarah

   Graf berarah adalah graf yang sisinya memiliki arah.

C. Pengaplikasian Graf

Hal-hal yang di pelajari dalam teori graf banyak digunakan untuk pemecahan di dunia nyata. Contoh yang paling sering digunakan adalah *Travelling Salesman Problem*. *Travelling Salesman Problem* mendeskripsikan sebuah permasalahan dimana ada seorang pembisnis yang ingin menelusuri seluruh

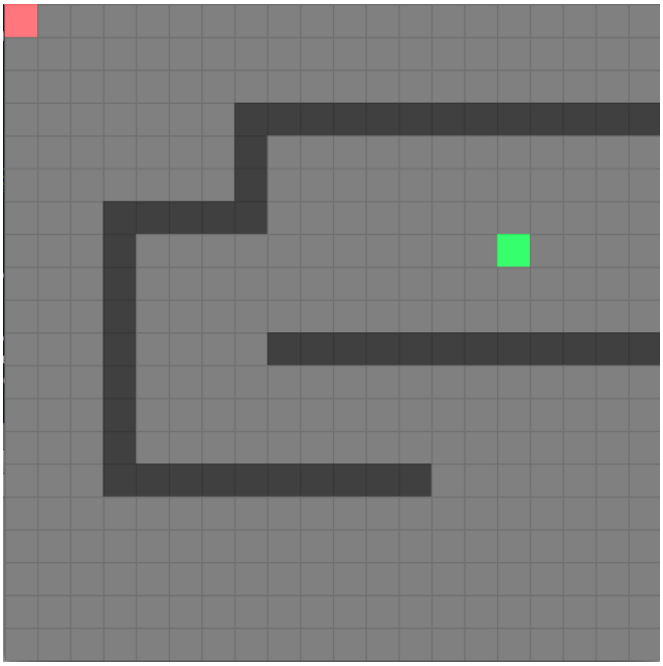
kota yang ada dan kembali ke kota semula dengan waktu yang sesingkat mungkin tetapi untuk berpindah dari satu kota ke kota lainnya membutuhkan waktu yang berbeda-beda untuk setiap pasangan kota. Permasalahan tersebut dibahas dalam teori graf sebagai *Hamilton Cycle.*

Pengaplikasian graf lainnya adalah pencarian jalur terpendek antar 2 titik pada peta, peta dapat dimodelkan sebagai graf dimana persimpangan dimodelkan sebagai simpul dan jalan sebagai sisi dengan bobot sisi setara dengan panjang jalan. Untuk mencari lintasan terpendek, dapat digunakan *Dijkstra's Algorithm* ataupun A* Algorithm.

III. Video Games World

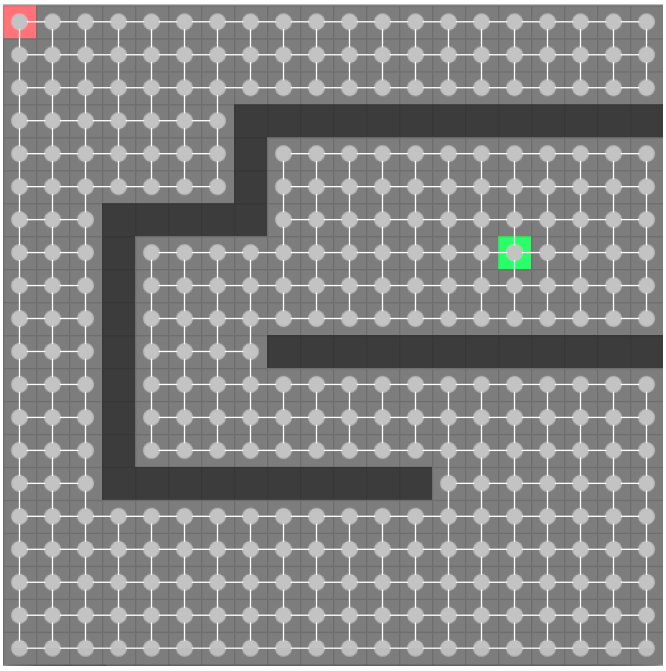
Video games pada umumnya terbagi atas beberapa *genre* seperti *Action, Adventure, Role Playing Games* (RPG) dan *Sports*. *Video games* juga terbagi menjadi 2 jenis yaitu *single-player game* dan *multi-player game*. Jenis-jenis *video games* tersebut mendeskripsikan jenis konten yang terdapat pada game itu sendiri beserta hal-hal apa yang harus di harapkan akan ditemui oleh sang pemain ketika dimainkan.

Pada umumnya, implementasi dunia game dipilih berdasarkan jenis game tersebut. Contohnya untuk permainan RTS 2D, biasanya dunia game diimplementasikan sebagai matriks dari petak, dimana setiap petak menampung informasi yang dibutuhkan oleh permainan. Matriks tersebut secara tidak langsung merupakan graf yang tidak berbobot dan tidak berarah dimana setiap petak merupakan simpul yang dihubungkan dengan 4 petak bersebelahan (atas, kiri, bawah dan kanan).



Gambar 1a. Contoh dunia pada 2D tile-based.

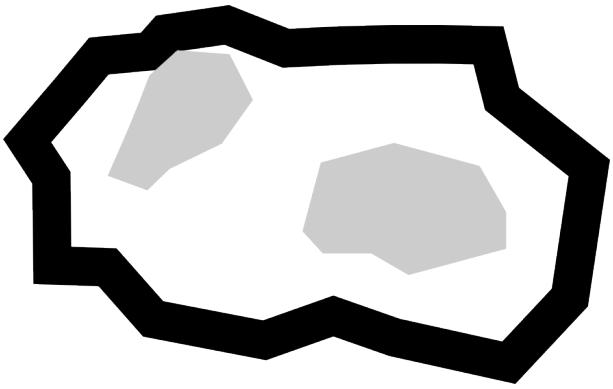
Pada contoh diatas, petak abu-abu terang merupakan petak kosong yang dapat dilalui. Sedangkan petak abu-abu gelap merupakan rintangan/petak yang tidak dapat dilalui. Representasi diatas sebenarnya secara implisit merupakan graf. Dapat dilihat dari gambar dibawah representasi dunia 2D tile-based tersebut sebagai graf.



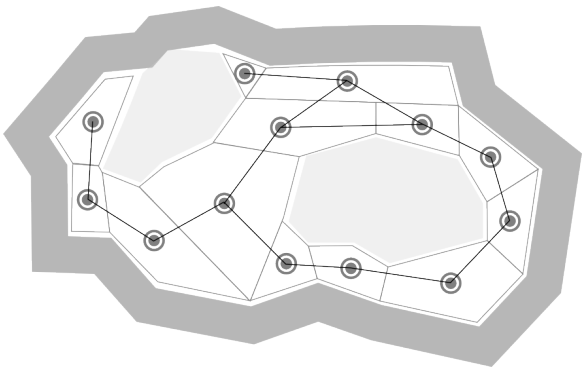
Gambar 1b. Representasi graf dari dunia 2D tile-based.

Untuk game open-world 3D, tentu saja dunia tidak diimplementasikan sebagai matriks dari petak melainkan sebagai ruang vektor 3D.

Game open-world 3D yang luas dimana pergerakkan dari karakter game tidak dibatasi oleh petak-petak melainkan melalui *collision map* yang memetakan batasan-batasan pergerakan karakter. Karena ruang vektor 3D merupakan floating-point, representasi graf dari dunia tersebut tidak dilakukan dengan cara menentukan titik-titik tertentu pada ruang vektor sebagai suatu simpul karena pendekatan tersebut menggunakan memory yang relatif besar. Pada umumnya, pada dunia 3D tersebut akan ditentukan *navigation mesh* yang menyatakan bagian dari dunia permainan yang dapat ditelusuri oleh karakter permainan. Kesatuan *navigation mesh* tersebut merepresentasikan simpul dan setiap *navigation mesh* terhubung dengan *navigation mesh* yang bersebelahan.



Gambar 2a. Contoh dunia permainan.



Gambar 2b. *Navigation Mesh* yang terbentuk.
Sumber: http://jceipek.com/Olin-Coding-Tutorials/pathing.html

IV. PATHFINDING

Suatu permainan tentu saja tidak akan seru bila tidak memiliki rintangan. Salah satu jenis rintangan yang sering ditemukan adalah NPC (*Non-player character*) sebagai musuh dari sang pemain seperti monster ataupun alien yang ingin menjajah planet asal karakter pemain. Salah satu aksi yang perlu dilakukan NPC tersebut adalah menjelajahi dunia permainan yang tentu saja bertujuan untuk mengalahkan sang pemain. Untuk menjelajahi dunia, sebuah NPC tentu saja perlu mengetahui cara menuju suatu tujuan dari posisi asal NPC tersebut. Hal ini dinamakan *pathfinding* atau pencarian jalur.

Pathfinding merupakan proses pencarian jalur dari suatu titik ke titik lainnya. Hal ini diperlukan untuk membuat AI (*Aritificial Intelligence)* yang mampu berinteraksi dengan dunia permainan dan mengubah alur dari permainan untuk memberi tantangan kepada pemain maupun meningkatkan tingkat kesulitan.

V. DIJKSTRA'S ALGORITHM FOR PATHFINDING

Dijkstra's Algorithm atau Algoritma Dijkstra adalah algoritma yang dikemukakan oleh Edsger W. Dijkstra untuk mencari lintasan terpendek antar dua simpul pada suatu graf. Algoritma Dijkstra merupakan algoritma best-first search yang menelusuri setiap simpul dan mencatat jarak terpendek untuk mencapai simpul tersebut. Ketika simpul tujuan telah ditelusuri, maka telah ditemukan linstasan terpendek dari simpul asal ke simpul tujuan.

Tahapan Algoritma Dijkstra sebagai berikut: [1]

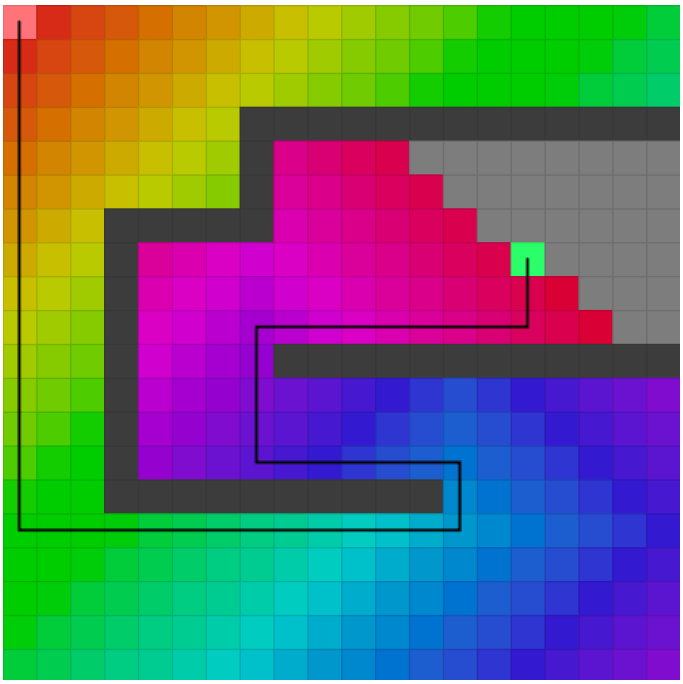
1. Tetapkan jarak sementara untuk simpul awal sebagai 0 dan tak hingga untuk setiap simpul lainnya.
2. Tetapkan simpul awal sebagai simpul sekarang dan tandai seluruh simpul lainnya sebagai tidak pernah dikunjungi.
3. Untuk setiap tetangga dari simpul sekarang, bila jarak antar simpul sekarang dan tetangga ditambahkan dengan jarak sementara simpul sekarang lebih kecil dari nilai jarak sementara yang telah ditetapkan di simpul tetangga, tetapkan jarak antar simpul ditambah dengan jarak sementara simpul sekarang sebagai nilai jarak sementara simpul tetangga tersebut.
4. Setelah setiap tetangga telah ditetapkan nilainya, tandai simpul sekarang sebagai telah dikunjungi. Simpul yang telah dikunjungi tidak akan dikunjungi lagi.
5. Bila simpul tujuan telah ditandai sebagai telah dikunjungi, maka telah ditemukan lintasan terpendek dan algoritma dihentikan.
6. Bila tidak, pilih simpul yang belum di kunjungi dengan nilai jarak sementara terendah. Bila terdapat lebih dari satu, pilih salah satu secara acak.

Berikut adalah contoh implementasi Algoritma Dijkstra dalam bahasa pemogramman Python 3:

```
import math
def Dijkstra(Graph, source, dest):
    # Unvisited set.
    Q = set()
    # Initialization.
    dist = {}
    # The previous node of a node to
traverse.
    prev = {}
    for v in Graph:
        # Infinite distance to all node.
        dist[v] = math.inf
        # Previous node of all nodes is
undefined.
        prev[v] = None
        # Add all nodes to unvisited.
        Q.add(v)
    # Distance from source to source is 0.
    dist[source] = 0
    while Q:
        # Node with the least distance.
        u = min(Q, key=lambda x: dist[x])
        # Found destination node.
        if u == dest:
            break
        # Remove the selected node.
        Q.remove(u)

        for v in u.neighbors:
            alt = dist[u] + length(u, v)
            # If a shorter path was found.
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
    # Recreate the path.
    u = dest
    path = [u]
    while prev[u]:
        u = prev[u]
        path.insert(0, u)
    return path
```

Contoh visuasilasi dari Algoritma Dijkstra pada tile-based map pada *Video Games*:



Gambar 3. Visualisasi algoritma Dijkstra.

Petak berwarna merah muda merupakan posisi awal sedangkan petak berwarna hijau muda merupakan posisi tujuan. Petak berwarna abu-abu gelap merupakan penghalang sedangkan petak berwarna abu-abu terang merupakan petak kosong. Petak yang berwarna warni, yang berubah warna berdasarkan jarak dari titik asal, melambangkan petak yang telah ditelusuri dengan algoritma Dijkstra dan garis hitam menandakan lintasan terpendek yang ditemukan.

Dari contoh dapat dilihat bahwa hampir seluruh petak ditelusuri hingga bagian bawah peta walaupun jelas bahwa petak tujuan berada di bagian atas. Hal ini dikarenakan dengan algoritma Dijkstra, program tidak mengetahui arah yang seharusnya dituju untuk mencapai tujuan. Program hanya menelusuri seluruh petak dengan jarak minimum hingga petak tujuan ditemukan. Untuk memperbaiki hal tersebut, kita dapat mengimplementasikan *Heuristic* yang digunakan untuk memberi tahu program arah yang harus dituju. Perkembangan dari *Dijkstra's Algorithm* tersebut lebih dikenal sebagai *A* Algorithm*.

VI. A* Algorithm for Pathfinding

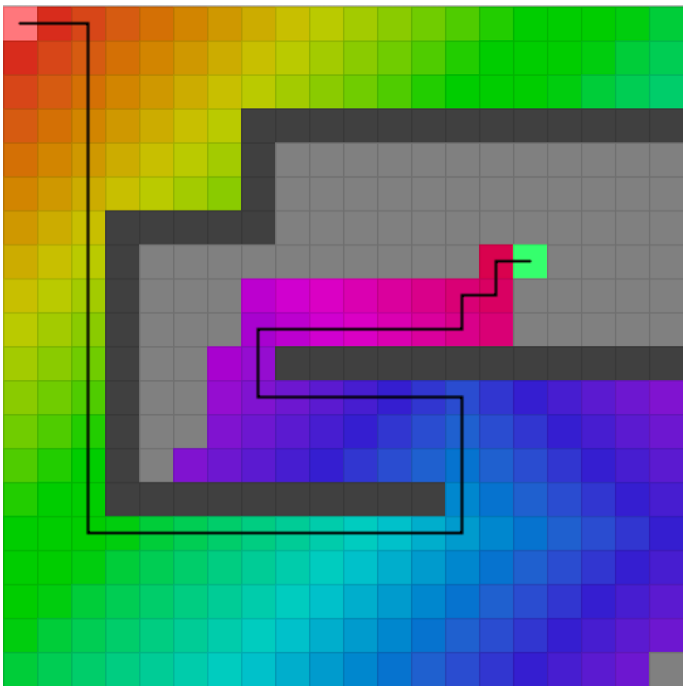
A Algorithm* adalah perkembangan dari *Dijkstra's Algorithm* yang mengimplementasikan *Heuristic* yang membantu program untuk mengetahui petak selanjutnya yang "kira-kira lebih dekat" terhadap petak tujuan. *Heuristic* adalah strategi pendapatan informasi yang dapat memberi gambaran akan solusi yang dituju [2].

Berikut adalah contoh implementasi *A** Algorithm dalam bahasa pemogramman Python 3:

```
import math
def A_star(Graph, source, dest):
    # Set of discovered node.
    # Initially contains the source node.
    openSet = {source}
    # The cost of getting from the source to a
node.
    gScore = {}
    # Heuristic cost of a getting from the source
to a node,
    # Partly known, partly heuristic.
    fScore = {}
    # The previous node of a node to traverse.
    prev = {}
    for v in Graph:
        # Infinite distance to all node.
        gScore[v] = math.inf
        # Infinite score to all node.
        fScore[v] = math.inf
        # Previous node of all nodes is
undefined.
        prev[v] = None
    # The cost of getting from the source to
source is 0.
    gScore[source] = 0
    # For the source node, the value is
completely heuristic.
    fScore[source] = estimate_cost(source, dest)
    # While there is discovered node that has not
been evaluated
.    while openSet:
        # Node with the smallest heuristic score.
        u = min(openSet, key=lambda x: fScore[x])
        # Found destination node.
        if u == dest:
            break
        # Remove the selected node from the
openSet.
        openSet.remove(u)
        for v in u.neighbors:
            tentative_gScore = gScore[u] +
length(u, v)
            # A shorter path was found.
            if tentative_gScore < gScore[v]:
                prev[v] = u
                gScore[v] = tentative_gScore
                fScore[v] = gScore[v] +
estimate_cost(v, dest)
                if v not in openSet:
                    # Discover a node.
                    openSet.add(v)
    # Recreate the path if a path was found.
    u = dest
    path = [u]
    while prev[u]:
        u = prev[u]
        path.insert(0, u)
    return path
```

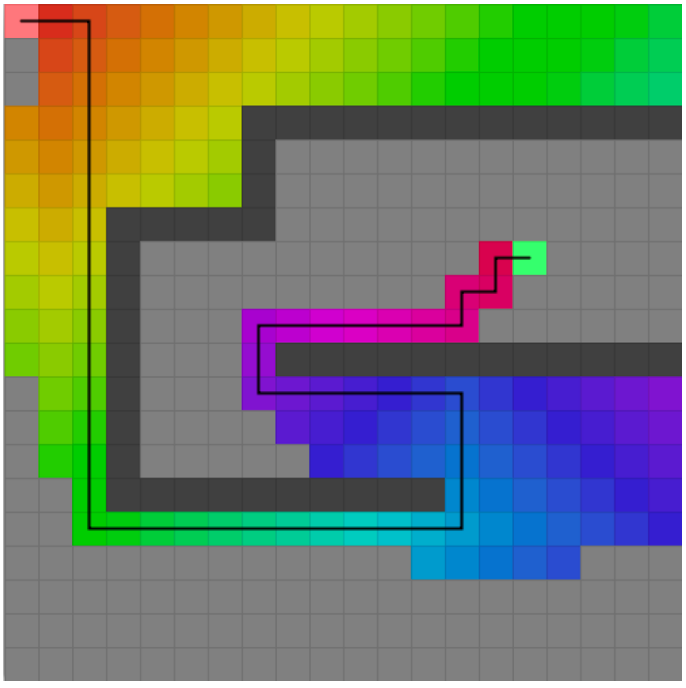
Pada contoh implementasi tersebut, fungsi *estimate_cost* merupakan fungsi *heuristic* yang berupa perkiraan jarak dari suatu simpul ke simpul tujuan.

Pada contoh implementasi algoritma Dijkstra pada tile-based 2D game diatas, kita dapat mengimplementasikan fungsi *heuristic* sebagai jarak antar dua titik. Misalkan untuk gambar dibawah, fungsi *heuristik* = jarak antar kedua titik.



Gambar 4a. Visualisasi *A* algorithm*.

Contoh diatas menggunakan jarak aktual antar kedua titik sebagai fungsi *heuristic*. Dapat dilihat bahwa hanya terjadi sedikit perkembangan dari algoritma Dijkstra di atas. Hal ini karena pemilihan fungsi *heuristic* yang linier. Apabila fungsi *heuristic* diubah menjadi kuadrat dari jarak antar kedua titik, maka program akan berjalan seperti dibawah ini.

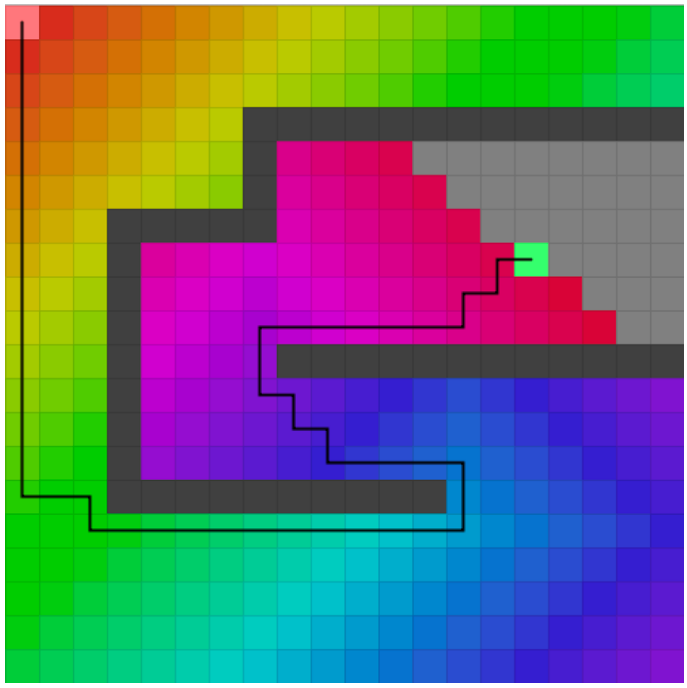


Gambar 4b. *A* algorithm* dengan fungsi *heuristic* kuadratik.

Dengan pemilihan fungsi *heuristic* tersebut, program menelusuri jauh lebih sedikit simpul dibandingkan sebelumnya. Dari contoh ini jelas terlihat bahwa pemilihan fungsi *heuristic* sangat berpengaruh akan jalannya *A* algorithm* tersebut.

Pemilihan fungsi *heuristic* kuadrat dari jarak kedua titik lebih efektik pada kasus ini dikarenakan menurut estimasi, bobot untuk menelusuri petak yang lebih jauh ke tujuan jauh lebih besar dibandingkan petak yang lebih dekat ke tujuan. Hal ini menyebabkan kecenderungan untuk memilih petak yang langsung dianggap jauh lebih baik. Bila dibandingkan dengan fungsi *heuristic* yang sama dengan jarak antar kedua petak, program akan memilih petak sebelumnya yang walaupun lebih jauh ke tujuan, tetap dianggap berbobot sama dengan petak yang lebih dekat.

Satu hal yang menarik adalah bila fungsi *heuristic* kita tetapkan sebagai suatu konstanta, *A* Algorithm* akan bertindak layaknya seperi *Dijkstra's Algorithm*.



Gambar 4c. *A* Algorithm* yang bertindak seperti algoritma Dijkstra.

Dari contoh diatas jika di bandingan dengan Gambar 3, terlihat bahwa hasil penelusuran oleh *A* Algorithm* sama dengan hasil penulusuran oleh algoritma Dijkstra. Perbedaan dalam pemilihan jalur akhir hanya dikarenakan cara implementasi yang berbeda. Dari contoh ini, dapat disimpulkan bahwa algoritma Dijkstra merupakan subset dari *A* Algorithm*.

VII. KESIMPULAN

Dunia game yang sangat beragam dirancang sedemikian rupa agar dapat direpresentasikan sebagai graf. Hal ini karena pada umumnya sistem navigasi menggunakan algoritma *pathfinding* yang memerlukan graf sebagai representasi peta. Algoritma *pathfinding* yang paling umum digunakan adalah *A* Algorithm* yang merupakan perkembangan dari *algoritma Dijkstra*.

REFERENCES

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.

[2] Pearl, Judea (1983). Heuristics: Intelligent Search Strategies for Computer Problem Solving. New York, Addison-Wesley, p. vii. ISBN 978-0-201-05594-8

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, 3 Desember 2017

Suhendi - 13516048