# Implementasi Algoritma *Path Planning* A-Star untuk *Video-Game*

Emilia Andari Razak - 13515056
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jln. Ganesha 10, Bandung 40132, Indonesia*
13515056@std.stei.itb.ac.id

***Abstrak*—Makalah ini membahas tentang bagaimana implementasi pencarian jarak terpendek untuk penentuan rute suatu entitas dalam *video game*. Ada kalanya suatu entitas perlu berpindah tempat dari *start state* ke *goal state* dengan jarak terpendek. Seiring perkembangan *video-game* yang semakin kompleks, maka diperlukan juga algoritma pencarian rute yang lebih kompleks supaya video-game semakin menantang. Pencarian rute terpendek pada makalah ini yang dibahas adalah pencarian rute menggunakan algoritma A-Star. Selain A-Star, algoritma yang dapat digunakan untuk pencarian jarak terpendek adalah Dijkstra dan BFS/DFS.**

***Kata Kunci—A-Star, path planning, penentuan rute, video-game***

## I. PENDAHULUAN

Dewasa ini perkembangan *video-game* semakin meningkat, sehingga tercipta *game-game* yang lebih kompleks dibandingkan sebelum tahun 2000. Banyak video-game yang memerlukan entitas-entitas di dalamnya untuk berpindah dari suatu *state* ke *state* lainnya, sehingga sebagian besar video game pasti menggunakan algoritma pencarian rute. Algoritma pencarian yang diterapkan bisa berbagai macam. Mulai dari alogritma *greedy* seperti Dijkstra atau *Greedy-Best-First-Search*, atau algoritma BFS/DFS. Karena semakin berkembangnya *video-game* yang semakin kompleks maka diperlukan juga algoritma yang lebih kompleks untuk pencarian rutenya (demi solusi terbaik), supaya game tersebut semakin menantang. Banyak algoritma pencarian rute yang dalam beberapa kasus kurang optimal. Bisa optimal untuk kasus tertentu, akan tetapi untuk kasus lainnya memberikan hasil yang kurang optimal. Atau lebih buruknya lagi, untuk algoritma yang tidak komplit, entitas yang dijalankan terjebak dalam local-minima sehingga tidak dapat bergerak. Hal seperti ini masih bisa diatasi pada *video-game* yang simple, tetapi dapat menyebabkan bug pada *video-game* skala besar. Hal-hal seperti ini harus dihindari.

Di sinilah algoritma A* berperan. Sebagai salah satu algoritma pencarian rute, algoritma A* kerap digunakan di berbagai video game. Akan tetapi, algoritma A* tidak selalu dapat digunakan. Oleh karena itu, walaupun algoritma A* cukup optimal (dan complete), akan tetapi tidak selalu mudah untuk diterapkan. Hal ini karena algoritma A* membutuhkan diketahui cost untuk setiap sisi dan adanya heuristik. Apabila tidak dibutuhkan/tidak diketahui costnya, algoritma yang dapat diterapkan adalah BFS/DFS. Apabila tidak ada/tidak diketahui heuristik, maka algoritma yang diterapkan adalah Dijkstra.

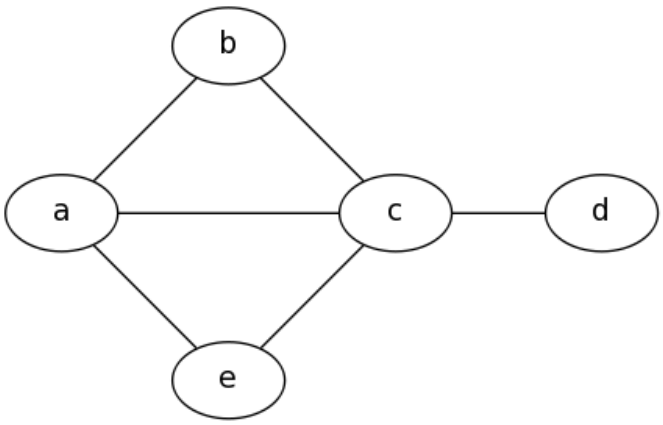
## II. DASAR TEORI

### A. Teori Graf

Sebelum kita masuk ke teori-teori pencarian algoritma maka dibutuhkan juga pengetahun mendasar tentang teori graf. Karena, map pada video-game diabstraksikan dalam bentuk graf.

"Graf G didefinisikan sebagai pasangan himpunan (V,E), ditulis dengan notasi G=(V,E), yang dalam hal ini V adalah himpunan tidak kosong dari simpul-simpul (*vertices* atau *node*) dan E adalah himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul. " -Rinaldi Munir, Matematika Diskrit Ed. 3, halaman 356.

Graf G=(V,E), yang dalam hal ini:
V = himpunan tidak kosong dari simpul-simpul (*vertices*)
= {v1, v2, … , vn}
E = himpunan sisi yang merupakan sepasang simpul (*edges*)
= {e1, e2, … , en}




*Gambar 2.1: Gambar Graf G Tak Berarah*

Contohnya, pada gambar di atas untuk Graf G=(V,E), dengan V adalah himpunan simpul dan E adalah himpunan sisi, maka terdefinisi:

V = {a, b, c, d, e}

E = { (a,b), (a,e), (a,c), (b,c), (c,d), (c,e) }

Contoh graf pada gambar 2.1 adalah contoh graf tidak berarah.

- **Graf tak-berarah** (*undirected graph*)

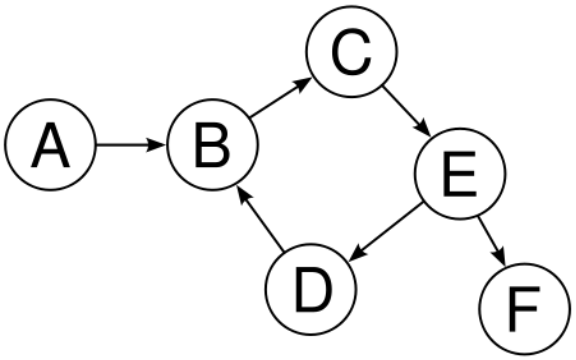
  Graf tak-berarah adalah graf yang sisinya tidak mempunyai orientasi arah, maka untuk contoh gambar 2.1 sisi E=(a,b) sama dengan E=(b,a).

  (a,b)=(b,a).

- **Graf berarah** (*directed graph*)

  Graf berarah adalah graf yang sisinya mempunyai orientasi arah, maka untuk contoh gambar 2.1 sisi E=(a,b) tidak sama dengan E=(b,a).

  (a,b) ≠(b,a).




*Gambar 2.2: Gambar Graf G Berarah*

Contohnya, pada gambar di atas untuk Graf G=(V,E), dengan V adalah himpunan simpul dan E adalah himpunan sisi, maka terdefinisi:

V = {A, B, C, D, E, F}

E = {(A,B), (B,C), (D,B), (E,D), (C,E), (E,F)}

Perlu diperhatikan bahwa pada graf berarah, orientasi sisinya sesuai tanda panah dari suatu simpul ke simpul. Misalnya, sisi (D, B) ada karena terdapat arah dari simpul D ke B, tetapi sisi (B,D) tidak ada karena tidak ada arah dari simpul B ke D.

**Terminologi Graf**

- Ketetanggaan (*Adjacency*)

  Dua buah simpul dikatakan bertetangga (*adjacent*) apabila keduanya terhubung sebuah sisi secara langsung. Tinjau gambar 2.1, simpul a *adjacent* dengan c, tetapi simpul a tidak *adjacent* dengan d.

- Bersisian (*Incidency*)

  Sebuah sisi dikatakan bersisian dengan simpul x apabila sisi tersebut mengandung simpul x. Misalkan untuk e1=(x,y), maka e1 bersisian dengan x dan e1 bersisian dengan y. Tinjau gambar 2.1, untuk sisi e=(a,b) maka sisi tersebut bersisian dengan simpul a dan simpul b.

**Graf Berbobot**

Graf Berbobot adalah graf yang setiap cabangnya memiliki bobot numerik. Graf berbobot adalah subtype graf berlabel (*labeled graph*) dimana setiap sisinya memiliki bobot numerik.

*B. A-Star Search*

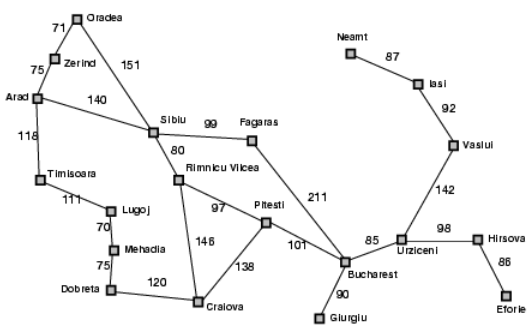
A* adalah algoritma pencarian terinformasi (*informed search algorithm*), yaitu algoritma yang menyelesaikan masalah dengan mencari semua solusi yang mungkin untuk menuju *goal-state* dan memilih solusi dengan *cost* terendah.

Persoalan digambarkan menggunakan graf berbobot, dengan bobot yang digambarkan adalah *cost* yang dimaksud pada persoalan. Pencarian dimulai dari sebuah simpul spesifik (yang dijadikan *start-state*) menuju sebuah simpul spesifik lainnya (yang dijadikan *goal-state*). Solusi dikonstruksikan dengan menggambar *tree* yang merupakan jalur yang pencarian. Simpul pertama *tree* adalah *start-state* pencarian, lalu diikuti dengan jalur2 yang mungkin menuju simpul selanjutnya. Simpul yang dipilih adalah simpul dengan *cost* terendah. Ide utama dari A* adalah menghindari mengembangkan simpul dengan *cost* yang terlalu mahal. Apabila *cost* dari simpul yang dikembangkan sudah terlalu mahal, maka A* akan kembali lagi untuk mencari *cost* yang lebih rendah lalu mengembangkan simpu tersebut.

Fungsi untuk meminimasi cost A*:

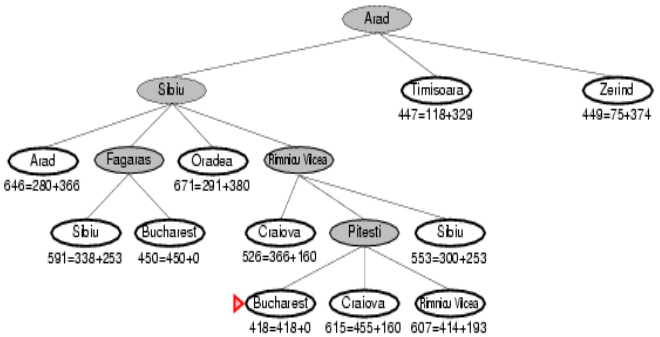
$$f(n) = g(n) + h(n)$$

g(n) adalah *cost* dari *start state* ke node n, sedangkan h(n) adalah estimasi *cost* dengan *cheapest path* dari n ke *goal state* (diketahui menggunakan heuristik). Maka, f(n) memenuhi estimasi *cost* yang merupakan *cheapest solution*. Strategi ini lebih dari *reasonable*, apabila fungsi heuristik h(n) memenuhi kondisi tertentu, maka A* *complete* juga optimal.




| Straight–line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 176 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 10 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

*Gambar 2.3: Graf Berbobot Arad-Bucharest*
*Sumber: Artificial Intelligence: A Modern Apporach, Third Edition, Stuart Russell*




*Gambar 2.4: Konstruksi tree untuk solusi Arad-Bucharest*
*Sumber: Artificial Intelligence: A Modern Apporach, Third Edition, Stuart Russell*

| Iterasi | Simpul-Ekspan | Simpul Hidup |
|---|---|---|
| 1 | Arad-366 | Sibiu-393, Timisoara-447, Zerind-449 |
| 2 | Sibiu-393 | Rimnicu Vilcea-413, Fagaras-415, Timisoara-447, Zerind – 449, Arad-646, Oradea-671 |
| 3 | Rimnicu Vilcea-413 | Fagaras-415, Pitesti-417, Timisoara-447, Zerind – 449, Craiova-526, Sibiu-553, Arad-646, Oradea-671 |
| 4 | Fagaras-415 | Pitesti-417, Timisoara-447, Zerind – 449, Bucharest-450 Craiova-526, Sibiu-553, Sibiu-591, Arad-646, Oradea-671 |
| 5 | Pitesti-417 | Bucharest-418, Timisoara-447, Zerind – 449, Bucharest-450 Craiova-526, Sibiu-553, Sibiu-591, Rimnicu Vilcea-607, Craiova-615, Arad-646, Oradea-671 |
| 6 | Bucharest-418 | Solusi ditemukan |

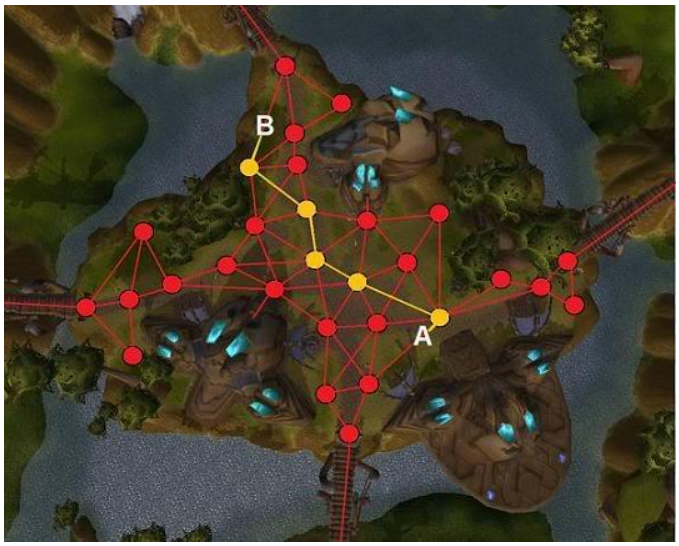
*Tabel 2.1: Solusi Arad-Bucharest*

## III. IMPLEMENTASI ALGORITMA A* PADA VIDEO-GAME

Ada dua cara untuk merepresentasikan peta pada video-game dalam bentuk graf, yaitu menggunakan *waypoint graph* dan *navigation meshes*. Tools dalam *game development* digunakan untuk mengkonstruksi graph ini secara otomatis yang nantinya akan digunakan konstruksi solusi *pathfinding*nya menggunakan algoritma A*.

### *A. Waypoint Graph*

Pada *waypoint graph*, setiap sudut yang merupakan *obstacle* (jalur yang tidak dapat dilalu) digambarkan sebagai simpul. Konstruksi graph menggunakan *waypoint graph* terlihat lebih simpel, akan tetapi terkadang konstruksinya akan menimbulkan terlalu banyak simpul. Hal ini menyebabkan algoritma A* akan bekerja lebih lama.




*Gambar 3.1: Konstruksi graph menggunakan waypoint graph*

### *B. Navigation Meshes*

Untuk *navigation meshes*, konstruksinya berdasarkan jalur yang dapat dilalui. Setiap jalur direpresentasikan sebagai sebuah simpul. Jalur ini sebenarnya adalah sebuah area yang sudut-sudutnya dibatasi oleh *obstacle*, dan area ini direpresentasikan sebagai simpul.


*Gambar 3.2: Konstruksi graph menggunakan navigation meshes*

### *C. Heuristik*

Untuk mengkomputasi *pathfinding* menggunakan algoritma A*, maka diperlukan heuristik yang *admissible*, karena algoritma A* bekerja sebaik heuristiknya. Heuristik yang sering digunakan untuk *pathfinding* A* pada *video-game* adalah *manhattan distance*.

Konstruksi graph baik menggunakan *waypoint graph* atau *navigation meshes* ini diletakkan di atas sebuah grid sehingga

setiap simpul ini menjadi sebuah titik P yang memiliki koordinat (x,y).

*Manhattan distance* adalah jarak antara kedua buah titik p1(x1,y1) dengan p2 (x2,y2), dimana jaraknya (D) adalah:

$$D = |x1-x2| + |y1-y2|$$

Selain itu, heuristik yang masih sering digunakan adalah *Chebyshev Distance* dan *Euclidean Distance*.

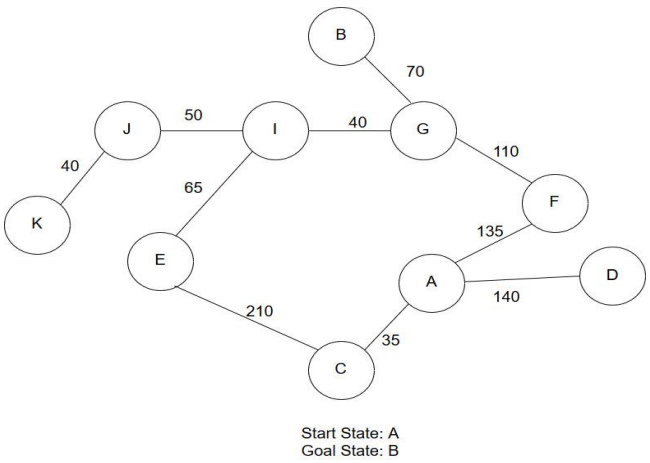
*D. Implementasi A**

Misalkan terdapat *navigation mesh* seperti gambar 3.2 sebagai abstraksi gambar dari *video-game*. Di sini saya gunakan *navigation mesh* karena jumlah simpulnya lebih sedikit. Akan tetapi untuk *game development*, *waypoint graph* dan *navigation mesh* sama-sama kerap digunakan.

Diketahui graph berbobot suatu video-game adalah sebagai berikut:


*Gambar 3.3: Graph Navigation Mesh dengan bobot*

Karena graph *nagivation mesh* menggunakan suatu area sebagai simpulnya, maka akan sulit untuk membayangkan simpul-simpulnya. Sementara, yang berbentuk bulat adalah sisinya. Berikut adalah representasi graph untuk peta video game ini, untuk mempermudah penglihatan:




*Gambar 3.4: Representasi graph berbobot dari graph navigation mesh dari video-game*

Sekarang kita sudah memiliki representasi graph dengan simpul yang lebih jelas, kita juga sudah memiliki bobotnya. Apa yang kurang? Kita masih belum memiliki heuristiknya. Karena kita akan menggunakan heuristik *manhanttan distance*, kita perlu mengetahui koordinat dari setiap simpul. Oleh karena itu, kita membutuhkan *node grid*. Biasanya *node grid* dapat di-*generate* secara otomatis menggunakan *game-development tools*. Tetapi karena jumlah simpul yang sedikit, akan saya konstruksi secara manual.

| **Koordinat** | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| 10 | | | | | | |
| 20 | | | | | | |
| 30 | | | | B<br>70 | | |
| 40 | | | | 70<br>G 110<br>40 | 110 F<br>135 | |
| 50 | | K 40 | 40 J<br>50 | 40<br>50 I<br>65 | 135<br>A 140<br>35 | 140 D |
| 60 | | | | 65<br>E 210 | 35<br>210 C | |

*Tabel 3.1: Node grid untuk graph 3.4*

*Start State* untuk graph ditandai dengan warna biru yaitu dengan simpul A, dan *goal state*nya ditandai dengan warna kuning yaitu simpul B. Pada setiap simpul, dituliskan bobot untuk ke simpul-simpul tetangganya.

Karena sudah mendapatkan koordinatnya, pertama kita akan mengkonstruksi *manhattan distance* setiap simpul ke *goal-state* yaitu B untuk digunakan sebagai heuristik.

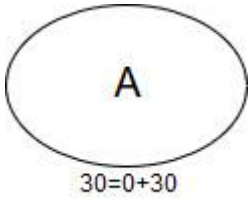
| **Simpul** | **Manhattan Distance ( h(n) )** |
|---|---|
| A | \|50-40\|+\|50-30\|=10+20= 30 |
| B | 0 (B adalah *goal state*) |
| C | \|50-40\|+\|60-30\|=10+30=40 |
| D | \|60-40\|+\|50-30\|=20+20=40 |
| E | \|40-40\|+\|60-30\|=30 |
| F | \|50-40\|+\|50-40\|=10+10=20 |
| G | \|40-40\|+\|40-30\|=10 |
| I | \|40-40\|+\|50-30\|=20 |

| | |
|---|---|
| J | \|30-40\|+\|50-30\|=10+20=30 |
| K | \|20-40\|+\|50-30\|=20+20=40 |

*Tabel 3.2: Heuristik Graph 3.4 menggunakan manhattan distance*

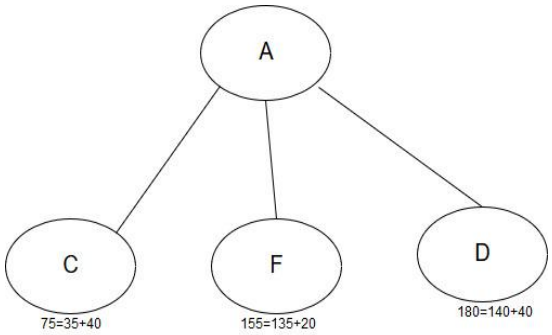
Setelah diketahui heuristiknya, maka kita dapat langsung mengkonstruksi solusi menggunakan algoritma A* berdasarkan heuristik yang ada.

- Initial State



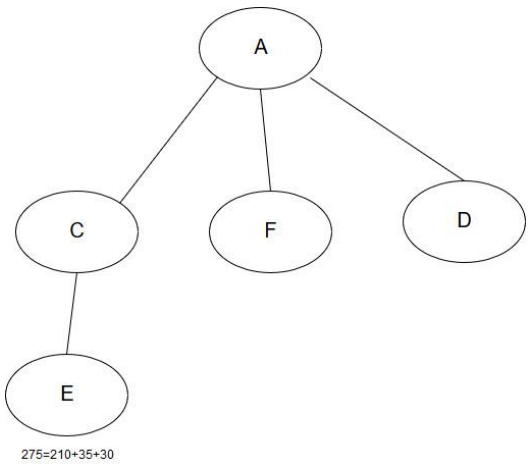

f(n)=g(n) + h(n) = 0+30=30

- Iterasi 2




Karena simpul C memiliki cost terkecil, maka simpul C dipilih dengan f(n)=75.

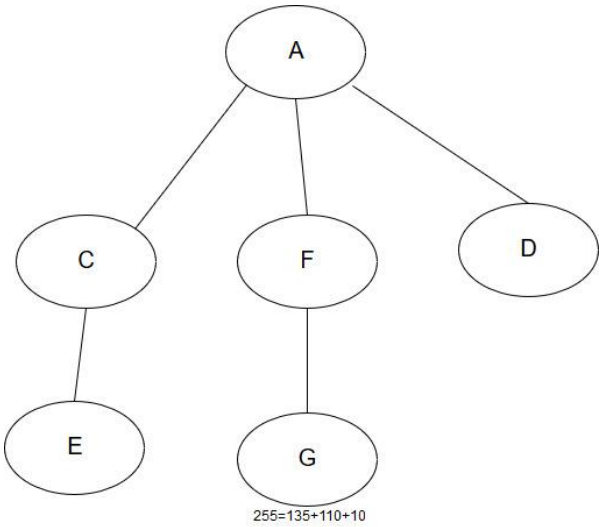
- Iterasi 3




f(n)=210+35+30=275

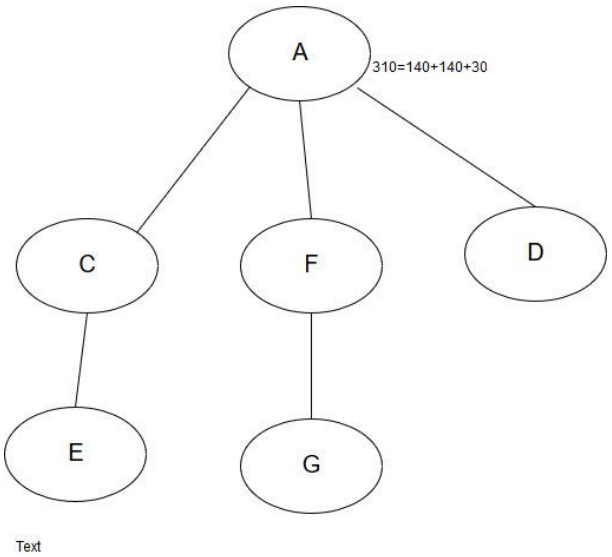
Karena terlalu mahal, dan masih ada simpul yang memiliki cost lebih rendah, maka Algoritma A* tidak mengembangkan E tapi mengembangkan yang lebih rendah antara F atau D.

- Iterasi 4



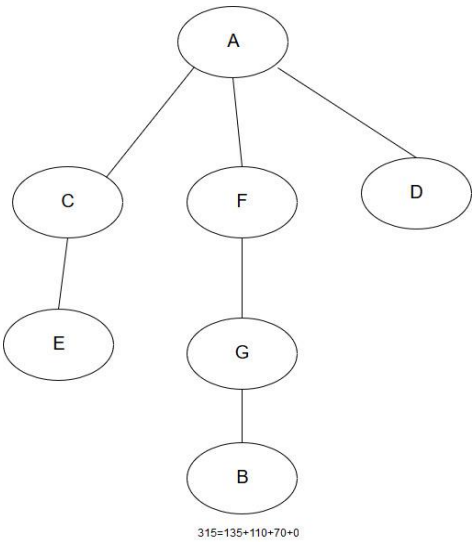

f(n)=135+110+10=255, masih lebih mahal dibandingkan simpul D

- Iterasi 5



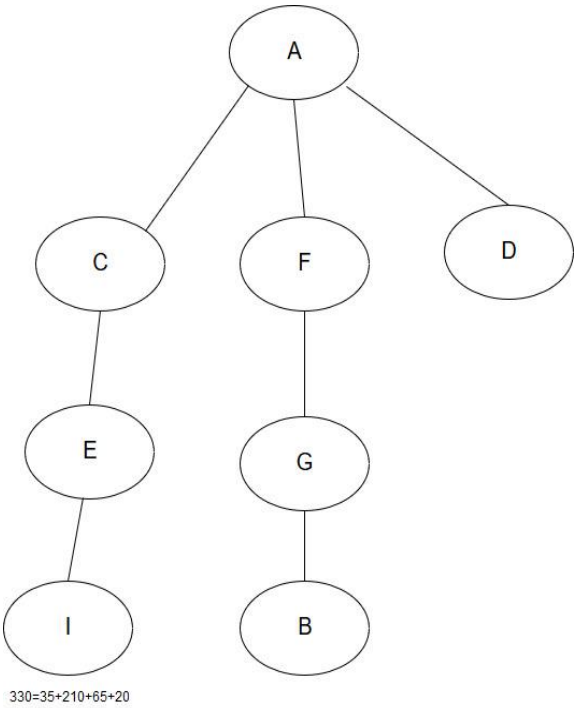

f(n)=140+140+30=310, akan tetapi karena simpul D tidak memiliki tetangga lainnya maka pilihan lainya hanya kembali ke A dan ini menyebabkan *cost* menjadi lebih mahal.

- Iterasi 6



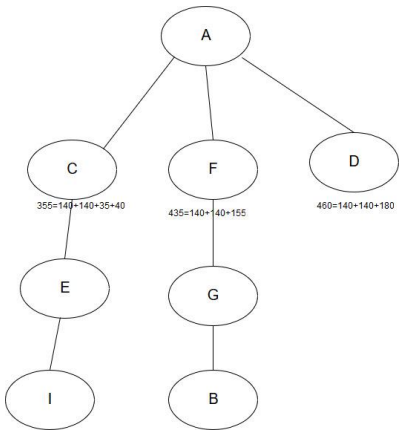

f(n)=135+110+70+0=315, pencarian belum selesai karena masih ada cost lebih rendah selain menuju simpul B.

- Iterasi 7

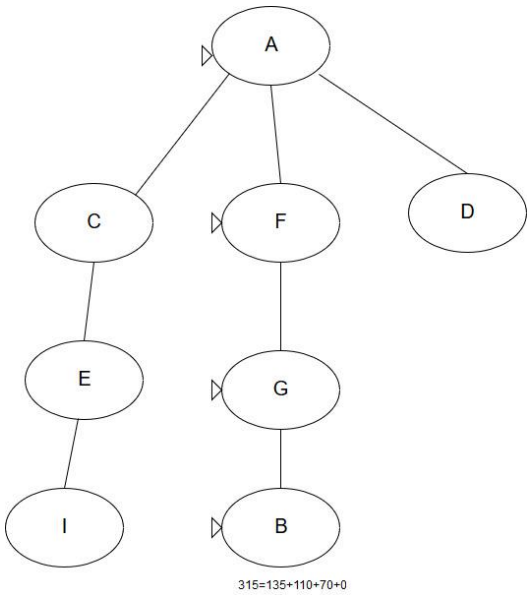



f(n)=35+210+65+20=330

- Iterasi 8




- Solusi akhir




| Iterasi | Simpul-Ekspan | Simpul Hidup |
|---|---|---|
| 1 | A-30 | C-75, F-155, D-180 |
| 2 | C-75 | F-155, D-180, E-275 |
| 3 | F-155 | D-180, G-255, E-275 |
| 4 | D-180 | G-255, E-275, A-310 |
| 5 | G-255 | E-275, A-310, B-315 |
| 6 | E-275 | A-310, B-315, I-330 |
| 7 | A-310 | B-315, I-330, C-255, F-435, D-460 |
| 8 | B-315 | Goal State |

*Tabel 3.3: Ilustrasi Algoritma A* pada graph*

Pencarian berakhir pada *goal state* yaitu graph B dengan *total cost* f(n)=315.

## IV. KESIMPULAN

Ada banyak algoritma yang dapat digunakan untuk pencarian rute (*pathfinding*) pada video-game. Untuk setiap kasus suatu pencarian algoritma bisa jadi lebih optimal dari

alogirtma lainnya. Akan tetapi, untuk kasus lain suatu algoritma tersebut bisa lebih buruk dari algoritma lainnya. Kelebihan algoritma A* adalah *complete* dan optimal. Algoritma ini menjamin tidak akan menjebak pemakainya pada *local minima* seperti algoritma *greedy-best-first-search*. Selain itu, algoritma A* juga memberikan hasil yang optimal, dibandingkan algoritma lainnya. Hal ini dikarenakan algoritma A* memperhitungkan cost yang sudah dilalui g(n) dengan heuristiknya h(n). Walaupun begitu, keoptimalan algoritma A* bergantung pada heuristiknya. Kelemahan algoritma A* seperti yang bisa kita lihat pada pembahasan di bab III, adalah pengerjaannya yang cukup lama, ekspansinya cukup banyak untuk mencari solusi yang komplit.

## V. REFERENSI


[1] Russel, Stuart. Artifical Intelligence: A Modern Apporach 3rd Edition

[2] Munir, Rinaldi. Matematika Diskrit 3rd Edition

[3] www.ai-blog.net

[4] https://xlinux.nist.gov/dads/HTML/manhattanDistance.html

[5] http://mathworld.wolfram.com/WeightedGraph.html

[6] http://www.mrgeek.me/wp-content/uploads/2014/04/directed-graph.png

[7] http://graphs.grevian.org/resources/static/images/example1.png

[8] https://upload.wikimedia.org/wikipedia/commons/thumb/b/bc/CPT-Graphs-directed-weighted-ex1.svg/2000px-CPT-Graphs-directed-weighted-ex1.svg.png


## 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, 29 April 2012

Emilia Andari Razak - 13515056