Memecahkan Algoritma Pathfinding DotA 2 dengan Pendekatan Algoritma Dynamic Programming

Muhammad Tito Prakasa 13519007
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): muhammadtitoprks@gmail.com

***Abstract*—Seiring berkembangnya teknologi, semakin berkembang juga kompleksitas *game-game* saat ini. Salah satunya adalah *game* DotA 2 yang memanfaatkan *pathfinding* di dalamnya. *Pathfinding* ini dapat didekati dengan algoritma *dynamic programming*, atau algoritma yang memiliki ciri khas membagi persoalan menjadi beberapa *stage* dan mendapatkan solusi dari hubungan *stage-stage* tersebut.**

***Keywords*—DotA, *Pathfinding*, *dynamic programming*, rute, optimal**

I. Pendahuluan

Permainan merupakan salah satu objek hiburan bagi manusia. Pada perkembangannya permainan dapat didukung oleh teknologi secara *offline* maupun *online* (biasanya menggunakan basis *online* untuk bermain bersama). Salah satu dari permainan yang didukung teknologi adalah DotA 2, permainan 5 *vs* 5 berbasis *Multiplayer Online Battle Arena*.

Gambar 1. Game DotA 2

DotA 2 merupakan permainan yang cukup kompleks. Namun sebenarnya objektif dari game ini cukup sederhana, yaitu masing-masing tim harus mempertahankan *ancient* (semacam base utama) dan menghancurkan *ancient* lawan. Tim yang kalah adalah tim yang *ancient*-nya hancur lebih dulu. Kompleksitasnya terkandung dalam strategi untuk menghancurkan *ancient* lawan tetapi di saat yang bersamaan juga harus mempertahankan *ancient* sendiri.

Teknologi krusial yang mendukung permainan ini adalah algoritma dalam mekanikalnya seperti, algoritma pergerakan *creep* (pasukan "bot" yang muncul secara periodik tiap 30 detik), algoritma distribusi *exp* dan *gold*, dan salah satu yang paling krusial adalah algoritma *pathfinding*. *Pathfinding* merupakan pencarian jalur dari suatu titik ke titik yang dituju dengan jarak seminimal mungkin. Dalam DotA 2 pergerakan pemain dilakukan dengan mengklik sebuah area dalam peta lalu selanjutnya *hero* kita akan berjalan ke arah area tersebut dengan jalur yang minimal.

Sayangnya, algoritma *pathfinding* di DotA 2 tidak diberitahukan secara eksplisit oleh *developer* bagaimana cara kerjanya. Oleh karena itu, penulis berusaha memecahkan algoritma tersebut dengan pendekatan algoritma *Dynamic programming*. Dalam makalah ini akan dijelaskan bagaimana cara kerja dan pendekatan dari algoritma *Dynamic programming* dengan *pathfinding*.

II. Landasan Teori

A. Algoritma Dynamic Programming

Dynamic programming adalah metode pemecahan masalah dengan men-dekomposisi masalah menjadi *stage-stage* yang berkaitan. Solusi didapatkan dengan mencari nilai optimal (lokal) pada setiap *stage*-nya dengan harapan pada *stage* akhir didapatkan solusi yang optimal (global).

Gambar 2. Ilustrasi Dynamic Programming
Sumber: The importance of dynamic programming (lynda.com)

Kata dinamis dalam *dynamic programming* memiliki arti perhitungan solusi memanfaatkan ruang nilai (biasanya

menggunakan tabel) yang dapat berkembang secara dinamis. Ruang nilai ini berguna menyimpan nilai-nilai optimal lokal sehingga dalam pencarian solusi kita dapat memanfaatkan nilai-nilai tersebut.

Mendengar kata "optimal lokal" dan "optimal global", sekilas algoritma ini mirip dengan algoritma *greedy* yang juga memiliki ide untuk terus mencari optimal lokal dengan harapan solusi akan menuju optimal global. Namun perbedaan mendasar kedua algoritma ini adalah jumlah pengambilan keputusannya. Dalam algoritma *greedy* hanya ada satu rangkaian pengambilan keputusan untuk tiap solusinya. Berbeda dengan *dynamic programming* yang menyimpan nilai setiap optimal lokal sehingga mampu menciptakan beberapa pengambilan keputusan pada prosesnya dan menghasilkan beberapa rangkaian pengambilan keputusan pada solusinya.

1. Prinsip Optimalitas

   Prinsip optimalitas berbunyi, "Jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal." (Diktat Kuliah Strategi Algoritmik IF2211, *Dynamic Programming*). Prinsip ini memiliki arti jika kita ingin melanjutkan pencarian solusi dari *stage k* ke *stage* k+1, maka bisa dipastikan nilai pada *stage k* telah optimal (tanpa harus mengecek ulang atau menghitung ulang). Jadi, bisa dibayangkan nilai pada *stage k+1* adalah

   $$nilai(k+1) = nilaiOptimal(k) + nilaiPindah(k,$$

2. Karakteristik Persoalan dengan *Dynamic Programming*

   2.1. Persoalan bisa dibagi menjadi beberapa *stage* yang pada setiap *stage*-nya diambil satu keputusan.

   2.2. Setiap *stage* terdiri dari beberapa status yang merupakan kemungkinan masukan untuk *stage* tersebut.

   2.3. Hasil dari keputusan yang diambil pada setiap *stage*-nya akan disimpan dan digunakan sebagai referensi untuk pengambilan keputusan *stage* selanjutnya.

   2.4. Nilai pada setiap *stage*-nya meningkat secara teratur.

   2.5. Adanya hubungan rekursif antara keputusan-keputusan yang diambil.

   2.6. Prinsip optimalitas bekerja di situ.

3. Dua Pendekatan *Dynamic Programming*

   Dalam pengembangan ruang nilai, *dynamic programming* dibagi menjadi dua pendekatan, yaitu:

   - *Top-down* (*Forward*)

     Pembangunan ruang solusi dimulai dari *stage* 1,2,3,...,n

   - *Bottom-up* (*Backward*)

     Pembangunan ruang solusi dimulai dari *stage* n,n-1,n-2,...,1

4. Langkah-langkah Pengembangan Algoritma

   4.1. Karakteristikkan struktur solusi optimal, berupa *stage*, tahap, variabel penentu keputusan, dll.

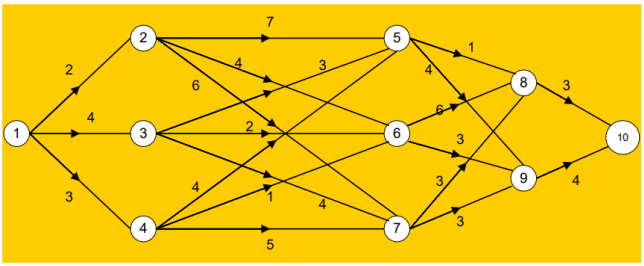
   4.2. Definisikan secara rekursif nilai dari solusi optimal, hubungkan nilai dari suatu *stage* dengan nilai dari *stage* sebelumnya.

   4.3. Hitung nilai secara maju atau mundur (dari dua pendekatan *dynamic programming*).

   4.4. Rekonstruksi solusi optimal untuk memungkinkan kemunculan solusi lebih dari satu rangkaian keputusan.

5. Contoh Persoalan

   Mencari lintasan terpendek dari titik 1 ke titik 10 pada gambar ini.



Gambar 3. Contoh Persoalan Pencarian Lintasan Terpendek
Sumber: Diktat Kuliah IF2211 Strategi Algoritma, Dynamic Programming

Langkah-langkah penyelesaian:

- Karakteristikkan struktur solusi optimal

  Stage (k) merupakan pemilihan titik selanjutnya dengan penentuan pemilihan titik selanjutnya berdasarkan lintasan yang memiliki ongkos paling kecil. Sedangkan

  status (s) merupakan titik-titik di dalam setiap *stage*-nya.

  Stage dibagi menjadi x1 (titik 1), x2 (titik 2,3,4), x3 (titik 5,6,7), x4 (titik 8,9), x5 (titik 10).

- Pendefinisian secara rekursif

  Basis: $f_1(s) = c_{x1,s}$

  Rekurens: $f_k(s) = \min(f_{k-1}(x_k) + c_{xk,s})$

  a. x_k : peubah keputusan pada tahap k (k = 2, 3, 4).

b. s : status pada setiap tahap

c. $c_{xk,s}$: bobot (cost) sisi dari ke x_k ke s

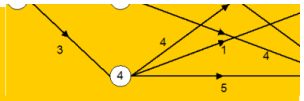
d. f_k (s) : nilai minimum dari $f_k(x_1, s)$

e. $f_{k-1}(x_k)$: nilai minimum tahap sebelumnya dari x_k ke s

- Hitung maju

Tahap 1:

$$f_1(s) = c_{x1s}$$



s	Solusi Optimum	
	$f_1(s)$	x_1^*
2	2	1
3	4	1
4	3	1

$f_1(s) = c_{x1,s}$

Catatan: x_k^* adalah nilai x_k yang meminimumkan f_s.

Tahap 2:

$$f_2(s) = \min_{x_2}\{f_1(x_2) + c_{x_2 s}\}$$

s \ x_2	$f_2(s) = f_1(x_2) + c_{x2,s}$			Solusi Optimum	
	2	3	4	$f_2(s)$	x_2^*
5	9	7	7	7	3 atau 4
6	6	6	4	4	4
7	8	8	8	8	2, 3, 4

$f_k(s) = \min\{f_{k-1}(x_k) + c_{xk,s}\}$

Tahap 3:

$$f_3(s) = \min_{x_3}\{f_2(x_3) + c_{x_3 s}\}$$

s \ x_3	$f_3(s) = f_2(x_3) + c_{x3,s}$			Solusi Optimum	
	5	6	7	$f_3(s)$	x_3^*
8	8	10	11	8	5
9	11	7	11	7	6

Tahap 4:

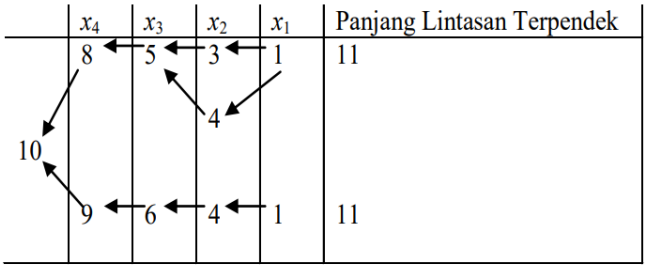
$$f_4(s) = \min_{x_4}\{f_3(x_4) + c_{x_4 s}\}$$

s \ x_4	$f_4(s) = f_3(x_4) + c_{x4,s}$		Solusi Optimum	
	8	9	$f_4(s)$	x_4^*
10	11	11	11	8 atau 9

$f_k(s) = \min\{f_{k-1}(x_k) + c_{xk,s}\}$

Gambar 4. Kumpulan Perhitungan Solusi
Sumber: Diktat Kuliah IF2211 Strategi Algoritma, Dynamic Programming

- Rekonstruksi solusi



Jadi ada tiga lintasan terpendek dari 1 ke 10, yaitu

$1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10$
$1 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 10$
$1 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 10$

yang mana panjang ketiga lintasan tersebut sama, yaitu 11.

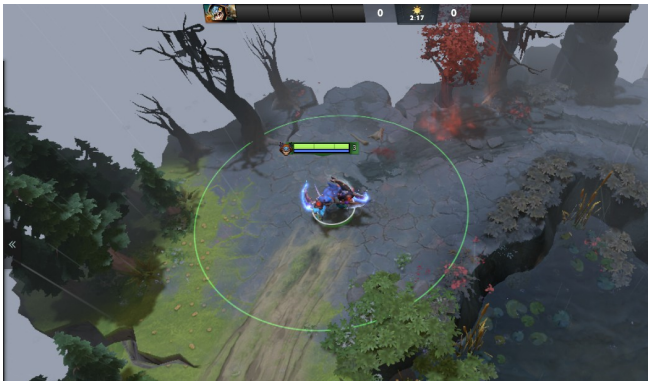
Gambar 5. Rekonstruksi Solusi
Sumber: Diktat Kuliah IF2211 Strategi Algoritma, Dynamic Programming

B. Mekanik Dasar pada Game DotA 2

Sebagai *game* berbasis MOBA, DotA 2 memiliki banyak mekanik di dalamnya. Tetapi kali ini akan penulis jelaskan mekanik yang berhubungan dengan *pathfinding* di DotA 2. Berikut beberapa mekanik tersebut:

- Besaran Unit

  Semua entitas di dalam DotA menggunakan satuan unit dan unit mewakili sebuah lingkaran. Semakin besar unit suatu entitas maka semakin besar pula lingkaran tersebut.



Gambar 6. Contoh Besar Unit 400 (Lingkaran Hijau)

- *Movement*

  Pergerakan *hero/player* terbagi menjadi tiga jenis, yaitu *move tool* yang berfungsi mencari jalur terpendek ke titik yang ingin dituju. Selanjutnya

terdapat *attack tool* yang berfungsi mencari entitas terdekat yang bisa diserang. Dan terakhir adalah *directional tool* yang berfungsi menggerakan secara lurus (tanpa memikirkan entitas lain) ke titik tujuan. Pada pembahasan selanjutnya akan difokuskan pada *move tool*.

Kecepatan pergerakan diwakili oleh suatu nilai yang satuannya *ms*. 1 *ms* memiliki arti pemain bergerak sejauh 1 unit selama 1 detik.

Gambar 7. Contoh Movement Speed 300 ms

- Benda-benda di DotA 2

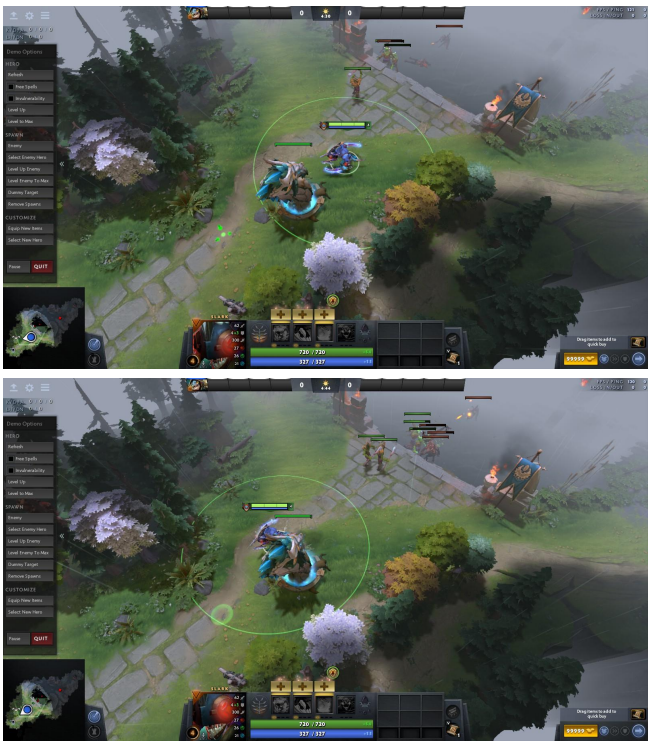
  Jika *hero/player* adalah entitas yang dinamis (bisa bergerak) di dalam *map*. Maka ada juga entitas yang statis seperti *tower, tree, barrack, wall,* dll. Masing-masing entitas statis memiliki nilai unit yang fix dengan nilai sebagai berikut (khusus *tree* memiliki besar 128 unit).

Hull Name	Size
DOTA_HULL_SIZE_BUILDING	298
DOTA_HULL_SIZE_TOWER	144
DOTA_HULL_SIZE_FILLER	96
DOTA_HULL_SIZE_HUGE	80
DOTA_HULL_SIZE_HERO	24
DOTA_HULL_SIZE_REGULAR DOTA_HULL_SIZE_SIEGE	16
DOTA_HULL_SIZE_SMALL	8
DOTA_HULL_SIZE_SMALLEST	2

Gambar 8. Nilai Unit Masing-masing Entitas di DotA 2
Sumber: Collision size - Dota 2 Wiki (fandom.com)

- *Collision*

  Entitas dinamis ketika bergerak tidak bisa melewati entitas lain (statis maupun dinamis). Area yang tidak bisa dilewati sesuai dengan besaran unit pada masing-masing entitas di gambar sebelumnya. Ketika sebuah entitas mencoba untuk melewati area tersebut, maka otomatis akan bergerak menghindari area tersebut.



Gambar 9. Contoh Terjadinya Collision

- *Pathfinding*

  Sebagai simplifikasi, pergerakan yang diperhitungkan hanya pergerakan normal (bukan *flying* atau *phased* yang bisa menembus entitas lain). Seperti yang telah dijelaskan sebelumnya, *move tool* berfungsi untuk mencari jalur terpendek dari posisi *player* saat ini ke titik tujuan. Idealnya, dalam pencarian rute tidak ada entitas menghalangi, maka *player* akan bergerak secara lurus menuju titik tujuan.

Gambar 10. Ideal Pathfinding

Jika terdapat kondisi dalam penarikan garis lurus dari posisi saat ini ke posisi tujuan terhalang suatu entitas, maka secara otomatis *player* akan mencari jalan lain yang bebas dari entitas dan merupakan jarak terpendek dari semua kemungkinan.

Gambar 11. Pathfinding ketika Terhalang Entitas (Tree dan Wall)

III. Hasil dan Pembahasan

Secara sekilas contoh *pathfinding* telah diberikan pada pembahasan sebelumnya. Namun, kali ini penulis akan berusaha mem-*breakdown* apa saja yang terjadi ketika ketik mengklik *move tool*.

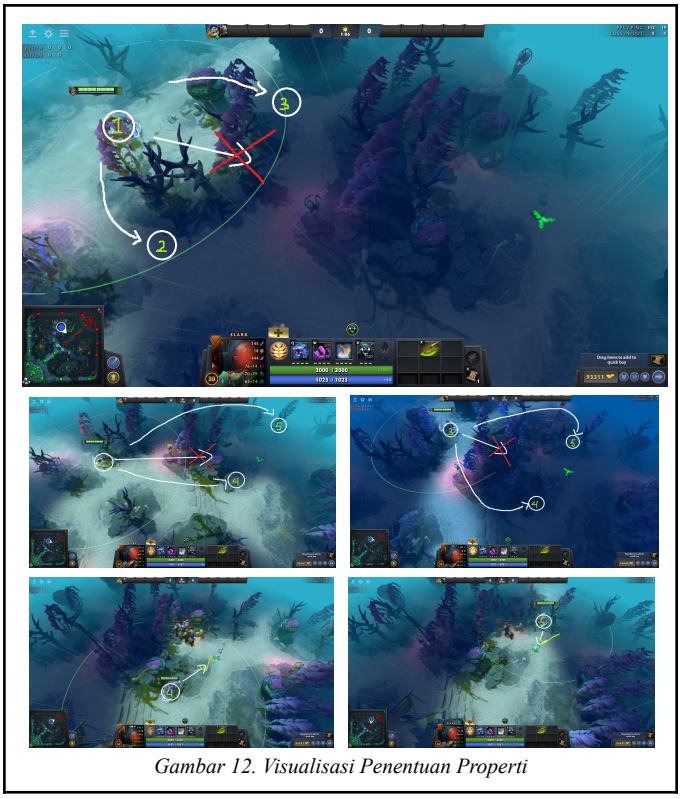
Dengan pendekatan *dynamic programming* penulis akan membagi pencarian rute dari posisi saat ini hingga posisi tujuan menjadi beberapa *stage*. Lalu bagaimana cara menentukan propertinya? Pertama-tama kita lihat bagaimana *behaviour* dari pergerakan *player* ketika terhalang oleh entitas.

Dapat dilihat pada gambar ke-2 pada gambar 11, *player* memutari kumpulan entitas dengan jarak seminimal mungkin (hampir menempel dengan entitas tersebut). Selanjutnya ketika sudah melewati kumpulan entitas pertama, *player* selanjutnya "menghindari" juga kumpulan entitas kedua dengan cara yang sama seperti tadi (menempel entitas). Terakhir ketika sudah tidak ada entitas yang menghalangi maka *player* dapat bergerak lurus ke titik tujuan dan berhenti jika sampai.

C. *Analisa Singkat*

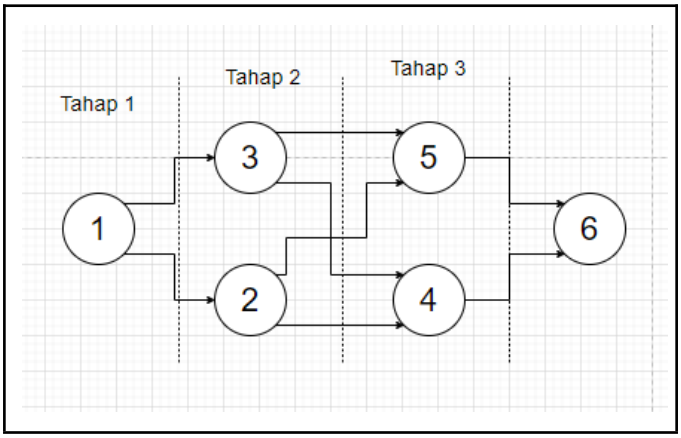
Dari penjelasan sebelumnya, penulis mengasumsikan entitas yang menghalangi akan menjadi *stage*. Kemudian pilihan *player* untuk memutar searah jarum jam (kanan), memutar lawan arah jarum jam (kiri), atau bahkan jika ternyata di tengah kumpulan unit terdapat *empty* unit dan besarnya sesuai dengan size *player* (tengah) menjadi status. Terakhir, variabel penentu keputusannya adalah ongkos (berupa waktu dalam s) yang dibutuhkan untuk mencapai suatu titik.

Penentuan apakah suatu kumpulan entitas perlu menjadi *stage* didapatkan melalui penarikan garis lurus dari posisi *player* tepat di sebuah status ke posisi tujuan. Apakah garis lurus tersebut terhalang kumpulan entitas lain? Jika ya maka buat *stage* baru, jika tidak maka *player* sudah bisa bergerak lurus ke posisi tujuan.



Gambar 12. Visualisasi Penentuan Properti

Dari gambar 12, asumsi awal cukup masuk akal untuk menjadikan kumpulan entitas sebagai *stage* dan pilihan arah gerak sebagai status. Selanjutnya untuk variabel penentu keputusan karena perhitungan jarak dalam game DotA 2 cukup sulit (perlu mengira-ngira dengan besaran unit) maka digunakan asumsi awal yaitu menggunakan waktu dalam s untuk mencapai suatu titik. Jadi jika dibuat dalam graf maka bentuk properti dari contoh tersebut:



D. Percobaan

Untung menghitung ongkos, penulis menggunakan *stopwatch handphone*, *hero* dengan *movement speed* 444 ms, dan didapatkan hasil sebagai berikut:

- Simpul 1 - 2 = 1.93 s
- Simpul 1 - 3 = 1.97 s
- Simpul 2 - 4 = 2.63 s
- Simpul 2 - 5 = 6.8 s
- Simpul 3 - 4 = 3.21 s
- Simpul 3 - 5 = 4.46 s
- Simpul 4 - 6 = 1.12
- Simpul 5 - 6 = 1.05 s

E. Perhitungan

1. Karakteristik Struktur Solusi Optimal
   Stage terbagi menjadi 3 sehingga terdapat tiga kali pengambilan keputusan tiap rangkaian solusi. *Stage* terakhir (4) tidak ditulis karena merupakan tujuan.

2. Pendefinisian Secara Rekursif
   Dengan ongkos merupakan waktu dalam s menuju simpul (status) selanjutnya, maka

   Basis: $f_1(s) = c_{x1,s}$

   Rekurens: $f_k(s) = \min(f_{k-1}(x_k) + c_{xk,s})$

3. Hitung Maju

   Tahap 1, $f_1(s) = c_{x1,s}$

s	Solusi Optimal	
	$f_1(s)$	x_1*
2	1.93	1
3	1.97	1

Tahap 2, $f_2(s) = \min (f_1(s) + c_{x2,s})$

s	Solusi Optimal	
	$f_1(s)$	x_1*
4	4.56	2
5	6.43	3

Tahap 3, $f_3(s) = \min (f_2(s) + c_{x3,s})$

s	Solusi Optimal	
	$f_1(s)$	x_1*
6	5.68	4

4. Rekonstruksi Solusi
   Jarak terpendek didapat dari rute 1 -> 2 -> 4 -> 6, yaitu memiliki ongkos 5.68 s dan tidak ada rangkaian keputusan yang memiliki ongkos kurang dari atau sama dengan rangkaian keputusan tersebut.

F. Analisa Mendalam

Dalam DotA 2, kecepatan memperhitungkan rute mana yang terdekat cukup krusial, terlebih *game* ini merupakan *multiplayer* dan daring sehingga *delay-delay* kecil akan cukup terasa dalam pengalaman bermain. Maka dari itu, pendekatan algoritma ini perlu diperhitungkan kompleksitas waktunya.

Flow dalam pencarian rute terdekat menurut penulis terbagi menjadi tiga bagian, yaitu:

1. Tarik garis lurus dari posisi saat ini ke posisi tujuan dan cek apakah ada entitas yang menghalangi
2. Jika ada, telusuri berapa banyak posisi sehingga jika ditarik garis lurus dari posisi tersebut ke posisi tujuan tidak lagi terhalang. Tambahkan dalam matriks jarak (2-dimensi) berapa saja ongkos dari posisi awal ke posisi hasil penelusuran.
3. Jika tidak, langsung masukkan jarak dari posisi saat ini ke posisi tujuan.
4. Tarik garis lurus kembali dari masing-masing posisi hasil penelusuran ke posisi tujuan. Lalu cek kembali apakah ada halangan, ulangi langkah 2 atau 3.
5. Terus ulangi langkah 2, 3, dan 4 hingga semua ongkos kemungkinan telah dicatat.
6. Jalankan algoritma *dynamic programming* seperti sebelumnya berdasarkan data jarak yang kita dapatkan.

Jika dibuat dalam pseudocode maka kira-kira akan seperti ini:

```
Array<Pos> findShortestRoute(Pos src, Pos dst){
  Array<Array<int>> matrixDistance = new
Array<Array<int>>;
  Array<Pos> possibleState = new Array<Pos>;
  Entities isHindered = checkLinier(src,dst);
  while(isHindered!=null){

recursiveUpdate(matrixDistance,p,isHindered,dst,possibleS
tate);
  }

  //Saat di sini, matrixDistance telah terisi lengkap
  return
processDynamicProgramming(src,dst,matrixDistance);
}
```

Menghitung kompleksitas waktu:

1. Membuat variabel-variabel, matrixDistance untuk menyimpan jarak, possibleState untuk menyimpan posisi hasil penelusuran. Constant-time O(1)
2. Fungsi checkLinier, fungsi yang berguna untuk mengecek apakah ada entitas yang menghalangi atau tidak. Constant-time O(1)
3. Fungsi recursiveUpdate, menelusuri entitas yang menghalangi (sebanyak misal n) lalu meng-*update* matrixDistance dan possibleState. Kemudian secara rekursif menghitung juga masing-masing status dalam possibleState (sebanyak misal m). MultiLinear-time O(nm)
4. Fungsi processDynamicProgramming, membuat susunan rute terdekat berdasarkan posisi awal, posisi tujuan, dan matrixDistance. Dilakukan penelusuran sebanyak *stage* sehingga jika jumlah *stage* adalah k, maka linier-time O(k).

Jadi, total kompleksitas waktu adalah

$$T(n) = O(1) + O(1) + O(nm) + O(k) = O(nm+k)$$

IV. Kesimpulan

Pendekatan *dynamic programming* pada *pathfinding* DotA 2 cukup berhasil karena menghasilkan rute yang sama dengan yang ada di DotA-nya langsung. Namun, untuk kompleksitas waktu algoritmanya masih cukup besar untuk ukuran *game multiplayer online*. Penulis mungkin akan mengoptimasi algoritma kedepannya, namun untuk saat ini cukup kita nikmati algoritma tersembunyi yang DotA 2 sediakan :).

Link Penjelasan Video

Silahkan akses link di bawah ini untuk visualisasi penjelasan yang lebih jelas.

https://youtu.be/XVkfrtEcnBE

Ucapan Terima Kasih

Penulis mengucapkan terima kasih dan puji syukur atas rahmat Allah SWT karena dengan izin-Nya penulisan makalah ini dapat selesai. Tidak lupa juga penulis mengucapkan terima kasih untuk Bapak/Ibu Dosen IF2211 Strategi Algoritma terutama Pak Rila selaku pengajar K1 atas ilmu berharganya yang menjadi dasar penulisan makalah ini. Selain itu, penulis juga ingin mengucapkan terima kasih kepada teman-teman IF 2019 atas pengalaman belajar bersamanya sehingga dapat memahami materi lebih baik. Terakhir, penulis merasa terima kasih dengan makalah ini sendiri karena memberi *insight* baru tentang dunia DotA 2 dari sisi yang sebelumnya tidak terbayangkan.

Referensi

[1] Rinaldi Munir, Diktat kuliah IF2211 Strategi Algoritma, Teknik. Informatika ITB.

[2] https://www.geeksforgeeks.org/dynamic-programming/. Diakses pada tanggal 10 Mei 2021.

[3] Collision size - Dota 2 Wiki (fandom.com). Diakses pada tanggal 11 Mei 2021.

[4] Trees - Dota 2 Wiki (fandom.com). Diakses pada tanggal 11 Mei 2021.

[5] Time Complexity: What is Time Complexity & Algorithms of it? (mygreatlearning.com). Diakses pada tanggal 11 Mei 2021.

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.

Jakarta, 11 Mei 2021

Muhammad Tito Prakasa 13519007