Perbandingan Efisiensi Algoritma *Path Finding* untuk Simulasi Permainan Labirin dalam *Game Engine*

Johanes Lee - 13521148
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521148@std.stei.itb.ac.id

***Abstract*—Video game development is no longer a foreign concept to many people. It is not uncommon for video game development to involve heavy calculations in the game's core logic. Since this logic often becomes a selling point and the most important aspect of the game being developed, an efficient algorithm is needed to ensure that the game can still run smoothly on the game engine without experiencing frame drops and provide the best experience for players. This paper highlights the importance of algorithm efficiency in achieving this goal. The test results show that selecting the appropriate algorithm can significantly enhance the capabilities of the developed video game in maintaining frame rates as the size of the data to be processed increases.**

Keywords—game development; game engine; path finding; algorithm

I. Pendahuluan

Industri *video game* telah berkembang pesat dan istilah *game development* sudah tidak asing lagi di banyak kalangan. Perkembangan tersebut telah melahirkan berbagai kakas atau *framework* yang mempermudah proses pengembangan *video game*. *Game engine* merupakan salah satu komponen utama dalam proses pengembangan dan pemeliharaan *video game*. Dengan tersedianya *framework* berupa *game engine* tersebut, para pengembang tidak perlu membangun ulang seluruh sistem yang diperlukan dalam permainan yang ingin dibuat. Selain itu, sudah banyak *game engine* yang disediakan gratis (walaupun jasa yang disediakan dapat dibatasi) oleh beberapa perusahaan yang berfokus pada bidang tersebut, seperti Godot Engine, Unity, dan Unreal Engine.

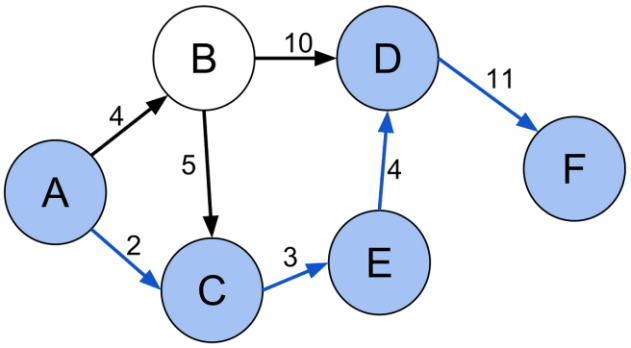
Multithreading merupakan konsep yang umum ditemukan pada *game engine*. Hal ini didukung kemajuan CPU yang sudah berkembang pesat sehingga memiliki banyak *core* yang dapat menjalankan banyak *thread* dalam suatu waktu. Salah satu *thread* di dalam *game engine* bertugas untuk menjalankan suatu tugas dalam periode tetap ataupun jangka waktu yang dapat berubah (berhubungan dengan *frame rate*). Salah satu contoh konsep yang mirip dapat terlihat pada *method* Update dan FixedUpdate pada kelas MonoBehavior yang disediakan *framework* Unity [8] (setiap *framework* dapat memiliki mekanisme yang berbeda). Perhitungan berat (memakan waktu lama) di dalam salah satu dari kedua *method* tersebut dapat menggangu jadwal eksekusi *method* dan membuat *game* tidak berjalan mulus (*frame drop*).

Tidak jarang suatu *video game* perlu melibatkan perhitungan yang berat. Tidak jarang pula perhitungan tersebut melibatkan logika utama permainan yang dikembangkan. Karena hal tersebut, banyak pengembang menggunakan aproksimasi ataupun heuristik untuk mempercepat perhitungan dan menghindari *frame drop* di dalam *game engine* akibat perhitungan tersebut. Walaupun pendekatan tersebut dapat memberikan aproksimasi perhitungan yang wajar, hal tersebut tidak cocok untuk logika permainan yang membutuhkan hasil akurat dan optimal. Oleh karena itu, diperlukan algoritma yang efisien dalam perhitungan berat untuk logika permainan yang dikembangkan. Makalah ini menyimulasikan kebutuhan algoritma yang efisien dalam perhitungan logika permainan labirin untuk mendapatkan *frame rate* permainan yang optimal.

II. Landasan Teori

A. Graf

Graf merupakan salah satu struktur data yang digunakan dalam menunjukkan hubungan berbagai objek ataupun data [9]. Objek atau data tersebut dinyatakan sebagai simpul (*node*) dan dapat dihubungkan dengan suatu sisi (*edge*). Dengan demikian, struktur data graf dapat direpresentasikan sebagai himpunan simpul dan himpunan sisi dengan masing-masing sisi menghubungkan dua buah simpul [6]. Suatu graf dapat berarah (*directed graph*) [9], yaitu suatu sisi hanya menghubungkan satu simpul ke simpul lainnya tetapi tidak sebaliknya. Selain itu, graf juga dapat memiliki bobot pada setiap sisinya (*weighted graph)* untuk memberikan informasi tambahan mengenai hubungan antar simpul [9]. Salah satu contoh *directed graph* sekaligus *weighted graph* terlihat pada gambar 1.



Gambar 1. Sirkuit dalam graf
(Sumber: https://medium.com/swlh/pathfinding-algorithms-6c0d4febe8fd, diakses pada 20 Mei 2023)

B. *Matriks Ketetanggaan*

Suatu graf dapat direpresentasikan menggunakan berbagai jenis struktur data. Matriks ketetanggaan (*adjacency matrix*) merupakan salah satu representasi graf yang memanfaatkan matriks persegi dengan ukuran yang sama dengan jumlah simpul yang ada pada graf. Elemen pada indeks i dan kolom j pada matriks ketetanggaan menyatakan bobot sisi yang menghubungkan simpul i ke simpul j [7]. Angka yang sangat besar (ataupun angka khusus lainnya) dapat digunakan untuk menyatakan tidak adanya sisi yang menghubungkan dua buah simpul. Pada graf tak berarah, matriks ketetanggaan akan berbentuk matriks simetris.

C. *Algoritma Jalur Terpendek dan Pencarian Rute*

Weighted graph (berarah maupun tidak berarah) dapat digunakan sebagai representasi peta sederhana, dengan simpul menyatakan suatu tempat dan sisi menyatakan jalan yang menghubungkan dua tempat. Dengan demikian, bobot pada suatu sisi menyatakan jarak yang harus ditempuh untuk berpindah dari suatu simpul ke simpul lainnya yang dihubungkan oleh sisi tersebut. Persoalan mencari jarak terdekat dari suatu simpul ke simpul lainnya disebut *shortest path problem* [9]. Umumnya, persoalan ini juga memerlukan informasi rute berupa urutan simpul yang perlu dilalui untuk mendapatkan jarak minimum tersebut. Hal ini dapat diselesaikan (ataupun diperoleh aproksimasinya) dengan berbagai algoritma pencarian rute yang sudah ada, seperti algoritma *Dijkstra*, *Uniform-Cost Search*, *Greedy Best First Search*, dan *A** [1, 2].

D. *Algoritma Brute Force*

Algoritma *brute force* merupakan algoritma yang bersifat terus terang dalam menyelesaikan suatu persoalan [4]. Dalam pendekatan ini, umumnya dicari semua kemungkinan solusi yang dapat dihasilkan untuk kemudian diperiksa/divalidasi satu per satu semua kandidat solusi yang telah diperoleh sebelumnya. Untuk persoalan optimasi, semua solusi yang ditemukan perlu disaring sehingga diperoleh himpunan solusi yang hanya memberikan nilai optimal. Walaupun algoritma ini dapat menjamin diperolehnya solusi yang dicari, *brute force* umumnya memiliki waktu eksekusi yang lama, terutama untuk persoalan yang kompleks dengan data yang besar.

E. *Algoritma Greedy*

Algoritma *greedy* umum digunakan dalam persoalan optimasi [5]. Algoritma ini memilih suatu langkah terbaik pada tahap tertentu dengan harapan langkah tersebut akan membawa proses pencarian atau perhitungan menuju hasil optimal. Suatu persoalan dapat diselesaikan dengan algoritma *greedy* apabila (1) solusi optimal suatu persoalan dapat ditentukan dari solusi optimal subpersoalan tersebut serta (2) terdapat pilihan langkah pada setiap subpersoalan dengan salah satu langkah menghasilkan solusi optimal untuk subpersoalan tersebut [10]. Namun, pada beberapa kasus persoalan, pendekatan *greedy* tidak menjamin diperoleh solusi optimal (secara global).

F. *Algoritma Dijkstra*

Algoritma Dijkstra merupakan algoritma pencarian rute terpendek yang memanfaatkan konsep *greedy* [10]. Pada proses pencarian, dilakukan pencatatan terhadap jarak terpendek sejauh yang sudah ditemukan untuk mencapai simpul tertentu dari simpul asal serta status suatu simpul sudah dikunjungi pada pencarian tersebut. Kelebihan algoritma ini adalah kemampuan mencari jarak terpendek dari suatu simpul ke seluruh simpul lainnya yang ada pada graf. Algoritma ini juga dapat mencatat informasi tambahan berupa simpul-simpul yang dilalui untuk mencapai setiap simpul pada graf (dari simpul asal) dengan jarak minimum tersebut.

G. *Algoritma Breadth-First Search*

Breadth-First Search (BFS) merupakan algoritma yang banyak dikenal untuk pencarian di dalam suatu graf [9]. Algoritma ini dapat melakukan pencarian di dalam graf dengan urutan pengunjungan simpul yang berdasarkan jumlah sisi yang perlu dilalui untuk mencapai simpul tersebut. Dengan demikian, algoritma BFS dapat memperoleh rute dari simpul satu ke simpul lainnya dengan melibatkan jumlah sisi minimum. Namun, algoritma ini melakukan pencarian tanpa memanfaatkan informasi bobot sisi yang dilalui sehingga rute yang diperoleh tidak dijamin optimal untuk pencarian di dalam *weighted graph*.

H. *Algoritma Uniform-Cost Search*

Algoritma *Uniform-Cost Search* (UCS) mengatasi kekurangan BFS dengan memanfaatkan nilai jarak yang telah dilalui dari simpul awal sebagai bobot simpul yang dilambangkan dengan $g(n)$ (n merupakan simpul pada graf) [1]. Algoritma ini melakukan ekspansi terhadap simpul dengan bobot terkecil hingga mencapai simpul tujuan. Dengan demikian, rute yang diperoleh ke simpul tujuan dijamin memiliki jarak minimum.

I. *Algoritma Greedy Best-First Search*

Algoritma BFS dan UCS termasuk ke dalam kategori *uninformed search* atau *blind search* akibat kurangnya informasi tambahan dalam pemilihan simpul yang lebih menjanjikan [3]. Sebaliknya, *informed search* melibatkan heuristik dalam pemilihan simpul yang akan ditelusuri selanjutnya. Dalam pencarian rute di dalam graf, heuristik yang dapat dimanfaatkan pada suatu simpul berupa estimasi jarak simpul tersebut ke

simpul tujuan dan dilambangkan dengan $h(n)$. Heuristik yang baik akan mempercepat proses pencarian rute dengan menunjukkan simpul yang lebih menjanjikan untuk mencapai tujuan dengan jarak minimum. Untuk mencapai hal ini, diperlukan heuristik yang *admissible*, yaitu tidak menaksir terlalu tinggi jarak suatu simpul ke simpul tujuan [2]. Dengan demikian, nilai heuristik suatu simpul harus lebih kecil dari jarak terpendek sebenarnya dari simpul tersebut ke simpul tujuan.

Algoritma *Greedy Best-First Search* (GBFS) merupakan salah satu algoritma pencarian graf yang melibatkan heuristik [3]. Dengan algoritma ini, penelusuran simpul akan mendahului simpul yang belum dikunjungi dengan nilai heuristik terkecil. Karena bersifat optimistik, heuristik yang kurang baik dapat mengarahkan pencarian ke simpul yang sebenarnya memiliki jarak yang lebih jauh atau bahkan tidak dapat mencapai simpul tujuan [1].

*J. Algoritma A**

Algoritma A* termasuk ke dalam algoritma pencarian yang juga memanfaatkan heuristik (*informed-search algorithm*) [1]. Namun, berbeda dengan algoritma GBFS, algoritma A* menggabungkan informasi heuristik dengan informasi jarak yang telah ditempuh dari simpul asal. Dengan demikian, bobot pada simpul dapat dilambangkan dengan $f(n)$ yang dihitung berdasarkan persamaan (1).

$$f(n) = g(n) + h(n) \quad (1)$$

III. Simulasi Permainan Labirin dalam Game Engine

Permainan labirin yang disimulasikan terdiri atas pemain dan robot yang mengejar pemain. Pemain dan robot awalnya berada di simpul yang berbeda (masing-masing pada simpul 1 dan simpul n) pada suatu peta yang direpresentasikan dengan graf berbobot dan tidak berarah. Untuk menjaga kesederhanaan implementasi simulasi permainan, pemain dibuat tidak dapat digerakkan melalui masukan *keyboard*. Hal ini juga ditujukan agar simulasi lebih berfokus pada perbandingan efisiensi algoritma, bukan pada pengembangan permainan. Namun, robot yang mengejar pemain dibuat dapat bergerak sebanyak satu simpul setiap detik mengikuti rute yang dicari oleh robot tersebut. Program kemudian akan berhenti melakukan pencarian ketika robot telah menangkap pemain. Hal ini ditujukan agar adanya acuan waktu berhentinya proses pencarian pada pengujian. Selain itu, implementasi program untuk menggerakkan robot lebih sederhana dibandingkan menggerakkan pemain berdasarkan input *keyboard* untuk simulasi permainan di dalam *game engine*. Implementasi program sepenuhnya menggunakan bahasa pemrograman Java.

A. Mekanisme Simulasi Timer pada Game Engine

Simulasi *timer* pada *game engine* memanfaatkan kelas Timer dan TimerTask pada *package* java.util. Dibuat kelas Runner yang menurunkan kelas TimerTask dan menerima objek yang menurunkan kelas PathFinder sebagai salah satu parameter konstruktor. *Interface* PathFinder mewajibkan kelas turunannya mengimplementasikan *method* findPath yang dapat menampilkan rute hasil pencarian dari simpul asal ke simpul tujuan. Selain itu, *method* ini mengembalikan nilai berupa simpul selanjutnya yang perlu dilalui tepat setelah simpul tempat robot berada.

Objek hasil instansiasi kelas Timer akan menjalankan TimerTask yang memiliki algoritma pencarian tertentu. TimerTask tersebut dijalankan secara periodik tanpa jeda awal. Periode yang digunakan adalah $20ms$, mengikuti jumlah pemanggilan *method* FixedUpdate setiap detik pada Unity. Dengan demikian, apabila lama eksekusi pencarian tidak melebihi $20ms$, program akan seolah-olah menyimulasikan permainan yang berjalan dalam 50 *frame per second* (FPS). Namun, berbeda dengan prinsip *FixedUpdate*, TimerTask pada program ini akan menunggu eksekusi pencarian sebelumnya selesai sebelum mengeksekusi pencarian berikutnya sehingga jumlah pemanggilan *method* pada TimerTask tersebut dapat berjumlah kurang dari 50 kali per detik. Hal ini terjadi apabila waktu eksekusi pencarian melebihi periode Timer yang telah disebutkan. Penurunan jumlah eksekusi pencarian per detik menyimulasikan *frame drop* pada permainan. (Mekanisme yang telah dijelaskan lebih mirip dengan mekanisme *method* Update dibandingkan *method* FixedUpdate pada Unity. Periode bawaan dalam pemanggilan *method* FixedUpdate hanya dijadikan sebagai acuan jumlah FPS.)

Selain melakukan pencarian rute, TimerTask yang dijalankan juga akan memperbarui posisi robot setiap detik. Posisi robot yang baru ditentukan oleh nilai simpul yang dikembalikan oleh pemanggilan *method* findPath. Pembaruan yang dilakukan setiap detik (bukan setiap *frame*) menyimulasikan adanya waktu yang dibutuhkan untuk berpindah posisi (walaupun dengan demikian kecepatan robot dapat berubah akibat lama perpindahan tidak memperhatikan bobot sisi graf yang dilalui). Selain perpindahan robot, TimerTask juga akan menampilkan jumlah pemanggilan *method* setiap detik sehingga diperoleh informasi jumlah FPS pada simulasi permainan tersebut.

Perlu diperhatikan bahwa tujuan utama simulasi ini adalah menunjukkan kebutuhan efisiensi algoritma dalam perhitungan logika pada *game engine*. Sebenarnya, pencarian rute tidak perlu dijalankan pada setiap pembaruan *frame* apabila pemain berada pada posisi yang tetap. Namun, apabila pemain dapat berpindah tempat, pencarian rute harus dilakukan setiap terjadi perpindahan pemain. Pada kasus terburuk, pemain akan selalu bergerak sehingga menyebabkan diperlukannnya eksekusi pencarian rute setiap pergantian *frame*. Program yang dibuat bertujuan untuk menyimulasikan kasus terburuk tersebut. Seperti yang telah dijelaskan, program tidak menerima input pergerakan pemain untuk menjaga kesederhanaan implementasi program.

B. Data Uji

Peta yang digunakan dalam pengujian berupa graf dengan n buah simpul. Sisi-sisi pada graf direpresentasikan menggunakan matriks ketetanggaan dengan setiap bobot sisi dihasilkan secara acak pada rentang 1 hingga 1000 (inklusif). Namun, sekitar sepertiga dari elemen matriks tersebut (ditambahkan elemen diagonal) bernilai -1 yang menandakan tidak adanya sisi yang

menghubungkan dua simpul tertentu (pemberian nilai -1 ini juga dilakukan secara acak). Jumlah simpul dipilih sebagai variabel dalam pengujian untuk membandingkan jenis algoritma pencarian rute berdasarkan besar data uji.

C. Pemilihan Heuristik

Heuristik yang digunakan memanfaatkan hasil pencarian jarak terpendek dari simpul target ke seluruh simpul lainnya menggunakan algoritma Dijkstra. Namun, dilakukan pengurangan dengan nilai bilangan bulat acak tertentu agar nilai heuristik tersebut tidak sepenuhnya akurat memberikan informasi jarak terpendek menuju simpul target. Bilangan acak yang digunakan dalam pengurangan tersebut berada pada rentang 0 hingga 100 (inklusif). Pengurangan yang menghasilkan bilangan negatif akan digantikan menjadi pemberian nilai nol untuk nilai heuristik simpul tersebut. Teknik pengurangan ini juga menjamin heuristik yang digunakan bersifat *admissible*.

Heuristik yang diperoleh menggunakan algoritma Dijkstra hanya bertujuan untuk memberikan contoh heuristik yang cukup baik dalam algoritma pencarian rute. Tentu penggunaan algoritma Dijkstra sudah cukup dalam menemukan jarak minimum beserta rutenya untuk mencapai simpul tujuan dari simpul asal. Dalam praktik nyata, perlu ditemukan perhitungan heuristik yang efisien dan cukup akurat dalam merepresentasikan informasi yang diinginkan. Salah satu alternatif perhitungan heuristik yang dapat digunakan adalah jarak garis lurus suatu simpul. Apabila menggunakan alternatif ini, suatu simpul perlu menyimpan informasi tambahan berupa titik atau lokasi simpul tersebut.

D. Pemilihan Algoritma

Pengujian melakukan perbandingan terhadap 4 jenis algoritma, yaitu algoritma *brute force*, UCS, GDFS, dan A*. Perbandingan dilakukan berdasarkan solusi yang dihasilkan serta waktu eksekusi pencarian. Perbandingan khusus terhadap *uninformed search* (UCS) dengan *informed search* (GDFS dan A*) juga dilakukan untuk menunjukkan pengaruh penggunaan heuristik terhadap efisiensi waktu eksekusi pencarian. Dalam hal ini, penggunaan heuristik diharapkan dapat mempercepat proses pencarian rute terpendek. Perbandingan ini didukung dengan heuristik yang cukup akurat untuk memberikan informasi jarak suatu simpul menuju simpul tujuan.

IV. Hasil Pengujian Algoritma

Pengujian dilakukan menggunakan graf acak dengan $n = 5, 10,$ dan 20 untuk n merupakan jumlah simpul pada graf tersebut. Pengujian pada setiap algoritma dilakukan dari nilai n terkecil dan pengujian untuk nilai n selanjutnya tidak dilakukan pada suatu algoritma apabila rata-rata waktu eksekusi untuk algoritma tersebut sudah melebihi periode *timer* ($20\ ms$) ataupun terjadi *error* tak terduga (selain *logic error*). Untuk setiap pengujian dengan nilai n yang telah disebutkan, dicatat rute yang diperoleh, jarak tempuh rute tersebut, waktu eksekusi pencarian, serta jumlah *frame* pada 1 detik terakhir. Berikut merupakan matriks ketetanggaan acak yang dibangkitkan untuk setiap nilai n yang digunakan pada saat pengujian. (Karena data yang cukup besar, matriks ketetanggaan untuk $n = 20$ tidak ditampilkan. Selain itu, heuristik yang dibangkitkan juga tidak ditampilkan pada makalah ini.)

```
n = 5

-1  423 939 755 488
423 -1  -1  -1  128
939 -1  -1  583 -1
755 -1  583 -1  369
488 128 -1  369 -1
```

```
n = 10
-1  -1  -1  732 555 757 -1  498 -1  -1
-1  -1  701 779 745 245 774 -1  -1  667
-1  701 -1  522 171 -1  711 385 -1  -1
732 779 522 -1  -1  -1  771 -1  10  596
555 745 171 -1  -1  994 37  -1  -1  154
757 245 -1  -1  994 -1  17  -1  -1  -1
-1  774 711 771 37  17  -1  -1  647 992
498 -1  385 -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  10  -1  -1  647 -1  -1  -1
-1  667 -1  596 154 -1  992 -1  -1  -1
```

A. Algoritma Brute Force

Berikut merupakan status eksekusi pencarian menggunakan algoritma *brute force*.

```
n = 5

1 5
Distance: 488
Execution time: 1 ms
...

1 5
Distance: 488
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

```
n = 10

1 5 10
Distance: 709
Execution time: 89 ms
...

Frame updates (in latest second): 17
5 10
```

```
Distance: 154
Execution time: 56 ms
...

5 10
Distance: 154
Execution time: 55 ms
Frame updates (in latest second): 18
Player caught!
```

B. Algoritma Uniform-Cost Search

Berikut merupakan status eksekusi pencarian menggunakan algoritma *UCS*.

```
n = 5

1 5
Distance: 488
Execution time: 2 ms
...

1 5
Distance: 488
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

```
n = 10

1 5 10
Distance: 709
Execution time: 1 ms
...

Frame updates (in latest second): 50
5 10
Distance: 154
Execution time: 0 ms
...

5 10
Distance: 154
Execution time: 0 ms
Frame updates (in latest second): 50
Player caught!
```

```
n = 20

1 18 19 17 5 20
Distance: 242
Execution time: 2 ms
...

Frame updates (in latest second): 50
18 19 17 5 20
Distance: 219
Execution time: 1 ms
...

5 20
Distance: 34
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

C. Algoritma Greedy Best-First Search

Berikut merupakan status eksekusi pencarian menggunakan algoritma *GBFS*.

```
n = 5

1 5
Distance: 488
Execution time: 1 ms
...

1 5
Distance: 488
Execution time: 0 ms
Frame updates (in latest second): 50
```

```
n = 10

1 5 10
Distance: 709
Execution time: 0 ms
...

Frame updates (in latest second): 50
5 10
Distance: 154
Execution time: 1 ms
...

5 10
Distance: 154
Execution time: 0 ms
Frame updates (in latest second): 50
Player caught!
```

```
n = 20
```

```
1 5 20
Distance: 758
Execution time: 1 ms
...

Frame updates (in latest second): 50
5 20
Distance: 34
Execution time: 0 ms
...

5 20
Distance: 34
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

*D. Algoritma A**

Berikut merupakan status eksekusi pencarian menggunakan algoritma *A**.

```
n = 5

1 5
Distance: 488
Execution time: 1 ms
1 5
...

1 5
Distance: 488
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

```
n = 10

1 5 10
Distance: 709
Execution time: 0 ms
...

Frame updates (in latest second): 50
5 10
Distance: 154
Execution time: 0 ms
...

5 10
Distance: 154
Execution time: 1 ms
Frame updates (in latest second): 50
Player catched!
```

```
n = 20

1 18 19 17 5 20
Distance: 242
Execution time: 1 ms
...

Frame updates (in latest second): 50
18 19 17 5 20
Distance: 219
Execution time: 2 ms
...

5 20
Distance: 34
Execution time: 1 ms
Frame updates (in latest second): 50
Player caught!
```

V. Analisis Perbandingan Algoritma

Pengujian pertama dengan ukuran data yang kecil ($n = 5$) menunjukkan tidak ada perbedaan waktu eksekusi untuk keempat algoritma yang dipakai. Keempat algoritma berhasil mempertahankan *frame rate* yang diatur pada program untuk ukuran data tersebut. Hal ini menunjukkan bahwa efisiensi algoritma tidak berpengaruh signifikan untuk ukuran data yang kecil. Selain itu, keempat algoritma juga menghasilkan solusi optimal untuk pengujian ini.

Pengujian kedua dengan ukuran $n = 10$ menunjukkan terjadinya *frame drop* dalam penggunaan algoritma *brute force*. Hal ini terjadi akibat waktu eksekusi pencarian yang mencapai $89\ ms$ walaupun periode *thread* diatur sebesar $20\ ms$. Hal ini menunjukkan bahwa algoritma *brute force* tidak dapat diandalkan ketika ukuran data sudah cukup membesar akibat kompleksitas waktu eksponensial dari algoritma tersebut. Sebaliknya, algoritma UCS, GBFS, dan A* masih memiliki waktu eksekusi yang kecil ($0—1\ ms$) dan tidak terlihat perbedaan signifikan pada waktu eksekusi ketiga algoritma tersebut. Selain itu, seluruh algoritma yang digunakan masih menghasilkan solusi optimal untuk pengujian kedua.

Pengujian ketiga dengan ukuran $n = 20$ menunjukkan bahwa algoritma UCS, GBFS, dan A* dapat diandalkan untuk pencarian jalur terpendek pada graf yang cukup besar. Bahkan, pengujian tambahan yang dilakukan dengan $n = 1000$ (walaupun tidak dicantumkan pada makalah ini akibat data yang sangat besar) menunjukkan bahwa algoritma UCS masih dapat melakukan pencarian dengan waktu eksekusi rata-rata di bawah $20\ ms$ (dengan beberapa fluktuasi waktu eksekusi mencapai $100\ ms$). Selain itu, waktu eksekusi pencarian menggunakan algoritma GBFS dan A* bahkan masih berada di bawah $5ms$. Hal ini menunjukkan bahwa efisiensi algoritma dalam logika

permainan berpengaruh besar terhadap kapabilitas permainan untuk mempertahankan *frame rate* seiring bertambahkan ukuran data yang harus diproses. Penggunaan heuristik juga mempercepat proses pencarian secara signifikan untuk ukuran data yang besar.

Pengujian ketiga juga menunjukkan algoritma GBFS tidak menjamin dihasilkannya solusi optimal. Pada pengujian tersebut, algoritma UCS dan A* menghasilkan solusi optimal dengan jarak tempuh rute sebesar 242 satuan. Namun, pencarian menggunakan algoritma GBFS menghasilkan rute dengan jarak tempuh sebesar 758 satuan. Hal ini disebabkan algoritma GBFS hanya memanfaatkan heuristik dan tidak mempertimbangkan jarak nyata yang perlu dilalui pada setiap sisi rute. Dengan demikian, algoritma ini tidak dapat digunakan apabila dibutuhkan akurasi pencarian rute terpendek pada permainan yang dikembangkan.

VI. Simpulan

Efisiensi algoritma memengaruhi kapabilitas *video game* dalam mempertahankan *frame rate* seiring bertambahnya ukuran data yang perlu diproses. Dalam simulasi permainan labirin yang dibuat, algoritma *brute force* hanya dapat diandalkan untuk ukuran data yang kecil. Sebaliknya, algoritma UCS, GBFS, dan A* dapat digunakan bahkan untuk ukuran data yang cukup besar. Penggunaan heuristik dapat meningkatkan efisiensi proses pencarian secara signifikan untuk ukuran data yang besar. Namun, algoritma GBFS hanya dapat digunakan apabila akurasi pencarian rute terpendek tidak terlalu dibutuhkan. Hal ini disebabkan algoritma tersebut tidak menjamin diperolehnya solusi optimal dari hasil pencarian.

Ucapan Terima Kasih

Penulis menyampaikan puji syukur kepada Tuhan Yang Maha Esa atas berkat-Nya yang memungkinkan penulis menyelesaikan makalah ini. Penulis juga berterima kasih kepada dosen pengampu mata kuliah Strategi Algoritma Semester Genap 2022/2023 kelas 02, Dr. Nur Ulfa Maulidevi, S.T, M.Sc. yang telah menyalurkan ilmu-ilmu yang digunakan dalam penulisan makalah ini. Tugas makalah ini telah menambah wawasan Strategi Algoritma yang lebih mendalam bagi penulis. Selain itu, tugas ini juga telah menambah pengalaman penulis dalam pembuatan makalah ilmiah.

References

[1] N. U. Maulidevi, "Penentuan Rute Bagian 1: BFS, DFS, UCS, Greedy Best First Search", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf, diakses pada 20 Mei 2023.

[2] N. U. Maulidevi, "Penentuan Rute Bagian 2: Algoritma A*", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf, diakses pada 20 Mei 2023.

[3] R. Munir and N.U. Maulidevi, "*Breadth/Depth First Search* (Bagian 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf, diakses pada 20 Mei 2023.

[4] R. Munir, "Algoritma *Brute Force* (Bag. 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf, diakses pada 20 Mei 2023.

[5] R. Munir, "Algoritma Greedy (Bagian 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf, diakses pada 21 Mei 2023.

[6] R. Munir, "Graf (Bag. 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, diakses pada 19 Mei 2023.

[7] R. Munir, "Graf (Bag. 2)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf, diakses pada 21 Mei 2023

[8] Technologies, U. (n.d.). Unity Scripting API, Unity, https://docs.unity3d.com/ScriptReference/.

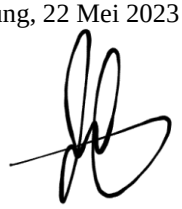
[9] Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 332—384.

[10] W. Gozali & A.F. Aji, "Pemrograman Kompetitif Dasar: Panduan Memulai OSN Informatika, ACM-ICPC, dan Sederajat", ver. 1.9, CV. Nulisbuku Jendela Dunia, pp. 128—133.

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, 22 Mei 2023



Johanes Lee 13521148