

LAPORAN TUGAS BESAR I
IF2211 STRATEGI ALGORITMA
*“Pemanfaatan Algoritma Greedy
dalam Bot Permainan Diamonds”*



Dosen:

Ir. Rila Mandala, M. Eng, Ph. D.

Monterico Adrian, S. T., M. T.

Kelompok 44:

13522131 Owen Tobias Sinurat

13522139 Attara Majesta Ayub

13522140 Yasmin Farisah Salma

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2023/2024

KATA PENGANTAR

Puji syukur kepada Tuhan Yang Maha Esa kami ucapkan atas keberhasilan dalam menyelesaikan Tugas Besar I IF2211 Strategi Algoritma yang berjudul “Pemanfaatan Algoritma Greedy dalam Pembuatan Bot Permainan Diamonds” ini.

Laporan ini merupakan dokumentasi dan penjelasan atas program dan algoritma yang telah kami implementasikan. Kami berusaha untuk merancang dan mengimplementasikan strategi *greedy* ini sedemikian rupa sehingga bot dapat mengambil keputusan yang optimal dalam permainan ini.

Pengembangan bot dan penyelesaian tugas besar ini tidak lepas dari bimbingan dan arahan yang diberikan oleh dosen pengampu mata kuliah Strategi Algoritma. Kami juga ingin mengucapkan terima kasih kepada rekan-rekan kami yang telah memberikan masukan, saran, dan inspirasi selama proses pengembangan bot ini. Selain itu, kami juga mengapresiasi dukungan yang diberikan oleh asisten dan semua pihak yang telah membantu, baik secara langsung maupun tidak langsung, dalam penyelesaian tugas besar ini. Kami berharap bahwa *output* dari tugas besar ini dapat memberikan kontribusi bagi pengembangan ilmu pengetahuan, khususnya dalam bidang algoritma dan pemrograman.

Kami menyadari bahwa tugas besar ini masih jauh dari sempurna. Oleh karena itu, kami sangat terbuka untuk menerima kritik dan saran yang konstruktif demi perbaikan di masa yang akan datang. Akhir kata, semoga tugas besar ini dapat bermanfaat bagi semua pihak yang berkepentingan.

Sumedang, 30 Februari 2024,
Kelompok 44.

DAFTAR ISI

KATA PENGANTAR.....	1
DAFTAR ISI.....	2
 BAB I	
DESKRIPSI TUGAS.....	5
1.1. Deskripsi Tugas.....	5
 BAB II	
LANDASAN TEORI.....	12
2.1 Dasar Teori Algoritma Greedy secara Umum.....	12
2.2 Cara Kerja Program Secara Umum.....	14
 BAB III	
APLIKASI STRATEGI GREEDY.....	16
3.1 Proses Mapping Persoalan Diamonds.....	16
3.1.1 Mapping Komponen Persoalan Umum Bot.....	16
3.1.2 Mapping Komponen Persoalan Pengumpulan Diamond.....	17
3.1.3 Mapping Komponen Persoalan Diamond Button.....	18
3.1.4 Mapping Komponen Persoalan Menghindari Obstacles.....	19
3.1.5 Mapping Komponen Persoalan Tackle.....	21
3.1.6 Mapping Komponen Persoalan Kembali ke Base Home.....	22
3.2 Eksplorasi Alternatif Solusi Greedy.....	23
3.2.2 Solusi Greedy Diamond Button.....	25

3.2.3 Solusi Greedy Menghindari Obstacles.....	25
3.2.4 Solusi Greedy Tackle.....	26
3.2.5 Solusi Greedy Kembali ke Base Home.....	27
3.3 Analisis Efisiensi dan Efektivitas.....	28
3.3.1. Analisis Efisiensi.....	28
3.3.2. Analisis Efektivitas.....	30
a. Strategi Pengumpulan Diamond.....	30
b. Strategi Menghindari Obstacles.....	31
c. Strategi Diamond Button.....	31
d. Strategi Menghindari Obstacles.....	32
e. Strategi Kembali ke Base Home.....	32
3.4 Strategi Greedy yang Dipilih.....	33

BAB IV

IMPLEMENTASI DAN PENGUJIAN.....	35
4.1 Implementasi Algoritma Greedy.....	35
4.1.1. Import.....	35
4.1.2. Functions.....	35
4.1.2. Procedure.....	44
4.2 Struktur Data.....	46
a) Main Method.....	47
b) Attributes.....	47
c) Other Functions.....	48
4.3 Analisis Desain Solusi Algoritma Greedy.....	54

BAB V

PENUTUP..... 62

5.1 Kesimpulan..... 62

5.2 Saran..... 62

5.3 Komentar atau Tanggapan..... 63

5.4 Refleksi terhadap Tugas Besar..... 63

5.5 Ruang Perbaikan atau Pengembangan..... 64

DAFTAR PUSTAKA..... 65

LAMPIRAN..... 66

6.1 GitHub Repository (Latest Release)..... 66

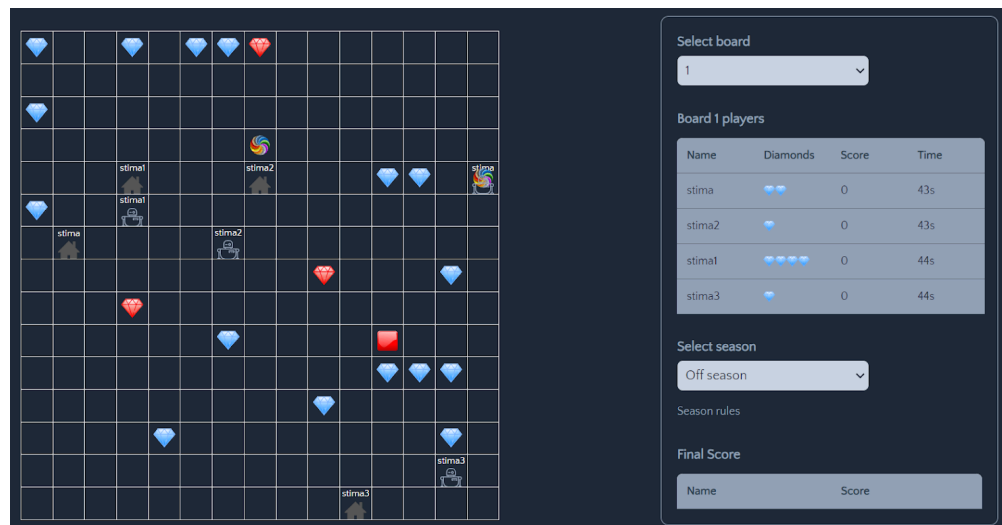
6.2 Video..... 66

BAB I

DESKRIPSI TUGAS

1.1. Deskripsi Tugas

Diamonds merupakan suatu *programming challenge* yang mempertandingkan bot yang anda buat dengan bot dari para pemain lainnya. Setiap pemain akan memiliki sebuah bot dimana tujuan dari *bot* ini adalah mengumpulkan diamond sebanyak-banyaknya. Cara mengumpulkan *diamond* tersebut tidak akan sesederhana itu, tentunya akan terdapat berbagai rintangan yang akan membuat permainan ini menjadi lebih seru dan kompleks. Untuk memenangkan pertandingan, setiap pemain harus mengimplementasikan strategi tertentu pada masing-masing botnya.



Program Permainan Diamonds terdiri atas:

a) *Game Engine*

Game engine terdiri dari dua bagian utama, yaitu *backend* dan *frontend* permainan. *Backend* memiliki peran penting dalam menentukan logika keseluruhan permainan, mencakup aturan, mekanisme permainan,

serta menyediakan API untuk komunikasi antara *frontend* dan program bot yang telah dikembangkan. Di sisi lain, *frontend* permainan bertugas untuk memvisualisasikan permainan kepada pemain. Komponen ini mengubah data dan status permainan yang kompleks menjadi representasi grafis yang mudah dimengerti, memungkinkan pemain mengikuti jalannya permainan secara intuitif. Visualisasi ini tidak hanya meningkatkan pengalaman bermain tetapi juga membantu pemain dalam merumuskan strategi mereka.

b) *Bot Starter Pack*

Bot starter pack, secara umum, terdiri dari beberapa komponen penting yang harus disusun dan dikembangkan dengan teliti untuk memastikan performa bot yang optimal. Pertama, terdapat program untuk memanggil Application Programming Interface (API) yang tersedia pada backend. Komponen ini sangat krusial karena menjadi jembatan antara bot yang kita kembangkan dengan server atau layanan eksternal, memungkinkan pertukaran data yang efisien dan efektif. Kedua, bagian inti dari pengembangan bot yaitu program *bot logic*. Pada tahap ini, para pengembang diharapkan untuk mengimplementasikan algoritma yang telah ditentukan, seperti algoritma *greedy*, untuk mengatur cara bot merespon atau bertindak dalam situasi yang berbeda-beda sesuai dengan tujuan kelompok. Implementasi yang tepat dan cermat pada bagian ini akan sangat menentukan kualitas dan kecerdasan bot dalam beroperasi. Terakhir, terdapat program utama (*main*) dan berbagai utilitas pendukung lainnya. Program utama berfungsi sebagai titik awal eksekusi bot, di mana semua komponen dan modul yang telah dikembangkan diintegrasikan dan dijalankan secara bersamaan. Utilitas pendukung lainnya, seperti fungsi-fungsi pembantu, modul untuk pengelolaan data, dan lain-lain,

memperkaya fitur bot dan meningkatkan kualitas interaksi bot dengan pengguna atau sistem lainnya.

Pengembangan bot dengan pendekatan yang sistematis dan terstruktur untuk memastikan semua komponen bekerja dengan baik secara individu dan juga secara keseluruhan adalah kunci untuk menciptakan bot yang tidak hanya berfungsi sesuai dengan harapan tetapi juga mampu memberikan pengalaman interaktif yang berkualitas bagi penggunanya.

Komponen-Komponen Permainan Diamonds:

1) **Diamonds**



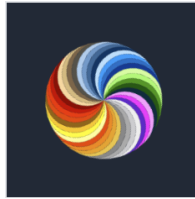
Untuk memenangkan pertandingan, kita harus mengumpulkan diamond ini sebanyak-banyaknya dengan melewati/melangkahinya. Terdapat 2 jenis diamond yaitu diamond biru dan *diamond* merah. Diamond merah bernilai 2 poin, sedangkan yang biru bernilai 1 poin. Diamond akan di-*regenerate* secara berkala dan rasio antara diamond merah dan biru ini akan berubah setiap *regeneration*.

2) **Red Button/Diamond Button**



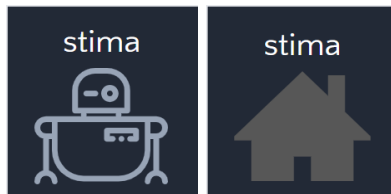
Ketika *red button* ini dilewati/dilangkahi, semua *diamond* (termasuk *red diamond*) akan di-generate kembali pada *board* dengan posisi acak. Posisi *red button* ini juga akan berubah secara acak jika *red button* ini dilangkahi.

3) Teleporters



Terdapat 2 teleporter yang saling terhubung satu sama lain. Jika bot melewati sebuah teleporter maka bot akan berpindah menuju posisi teleporter yang lain.

4) Bots and Bases



Pada game ini kita akan menggerakkan bot untuk mendapatkan diamond sebanyak banyaknya. Semua bot memiliki sebuah Base dimana Base ini akan digunakan untuk menyimpan diamond yang sedang dibawa. Apabila diamond disimpan ke base, score bot akan bertambah senilai diamond yang dibawa dan inventory (akan dijelaskan di bawah) bot menjadi kosong.

5) Inventory

Name	Diamonds	Score	Time
stima	💎💎	0	43s
stima2	💎	0	43s
stima1	💎💎💎💎	0	44s
stima3	💎	0	44s

Bot memiliki *inventory* yang berfungsi sebagai tempat penyimpanan sementara *diamond* yang telah diambil. *Inventory* ini memiliki kapasitas maksimum sehingga sewaktu waktu bisa penuh. Agar *inventory* ini tidak penuh, bot bisa menyimpan isi *inventory* ke *base* agar *inventory* bisa kosong kembali.

Cara Kerja Permainan Diamonds:

- 1) Pertama, setiap pemain (bot) akan ditempatkan pada board secara random. Masing-masing bot akan mempunyai home base, serta memiliki score dan inventory awal bernilai nol.
- 2) Setiap bot diberikan waktu untuk bergerak, waktu yang diberikan semua sama untuk setiap pemain.
- 3) Objektif utama bot adalah mengambil diamond-diamond yang ada di peta sebanyak-banyaknya. Seperti yang sudah disebutkan di atas, diamond yang berwarna merah memiliki 2 poin dan diamond yang berwarna biru memiliki 1 poin.
- 4) Setiap bot juga memiliki sebuah inventory, dimana inventory berfungsi sebagai tempat penyimpanan sementara diamond yang telah diambil. Inventory ini sewaktu-waktu bisa penuh, maka dari itu bot harus segera kembali ke home base.

- 5) Apabila bot menuju ke posisi home base, score bot akan bertambah senilai diamond yang tersimpan pada inventory dan inventory bot akan menjadi kosong kembali.
- 6) Usahakan agar bot anda tidak bertemu dengan bot lawan. Jika bot A menempa posisi bot B, bot B akan dikirim ke home base dan semua diamond pada inventory bot B akan hilang, diambil masuk ke inventory bot A (istilahnya tackle).
- 7) Selain itu, terdapat beberapa fitur tambahan seperti teleporter dan red button yang dapat digunakan apabila anda menuju posisi objek tersebut.
- 8) Apabila waktu seluruh bot telah berakhir, maka permainan berakhir. Score masing-masing pemain akan ditampilkan pada tabel Final Score di sisi kanan layar.

Mekanisme Teknis Permainan Diamonds:

Permainan ini merupakan permainan berbasis *web*, sehingga setiap aksi yang dilakukan – mulai dari mendaftarkan bot hingga menjalankan aksi bot – akan memerlukan HTTP *request* terhadap API *endpoint* tertentu yang disediakan oleh *backend*. Berikut adalah urutan *requests* yang terjadi dari awal mula permainan.

- 1) Program bot akan mengecek apakah bot sudah terdaftar atau belum, dengan mengirimkan POST *request* terhadap *endpoint* **/api/bots/recover** dengan body berisi email dan *password* bot. Jika bot sudah terdaftar, maka *backend* akan memberikan *response code* 200 dengan *body* berisi id dari bot tersebut. Jika tidak, *backend* akan memberikan *response code* 404.
- 2) Jika bot belum terdaftar, maka program bot akan mengirimkan POST request terhadap *endpoint* **/api/bots** dengan *body* berisi *email*, *name*,

password, dan *team*. Jika berhasil, maka backend akan memberikan *response code* 200 dengan *body* berisi id dari bot tersebut.

- 3) Ketika id bot sudah diketahui, bot dapat bergabung ke board dengan mengirimkan POST *request* terhadap *endpoint* `/api/bots/{id}/join` dengan *body* berisi *board* id yang diinginkan (*preferredBoardId*). Apabila bot berhasil bergabung, maka *backend* akan memberikan *response code* 200 dengan *body* berisi informasi dari *board*.
- 4) Program bot akan mengkalkulasikan *move* selanjutnya secara berkala berdasarkan kondisi board yang diketahui, dan mengirimkan POST request terhadap endpoint `/api/bots/{id}/move` dengan body berisi *direction* yang akan ditempuh selanjutnya (“NORTH”, “SOUTH”, “EAST”, atau “WEST”). Apabila berhasil, maka *backend* akan memberikan *response code* 200 dengan *body* berisi kondisi *board* setelah *move* tersebut. Langkah ini dilakukan terus-menerus hingga waktu bot habis. Jika waktu bot habis, bot secara otomatis akan dikeluarkan dari board.
- 5) Program *frontend* secara periodik juga akan mengirimkan GET *request* terhadap endpoint `/api/boards/{id}` untuk mendapatkan kondisi *board* terbaru, sehingga tampilan *board* pada *frontend* akan selalu *ter-update*.

BAB II

LANDASAN TEORI

2.1 Dasar Teori Algoritma Greedy secara Umum

Dalam pengembangan solusi komputasional untuk berbagai permasalahan, algoritma *greedy* merupakan salah satu pendekatan pemrograman yang pragmatis dan cukup efisien. Algoritma *greedy* merupakan sebuah pendekatan dalam pemecahan masalah yang dilakukan secara bertahap. Dalam setiap tahapannya, algoritma ini memilih opsi terbaik yang tersedia saat itu, berpegang pada prinsip “*take what you can get now!*” tanpa mempertimbangkan dampak jangka panjang dari keputusan tersebut. Algoritma ini beroperasi dengan harapan bahwa serangkaian pilihan optimum lokal, yaitu pilihan terbaik yang dapat dibuat di setiap langkah tanpa memikirkan masa depan, akan secara keseluruhan mengarah pada solusi optimum global, atau dengan kata lain, solusi terbaik untuk seluruh persoalan. Pendekatan ini berlandaskan pada premis bahwa pemilihan solusi yang optimal pada setiap tahapan atau sub-problema secara individual akan mengkumulasikan hasil akhir yang merupakan solusi terbaik untuk masalah secara keseluruhan. Karakteristik ini membuat algoritma *greedy* menjadi pilihan yang atraktif untuk menyelesaikan beragam masalah optimisasi, di mana efisiensi dan kecepatan dalam mencapai solusi menjadi prioritas utama.

Algoritma *greedy* memiliki dua karakteristik utama yang mendefinisikan pendekatannya dalam pemecahan masalah, yaitu sifat *myopic* dan *irrevocable*. Sifat *myopic*, atau pandangan jangka pendek, merupakan aspek penting dari algoritma ini, di mana keputusan diambil berdasarkan informasi dan situasi yang tersedia pada saat itu tanpa mempertimbangkan dampak jangka panjang dari keputusan tersebut. Di sisi lain, karakteristik

irrevocable menekankan bahwa setiap keputusan yang telah diambil oleh algoritma tidak dapat dibatalkan atau diubah. Sekali pilihan dibuat, proses tidak dapat kembali untuk merevisi atau memilih alternatif lain. Kedua sifat ini, meskipun dapat membatasi fleksibilitas algoritma, juga merupakan alasan efisiensi dan kesederhanaan dalam pemecahan berbagai masalah optimisasi dengan algoritma *greedy*.

Kelebihan utama dari algoritma *greedy* terletak pada kesederhanaannya dalam implementasi dan efisiensi waktu komputasi yang luar biasa. Pendekatan ini mampu memberikan solusi cepat dan efektif untuk berbagai jenis masalah optimasi berkat kemampuannya dalam membuat serangkaian keputusan lokal yang optimal. Dengan kelebihan ini, algoritma *greedy* sering menjadi pilihan utama ketika dibutuhkan solusi yang dapat dihasilkan dalam waktu singkat, terutama untuk masalah yang sifatnya dapat didekati dengan mempertimbangkan pilihan yang terbaik pada setiap langkah tanpa harus memproses ulang seluruh data. Namun, pendekatan *greedy* juga memiliki keterbatasan yang signifikan, terutama ketika dihadapkan pada masalah yang memerlukan pertimbangan komprehensif atas konsekuensi dari setiap keputusan di masa depan. Sifat *myopic* dari algoritma *greedy* yang hanya fokus pada pemecahan masalah secara segera tanpa mempertimbangkan dampak jangka panjang, dapat mengakibatkan kegagalan dalam mencapai solusi yang benar-benar optimal.

Jika dibandingkan dengan algoritma pemecahan masalah lainnya, seperti algoritma *dynamic programming* dan *backtracking*, algoritma *greedy* menunjukkan keunggulan dalam kecepatan dan kesederhanaan. Namun, algoritma *dynamic programming* lebih unggul dalam menangani masalah yang memerlukan pertimbangan terhadap *sub-problem* dan mengoptimalkan solusi global, sementara *backtracking* memiliki fleksibilitas dalam menjelajahi semua kemungkinan solusi hingga ditemukan yang optimal. Pilihan antara

algoritma ini seringkali didasarkan pada sifat spesifik dari masalah yang dihadapi dan *trade-off* antara kecepatan implementasi dan keakuratan solusi yang diperoleh.

2.2 Cara Kerja Program Secara Umum

Dalam pengembangan bot untuk permainan Diamonds, implementasi algoritma *greedy* menjadi inti dari proses pengambilan keputusan oleh bot dalam setiap langkah aksinya. Cara kerja program secara umum melibatkan beberapa tahapan utama yang dijalankan secara berurutan dan sistematis untuk memastikan bot dapat beroperasi secara optimal dan efisien dalam lingkungan permainan yang dinamis dan penuh tantangan.

Tahap pertama adalah inisialisasi, di mana bot dibangun dengan mengintegrasikan API yang tersedia dari *backend* permainan. Hal ini memungkinkan bot untuk berkomunikasi dan berinteraksi dengan server permainan, menerima data tentang keadaan permainan saat ini, seperti posisi *diamond*, posisi pemain lain, dan elemen-elemen penting lainnya di dalam *board* permainan.

Selanjutnya, implementasi algoritma *greedy* dalam bot dilakukan dalam pendefinisian fungsi-fungsi seleksi untuk memutuskan langkah apa yang akan diambil berdasarkan kondisi saat itu. Algoritma ini berfokus pada pengambilan keputusan yang mengoptimalkan keuntungan jangka pendek bot tanpa mempertimbangkan dampak jangka panjang. Pada setiap giliran, bot mengevaluasi semua pilihan aksi yang tersedia dan memilih aksi yang paling menguntungkan berdasarkan kriteria tertentu, seperti mendekatkan diri ke *diamond* terdekat, menghindari bot lawan, atau kembali ke *base* untuk menyimpan *diamond*.

Langkah berikutnya adalah eksekusi keputusan. Setelah bot menentukan langkah terbaik yang akan diambil, ia akan mengirimkan instruksi ke server

permainan melalui request HTTP. Server kemudian akan memproses langkah tersebut dan mengirimkan kembali respons tentang keadaan terbaru permainan setelah langkah itu dijalankan.

Proses ini diulangi secara berkelanjutan selama permainan berlangsung. Bot secara berkala menerima *update* terbaru dari server mengenai keadaan permainan dan menggunakan informasi tersebut untuk menghitung langkah selanjutnya menggunakan algoritma *greedy*. Melalui pendekatan ini, bot terus menerus beradaptasi dengan dinamika permainan dan berusaha untuk mengambil keputusan yang dapat meningkatkan peluangnya untuk memenangkan permainan.

Tahap akhir dalam pengembangan bot adalah pengujian dan penyesuaian. Berdasarkan hasil dari pertandingan-pertandingan sebelumnya dengan bot teman-teman di K-3, kami melakukan evaluasi terhadap kinerja dan strategi yang diambil oleh bot. Modifikasi dan peningkatan pada algoritma dapat dilakukan untuk mengatasi kelemahan atau memaksimalkan kekuatan bot berdasarkan analisis dari hasil pengujian tersebut. Dengan cara kerja seperti ini, bot yang dikembangkan dengan menggunakan algoritma *greedy* diharapkan dapat mengambil keputusan secara cepat dan efektif, sekaligus mempertahankan fleksibilitas untuk menyesuaikan strategi dalam menghadapi berbagai skenario permainan yang muncul.

BAB III

APLIKASI STRATEGI GREEDY

3.1 Proses *Mapping* Persoalan Diamonds

Pada permainan *Diamonds*, kemenangan dapat diraih dengan pemetaan persoalan yang akurat dan intuitif. Berikut adalah *mapping* dari persoalan-persoalan tersebut.

3.1.1 *Mapping* Komponen Persoalan Umum *Bot*

Permainan *Diamonds* ini memiliki objektif yang cukup jelas, yaitu mengumpulkan *diamonds* sebanyak mungkin di *base home*, dan pemenangnya ditentukan oleh skor terbanyak yang dihitung dari jumlah *diamond* yang dikumpulkan. Berikut *mapping* persoalan umum *bot*.

Komponen	Definisi
Himpunan kandidat	Seluruh permutasi langkah yang mungkin untuk setiap kesempatan melangkah.
Himpunan solusi	Permutasi langkah yang dapat mendapatkan skor tertinggi.
Fungsi solusi	Memeriksa apakah permutasi langkah dapat memastikan bot mendapatkan skor tertinggi.
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti.

	Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan nilai δx tidak sama dengan nilai δy .
Fungsi objektif	Mendapatkan permutasi langkah yang memenangkan permainan.

3.1.2 Mapping Komponen Persoalan Pengumpulan *Diamond*

Seluruh bot akan berusaha untuk mendapatkan *diamond* sebanyak mungkin untuk memenangkan permainan, jadi bisa dikatakan bahwa persoalan ini adalah kunci untuk memenangkan permainan. Pengumpulan *diamond* dapat dilakukan dengan banyak sekali cara dan faktor yang mempengaruhi langkah selanjutnya. Oleh karena itu, persoalan ini perlu diteliti lebih dalam jika dibandingkan dengan persoalan lain. Berikut *mapping* persoalan pengumpulan *diamond*.

Komponen	Definisi
Himpunan kandidat	Seluruh <i>diamonds</i> yang ada di <i>board</i> permainan.
Himpunan solusi	<i>Diamond</i> yang paling menguntungkan untuk diambil.
Fungsi solusi	Mengecek kapasitas tersisa di

	<i>inventory</i> karena tidak dapat menambah koleksi <i>diamond</i> apabila <i>inventory</i> penuh (kembali ke <i>base home</i> akan me-reset jumlah <i>diamond</i> di <i>inventory</i>).
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan x tidak sama dengan y.
Fungsi objektif	Mencari bobot <i>diamond</i> yang paling tinggi dengan mengukur jarak, waktu tempuh, dan poin <i>diamond</i> ..

3.1.3 Mapping Komponen Persoalan *Diamond Button*

Pada permainan Diamonds, terdapat objek permainan bernama *Diamond Button*. Objek ini, jika dilangkahi, akan me-reset *diamond-diamond* yang ada di *board* permainan termasuk *diamond button* itu sendiri. Berikut *mapping* persoalan umum *diamond button*.

Komponen	Definisi
Himpunan kandidat	Seluruh langkah yang mungkin pada posisi tertentu.
Himpunan solusi	Keputusan terkait jadi tidaknya <i>diamond button</i> dimanfaatkan.
Fungsi solusi	Mengecek syarat suatu <i>bot</i> pergi ke <i>diamond button</i> (tidak ada).
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan x tidak sama dengan y .
Fungsi objektif	Memanfaatkan <i>diamond button</i> untuk menguntungkan bot sendiri atau merugikan bot lawan.

3.1.4 Mapping Komponen Persoalan Menghindari *Obstacles*

Dalam perjalanan menuju lokasi tujuan, ada saja objek-objek yang dapat menjadi penghalang yang menyebabkan bot harus

mengerahkan langkah lebih untuk mencapai tujuannya. Objek-objek seperti *teleport* dan *bot* lawan adalah yang paling sering menjadi pemicu persoalan ini. Oleh karena itu, dibutuhkan *handler* untuk mengatasi persoalan ini.

Komponen	Definisi
Himpunan kandidat	Seluruh langkah yang mungkin pada posisi tertentu.
Himpunan solusi	Langkah yang masih mengarah ke tujuan, tetapi menghindari jalur yang akan melalui <i>obstacles</i> (tidak ada syarat untuk menghindari <i>obstacles</i>).
Fungsi solusi	Mengecek legalitas langkah yang diambil menuju lokasi.
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan x tidak sama dengan y.

Fungsi objektif	Menghindari segala <i>obstacles</i> yang ada untuk mencapai gerakan yang efektif dan efisien dalam berpindah ke lokasi tujuan.
-----------------	--

3.1.5 Mapping Komponen Persoalan *Tackle*

Dalam permainan *Diamonds*, mengumpulkan *diamonds* saja tidaklah cukup untuk dijadikan strategi utama. *Bot* perlu juga menerapkan strategi *tackle* untuk mendapatkan *diamond* dengan jumlah lebih banyak dalam waktu lebih singkat. Walaupun begitu, strategi ini juga ditemani oleh resiko yang cukup besar karena kesalahan perhitungan dapat membuat *bot* di *tackle* oleh lawan.

Komponen	Definisi
Himpunan kandidat	Seluruh langkah yang mungkin pada posisi tertentu.
Himpunan solusi	Langkah yang memastikan 100% keberhasilan <i>tackle</i> .
Fungsi solusi	Mengecek legalitas langkah yang diambil menuju lokasi (tidak ada syarat untuk melakukan <i>tackle</i>).
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti. Tingkat prioritas tersebut bersifat statik selama permainan

	berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan x tidak sama dengan y.
Fungsi objektif	Men- <i>tackle</i> <i>bot</i> lawan untuk mendapatkan <i>diamond</i> dalam waktu singkat sekaligus memperlambat penambahan skor lawan.

3.1.6 Mapping Komponen Persoalan Kembali ke *Base Home*

Pengumpulan *diamond* harus diikuti dengan perjalanan kembali ke *base home*, karena skor baru akan ditambahkan jika *diamond* sudah diantarkan ke *base home*. Volatilitas keadaan *board* permainan, membangkitkan urgensi untuk membuat kondisi-kondisi yang meng-*handle* kasus khusus.

Komponen	Definisi
Himpunan kandidat	Seluruh langkah yang mungkin pada posisi tertentu.
Himpunan solusi	Langkah yang menuju ke <i>base home</i> saat kondisi terpenuhi.
Fungsi solusi	Mengecek legalitas langkah yang diambil menuju lokasi (tidak ada

	syarat untuk pergi ke <i>base home</i>).
Fungsi seleksi	Memilih langkah berdasarkan data <i>game state</i> saat tertentu dan fungsi heuristik tingkat prioritas yang harus diikuti. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa validitas suatu langkah yaitu suatu langkah hanya boleh bernilai -1 hingga 1 dan x tidak sama dengan y.
Fungsi objektif	Menghindari segala <i>obstacles</i> yang ada untuk mencapai gerakan yang efektif dan efisien dalam berpindah ke lokasi tujuan.

3.2 Eksplorasi Alternatif Solusi Greedy

Diamonds merupakan permainan yang cukup kompleks karena banyaknya elemen yang mempengaruhi keberlangsungan suatu ronde. Banyaknya akses terhadap informasi seperti lokasi *diamond*, status robot lawan, waktu permainan tersisa, bahkan skor dari setiap robot membuka kemungkinan yang sangat luas untuk variasi solusi *greedy* itu sendiri. Berikut ini merupakan beberapa alternatif strategi solusi *greedy* yang dapat diimplementasikan pada *bot*.

3.2.1 Solusi *Greedy* Pengumpulan *Diamond*

Dari semua persoalan, mungkin persoalan inilah yang paling banyak memiliki variasi, karena persoalan inilah yang menjadi inti dari permainan *Diamonds* itu sendiri.

Solusi pertama yang dapat diimplementasikan adalah yang paling dasar, yaitu mencari *diamond* terdekat dengan posisi *bot* sekarang. Solusi ini sangat simpel, tapi bisa menjadi sangat kuat jika *diamond spawn* di daerah yang rawan *diamond* lain dan posisinya dekat dengan *base home*. Solusi ini memiliki kekurangan yaitu jika *diamond* terdekat justru jauh dengan *diamond - diamond* lainnya, yang membuat *bot* harus melangkah lebih jauh lagi untuk memenuhi kapasitas *inventory*-nya.

Solusi kedua sangat mirip dengan solusi pertama, tetapi ditambahkan satu argumen lagi berupa jumlah poin setiap *diamond* di *board*. Sehingga, dengan algoritma ini, *bot* akan memilih langkah yang mencari lokasi *diamond* yang paling “worth it” diukur dari poin dan jaraknya.

Solusi ketiga yaitu mencari *diamond* terdekat dan berada di kawasan 5x5 dengan densitas poin *diamond* tertinggi. Dengan menggunakan solusi ini, *bot* tidak perlu bergerak dari ujung ke ujung hanya untuk memenuhi kapasitas *inventory*-nya. Akan tetapi, ada *catch* pada solusi ini, yaitu solusi ini diterapkan hanya jika *inventory bot* masih kosong, sehingga setelah *bot* mendapatkan *diamond* terdekat yang berada di daerah rawan, algoritma akan kembali beralih ke solusi pertama yaitu mencari *diamond* terdekat. Dengan begitu, *robot* tidak akan terus-terusan mencari daerah rawan karena setelah pengambilan pertama, daerah rawan bisa saja berganti, yang justru bisa membuat

robot bergerak dari ujung ke ujung hanya untuk mendapatkan sedikit *diamond*.

3.2.2 Solusi Greedy Diamond Button

Diamond button akan mereset jumlah dan posisi *diamond* di *board*. Objek ini dapat dimanfaatkan untuk menguntungkan *bot* atau merugikan *bot* lawan.

Solusi pertama untuk memanfaatkan objek ini adalah untuk mengacak posisi *diamond* di akhir permainan agar *bot* yang sedang menuju suatu *diamond* tidak dapat sampai di lokasi tepat waktu. Namun, hal ini bisa menjadi bumerang bagi *bot* karena ada kemungkinan *bot* justru membantu lawan dengan mendekatkan *diamond* yang sedang dituju.

Solusi kedua adalah memanfaatkannya untuk mendekatkan *diamond* yang sudah terlalu jauh dari *bot*. Kondisi yang harus dipenuhi adalah *bot* harus berjarak lebih jauh >10 *tile* ke *diamond* terdekat daripada jarak ke *diamond button*. Solusi ini lebih efektif jika dibandingkan dengan solusi pertama karena ada kemungkinan lebih besar pengacakan *diamond* menguntungkan *bot* dan bukan sepenuhnya menguntungkan lawan.

3.2.3 Solusi Greedy Menghindari Obstacles

Perjalanan ke lokasi tujuan selalu dipakai untuk penentuan arah kemana langkah selanjutnya. Artinya, persoalan ini tidak kalah penting dengan persoalan pengumpulan *diamond*, karena persoalan ini dipakai di setiap iterasi.

Solusi pertama yaitu dengan berjalan ke arah horizontal hingga koordinat x titik *bot* sama dengan koordinat x titik tujuan, lalu

melakukan hal yang tetapi ke arah vertikal hingga koordinat y titik sama (atau sebaliknya). Solusi ini memiliki kelemahan jika terdapat *teleport* atau *bot* lawan di jalur yang akan dilalui *bot* karena dengan melangkahi salah satu objek tersebut, maka jalur ke lokasi tujuan kemungkinan besar akan membuat perjalanan menjadi lebih jauh dan *bot* bisa terkena *tackle* oleh *bot* lawan.

Solusi kedua sebenarnya mirip dengan solusi pertama, tetapi arah pergerakan bergantian tergantung posisi *obstacle* yang mungkin mengganggu perjalanan *bot* ke lokasi tujuan. Secara garis besar, jika ada *obstacle* di antara koordinat x titik awal dan koordinat x titik tujuan maka bergerak secara vertikal terlebih dahulu untuk melewati *obstacle* tersebut (sebaliknya untuk koordinat y, yaitu bergerak ke arah horizontal terlebih dahulu).

3.2.4 Solusi Greedy Tackle

Tackle adalah suatu aksi “membunuh” *bot* lawan dengan cara melangkahinya. *Diamond* milik *bot* lawan tersebut akan menjadi milik *bot* penulis dan juga *bot* lawan tersebut akan kembali ke *base home*-nya. Aksi *tackling* dapat memudahkan *bot* dalam proses pengumpulan *diamond*, hal ini dikarenakan untuk mendapatkan lebih dari satu *diamond*, *bot* hanya perlu pergi ke satu lokasi, yaitu lokasi *tackling* akan terjadi.

Solusi pertama untuk persoalan *tackle* adalah dengan mencari *bot* dengan *diamond* yang hampir memenuhi kapasitas *inventory*, lalu pergi ke *base home* dari *bot* tersebut, menunggu waktu yang tepat untuk melakukan *tackle*. Strategi ini cukup efektif karena setiap *bot* pasti akan kembali ke *base home* untuk menyimpan *diamond* yang sudah dikumpulkan. Permasalahan dari strategi ini adalah, jika *bot*

sudah sampai di *base home* lawan, tetapi *bot* lawan masih belum sampai, perlu dilakukan perhitungan untuk menentukan dari sisi mana *bot* lawan akan mendatangi *base home* dan kapan waktu perkiraannya. Perhitungan ini cukup kompleks disebabkan banyaknya kemungkinan pergerakan *bot* lawan.

Solusi kedua jauh lebih simpel dibandingkan strategi pertama, yaitu dengan memanfaatkan *diamond*. Di setiap saat, pasti ada setidaknya satu *bot* lawan yang sedang dalam perjalanan ke suatu lokasi *diamond*, hal ini dapat dimanfaatkan untuk membuat suatu jebakan. Jebakan yang dimaksud adalah penulis menjadikan *diamond* tersebut sebuah *bait* untuk melakukan *tackle*. *Bot* akan mencari lokasi *diamond* terdekat terhadap setiap *bot*, lalu mengecek apakah jarak masing-masing *bot* terhadap *diamond* terdekatnya tepat kurang 1 *tile* dibandingkan jarak *bot* penulis dengan *diamond* tersebut. Apabila kondisi terpenuhi, *bot* lawan masuk ke dalam status “*tackle-able*”, karena jika dua *bot* pergi ke lokasi yang sama, dan salah satu *bot* berjarak tepat kurang 1 *tile* dibandingkan jarak *bot* lainnya, dapat dipastikan *tackle* akan terjadi dan *bot* yang sampai terakhir di lokasi lah yang akan memenangkan *tackle* tersebut.

3.2.5 Solusi Greedy Kembali ke Base Home

Diamond yang sudah dikumpulkan perlu dipulangkan ke *base home* untuk bisa dihitung sebagai skor yang sah. Oleh karena itu, diperlukan strategi untuk menentukan kapan waktu atau keadaan yang tepat untuk kembali ke *base home*.

Solusi pertama adalah kembali ke *base home* saat *inventory* sudah memenuhi kapasitas yang ditentukan. Solusi ini cukup dasar secara logika karena hanya memerlukan satu kondisi untuk dilakukan

pengecekan. Akan tetapi, kelemahannya terletak pada kasus dimana *bot* sedang memegang beberapa *diamond* dan waktu permainan yang tersisa tidak cukup untuk *bot* kembali ke *base home*, yang membuat *diamond* yang terkumpulkan terbang sia-sia.

Pada solusi kedua, *bot* akan kembali ke *base home* dengan 2 kondisi, yaitu saat *diamond* yang ada di *inventory* sudah mencapai kapasitas maksimal **atau** saat robot memiliki *diamond* di *inventory* dan waktu permainan yang tersisa tepat cukup untuk robot kembali ke *base home*.

Kondisi yang pertama sudah sangat logis dan penulis yakin diterapkan juga di banyak bot lain, karena memang langkah terbaik adalah kembali ke *base home* saat *inventory* sudah penuh.

Kondisi kedua, robot akan kembali ke *base home* ketika robot memiliki *diamond* setidaknya 1 di dalam *inventory*-nya **dan** waktu yang tersisa tepat cukup untuk kembali ke *base home*. Tepat cukup artinya tidak ada waktu lebih untuk robot melakukan gerakan ke tujuan lain dan hanya fokus ke *base home*. Kondisi ini akan memastikan robot menyimpan semua *diamond* yang telah didapatkan selama permainan, sehingga tidak ada *diamond* yang terbang.

3.3 Analisis Efisiensi dan Efektivitas

3.3.1. Analisis Efisiensi

Banyak objek yang berada di dalam permainan *Diamonds* ini, seperti *diamond*, *diamond button*, *teleports*, dan *home base*. Dan informasi yang disediakan oleh masing-masing objek juga berbeda-beda. Hal ini membuat banyak sekali kemungkinan pemanfaatan informasi untuk membentuk algoritma yang dapat memenangkan permainan. Banyaknya informasi yang bisa dipakai tadi

mendorong suatu urgensi untuk menghitung kompleksitas algoritma agar setiap strategi yang akhirnya dipakai pada desain *bot* akhir terdiri dari strategi-strategi yang efisien.

Berikut adalah daftar tingkat efisiensi (diukur dengan kompleksitas waktu strategi menggunakan notasi Big O) masing-masing strategi:

1. Strategi Pengumpulan *Diamond*

Pada strategi ini, ada 2 kondisi yaitu saat *bot* tidak memiliki *diamond* di *inventory* atau *bot* memiliki setidaknya satu *diamond* di *inventory*. Pada kondisi pertama, *bot* akan mencari *diamond* terdekat yang terletak di daerah 5x5 dengan jumlah poin tertinggi, sehingga kompleksitas waktunya adalah $O(mn-4m-4n+16+p)$ dengan m adalah lebar *board*, n adalah panjang *board*, dan p adalah jumlah *diamond* yang terletak pada daerah 5x5 terpilih.

2. Strategi *Diamond Button*

Pada strategi ini, akan dicari lokasi *diamond button*. Karena jumlah *diamond button* hanya 1, kompleksitas waktunya adalah $O(1)$. Akan tetapi, karena strategi ini dipakai dalam suatu kondisi dimana jarak ke *diamond* terdekat lebih jauh 5 atau lebih *tiles* dibandingkan jarak ke *diamond button*, kompleksitas waktunya adalah $O(n+1)$ dengan n adalah jumlah *diamond* terdekat untuk keadaan ada minimal satu *diamond* di *inventory* dan $O(mn-4m-4n+16+p+1)$ untuk keadaan tidak ada *diamond* di *inventory* (definisi variabel sama dengan strategi di poin 1).

3. Strategi Menghindari *Obstacles*

Pada strategi ini, akan dicari lokasi *bot* lain dan semua *teleport*, sehingga kompleksitas waktunya adalah $O(m+n)$ dengan m adalah jumlah *bot* lain dan n adalah jumlah *teleport*.

4. Strategi *Tackle*

Pada strategi ini, akan dicari lokasi *bot* lain beserta *diamond* terdekatnya dan juga lokasi *diamond* terdekat dengan *bot* penulis, sehingga kompleksitas waktunya adalah $O(mn)$ dengan m adalah jumlah *bot* dalam permainan dan n adalah jumlah *diamond* di *board*.

5. Strategi Kembali ke *Base Home*

Pada strategi ini, ada dua kondisi yang dapat memenuhi yaitu *inventory* penuh dengan *diamond* atau setidaknya ada satu *diamond* di *inventory* dan waktu yang tersisa tepat cukup untuk *bot* kembali ke *base home*, sehingga kompleksitas waktunya konstan yaitu $O(1)$.

3.3.2. Analisis Efektivitas

Setiap komponen dari strategi utama memiliki efektivitas yang berbeda, berdasarkan strategi-strategi sebelumnya, berikut uraiannya.

a. Strategi Pengumpulan *Diamond*

Solusi pertama sangat simpel secara logika, tetapi tidak efektif pada beberapa kasus seperti lokasi *diamond* yang terpencil, walaupun terdekat dengan *bot* akan tetap dijadikan tujuan. Hal ini tentu tidak efektif karena *bot* harus menjalani jarak yang jauh untuk mendapatkan *diamond* selanjutnya.

Solusi kedua merupakan variasi dari solusi pertama, perbedaannya terletak pada parameter yang bertambah yaitu poin *diamond*. Solusi ini lebih efektif dari solusi pertama, tetapi masih tidak menyelesaikan permasalahan pada solusi pertama.

Solusi ketiga (yang diterapkan pada *bot*) menyelesaikan permasalahan di solusi pertama dan kedua dengan menjadikan

diamond yang berada di area 5x5 dengan densitas paling tinggi (dihitung dengan poin *diamond*). Dengan begitu, *bot* tidak perlu jalan dari ujung ke ujung untuk mengumpulkan *diamond*, karena setelah sampai di daerah rawan *diamond* tersebut *bot* bisa mendapatkan *diamond* tersisa dengan jarak yang sangat dekat.

b. Strategi Menghindari *Obstacles*

Perjalanan ke lokasi tujuan tidak dapat dilakukan hanya dengan bergerak secara horizontal lalu vertikal atau sebaliknya hingga sampai ke lokasi tujuan.

Strategi kedua (yang diterapkan pada *bot*) akan menghindari *obstacles* seperti *bot lawan* dan *teleport* yang dapat meningkatkan waktu yang dibutuhkan untuk mencapai lokasi tujuan karena *bot lawan* akan mengembalikan *bot* ke *base home* dan *teleports* akan memindahkan robot ke lokasi *teleport* lain yang bisa menjauhkan robot dari lokasi tujuan. Menghindari kedua *obstacles* tersebut akan meningkatkan efektivitas pergerakan robot agar terhindar dari gerakan yang sia-sia.

c. Strategi *Diamond Button*

Seperti yang sudah dijelaskan sebelumnya, *diamond button* akan mereset jumlah dan lokasi *diamond* yang ada di *board* jika dilangkahi.

Pada strategi pertama, robot akan pergi menuju *diamond button* untuk merugikan lawan dengan membuat *bot lawan* diharuskan mengulang kalkulasi untuk mencari *diamond* yang diincarnya.

Pada strategi kedua (yang diterapkan pada *bot*), robot akan pergi menuju *diamond button* jika tidak ada *diamond* di *inventory*

dan jarak yang harus ditempuh untuk bergerak ke lokasi *diamond* terdekat lebih jauh dibandingkan jarak yang harus ditempuh untuk ke *diamond button*. Hal ini memudahkan *bot* untuk mengumpulkan *diamond* dalam jumlah banyak karena *diamond* terkumpul dalam suatu area dengan jarak antar *diamond* sangat dekat.

d. Strategi Menghindari *Obstacles*

Strategi pertama untuk persoalan ini adalah memutar *obstacle*. Sehingga jika *obstacle* berada tepat di samping/depan/belakang *bot*, *bot* akan memutar *obstacle* tersebut untuk menghindarinya. Namun, kekurangan dari strategi ini adalah *bot* mau tidak mau harus berjalan lebih jauh dari *path* yang seharusnya dijalani, karena *bot* harus memutar *obstacle*.

Solusi kedua (yang diterapkan pada *bot*) menyelesaikan masalah dari solusi pertama dengan memprioritaskan pergerakan horizontal atau vertikal terlebih dahulu untuk menghindari *obstacle* yang ada di antara *bot* dan lokasi tujuan. Implementasi strategi ini tidak akan menambah langkah dari yang dibutuhkan untuk mencapai lokasi tujuan.

e. Strategi Kembali ke *Base Home*

Robot harus kembali ke *Base home*-nya untuk menyimpan *diamond* yang sudah didapat agar menambah skor di *leaderboard*, sehingga diperlukan beberapa pertimbangan terkait kapan waktu yang tepat untuk kembali ke *base home*.

Strategi pertama akan memerintahkan *bot* untuk pergi ke *base home* setelah *inventory* penuh. Hanya satu pengecekan yang dilakukan sehingga sangat murah secara kompleksitas waktu. Akan

tetapi, kekurangannya adalah, jika di akhir permainan *bot* tidak memiliki *inventory* penuh, *bot* tidak akan kembali ke *base home* walaupun sedang menyimpan beberapa *diamond* di *inventory*. Hal ini menyebabkan *diamond* yang sudah dikumpulkan di akhir permainan terbuang sia-sia karena tidak dipulangkan.

Strategi kedua (yang diterapkan pada *bot*) menggunakan kondisi pertama, tetapi ditambahkan kondisi baru yaitu waktu tersisa di permainan tepat cukup untuk *bot* kembali ke *base home*. Akan tetapi, kondisi baru tersebut terpenuhi jika disertai keadaan *inventory bot* kosong. Dengan begitu, *bot* akan memulangkan semua *diamond* yang dikumpulkan tanpa ada yang terbuang.

3.4 Strategi Greedy yang Dipilih

Sudah banyak *testing* yang dilakukan dengan berbagai bot lawan seperti bot *template* yang bergerak secara *random*, bot yang dibuat salah satu penulis secara individual, dan bot yang dibuat oleh kolega penulis menggunakan algoritma *greedy* pula. Setelah *testing* tersebut, ada beberapa substrategi yang dibuang dan yang bertahan.

Strategi *greedy* yang dipilih penulis adalah semua strategi yang sudah diuraikan pada sub bab 3.3 dengan kata kunci “yang diterapkan pada *bot*”. Setelah pemilihan sub strategi saja tidak cukup, untuk mengeluarkan potensi dari masing-masing sub strategi diperlukan urutan prioritas yang dapat mengaplikasikan seluruh sub strategi dengan efektif dan efisien menjadi suatu strategi *greedy* yang lengkap. Urutan prioritas ini juga dibutuhkan karena beberapa sub strategi memiliki syarat kondisi yang saling beririsan.

Penentuan urutan prioritas sub strategi ini dilakukan dengan *testing* dan evaluasi berulang terhadap beberapa urutan prioritas. Berikut adalah urutan prioritas final yang diimplementasikan pada bot:

1. Strategi *Tackle*
2. Strategi Menghindari *Obstacles*
3. Strategi Pengumpulan *Diamond*
4. Strategi *Diamond Button*
5. Strategi Kembali ke *Home Base*

Strategi *greedy* yang diimplementasikan sudah menangani sebagian besar kasus umum dan beberapa kasus khusus. Bot juga akan selalu menghasilkan gerakan valid, sehingga sudah mengurangi kemungkinan terjadinya gerakan yang “sia-sia”.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma Greedy

4.1.1. Import

```
// Import necessary libraries and modules
IMPORT random
IMPORT Optional type

// Import base logic for game
FROM game.logic.base IMPORT BaseLogic

// Import game models
FROM game.models IMPORT GameObject, Board, Position

// Import Position model
FROM current directory's models IMPORT Position
```

4.1.2. Functions

am_i_in_danger()

```
FUNCTION am_i_in_danger(bot: any, enemies: any) ->
boolean
    // Iterate through each enemy in the enemies
list
    FOR EACH enemy IN enemies:
        // Calculate the Manhattan distance between
the bot and the enemy
        IF (ABS(enemy.position.x - bot.position.x) +
ABS(enemy.position.y - bot.position.y) <= 2) THEN
            // If the distance is 2 or less, bot is
in danger
            -> True
```

```
// If none of the enemies are within the danger
distance, return False
-> False
```

is_in_between(point1: any, point2: any, point3: any)

```
FUNCTION is_in_between(point1: any, point2: any,
point3: any) -> bool
    // Returns True if point2 is between point1
and point3, inclusively
    -> point1 <= point2 AND point2 <= point3
```

get_direction(current_x: any, current_y: any, dest_x: any, dest_y: any,
board: any)

```
FUNCTION get_direction(current_x: any, current_y:
any, dest_x: any, dest_y: any, board: any) -> (int,
int)
    teleports <-
get_teleport_info(board.game_objects)
    diamondButton <-
get_diamond_button(board.game_objects)
    bots <- board.bots
    OUTPUT("bot nih bang", bots)
    horizontal <- True
    isDestDiaButton <- (dest_x ==
diamondButton.position.x AND dest_y ==
diamondButton.position.y)

    IF ((is_in_between(current_x,
diamondButton.position.x, dest_x) OR
is_in_between(dest_x, diamondButton.position.x,
current_x)) AND current_y == dest_y AND current_y ==
```

```

diamondButton.position.y AND NOT isDestDiaButton)
THEN
    IF (current_y != board.height - 1) THEN
        -> (0, 1)
    ELSE
        -> (0, -1)
    ELSE IF ((is_in_between(current_y,
diamondButton.position.y, dest_y) OR
is_in_between(dest_y, diamondButton.position.y,
current_y)) AND current_x == dest_x AND current_x ==
diamondButton.position.x AND NOT isDestDiaButton)
THEN
    IF (current_x != board.width - 1) THEN
        -> (1, 0)
    ELSE
        -> (-1, 0)

    IF ((is_in_between(current_y,
diamondButton.position.y, dest_y) OR
is_in_between(dest_y, diamondButton.position.y,
current_y)) AND (dest_x == diamondButton.position.x
OR current_y == diamondButton.position.y) AND NOT
isDestDiaButton) THEN
        horizontal <- False

    FOR EACH teleport IN teleports
        IF ((is_in_between(current_x,
teleport.position.x, dest_x) OR
is_in_between(dest_x, teleport.position.x,
current_x)) AND current_y == dest_y AND current_y ==
teleport.position.y) THEN
            IF (current_y != board.height - 1) THEN
                -> (0, 1)
            ELSE
                -> (0, -1)
        ELSE IF ((is_in_between(current_y,

```

```

teleport.position.y, dest_y) OR
is_in_between(dest_y, teleport.position.y,
current_y)) AND current_x == dest_x AND current_x ==
teleport.position.x) THEN
    IF (current_x != board.width - 1) THEN
        -> (1, 0)
    ELSE
        -> (-1, 0)

    IF (is_in_between(current_y,
teleport.position.y, dest_y) OR
is_in_between(dest_y, teleport.position.y,
current_y)) AND (dest_x == teleport.position.x OR
current_y == teleport.position.y) THEN
        OUTPUT("test gan")
        horizontal <- False

    delta_x <- clamp(dest_x - current_x, -1, 1)
    delta_y <- clamp(dest_y - current_y, -1, 1)

    IF (delta_x != 0 AND horizontal) THEN
        delta_y <- 0
    ELSE IF (delta_y != 0 AND NOT horizontal) THEN
        delta_x <- 0
    -> (delta_x, delta_y)

```

get_diamonds_info(game_objects: list)

```

FUNCTION get_diamonds_info(game_objects: list) ->
list
    -> LIST OF obj FOR EACH obj IN game_objects IF
obj.type == "DiamondGameObject"

```

get_teleport_info(game_objects: list)

```

FUNCTION get_teleport_info(game_objects: list) ->
list
    -> LIST OF obj FOR EACH obj IN game_objects IF
obj.type == "TeleportGameObject"

```

get_time_left(game_objects: List)

```

FUNCTION get_time_left(game_objects: List) -> Any
    FOR EACH item IN game_objects:
        IF (item.type == "BotGameObject") THEN
            -> item.properties.milliseconds_left

```

get_closest_diamond(pos: Position, diamonds: list)

```

FUNCTION get_closest_diamond(pos: Position,
diamonds: list) -> Diamond
    -> MIN(
        diamonds,
        key=lambda d: abs(d.position.x - pos.x) +
abs(d.position.y - pos.y),
    )

```

check_if_should_go_for_diamond_button(bot: Any, game_objects: Any, diamonds: Any)

```

FUNCTION check_if_should_go_for_diamond_button(bot:
Any, game_objects: Any, diamonds: Any) -> bool
    -> (
        get_time_to_location(bot.position,
get_closest_diamond_position(bot.position,
diamonds)) -

```



```

        get_time_to_location(bot.position,
get_diamond_button(game_objects).position)
        > 5000
    )

```

get_diamond_button(game_objects: List)

```

FUNCTION get_diamond_button(game_objects: List) ->
Item
    FOR item IN game_objects:
        IF (item.type == "DiamondButtonGameObject")
THEN
        -> item

```

get_closest_diamond_position(pos: Position, diamonds: list)

```

FUNCTION get_closest_diamond_position(pos: Position,
diamonds: list) -> Position
    -> get_closest_diamond(pos, diamonds).position

```

get_closest_blue_diamond_position(pos: Position, diamonds: list)

```

FUNCTION get_closest_blue_diamond_position(pos:
Position, diamonds: list) -> Position
    -> MIN([d FOR d IN diamonds IF
d.properties.points == 1], KEY=lambda d:
ABS(d.position.x - pos.x) + ABS(d.position.y -
pos.y)).position

```

get_closest_bot(pos: Position, bots: list)

```
FUNCTION get_closest_bot(pos: Position, bots: list)
-> (list, int)
    -> MIN(bots, KEY=lambda b: ABS(b.position.x -
pos.x) + ABS(b.position.y - pos.y))
```

get_closest_bot_to_diamond(diamond_pos: Position, bots: list)

```
FUNCTION get_closest_bot_to_diamond(diamond_pos:
Position, bots: list) -> BotGameObject
    -> MIN(bots, KEY=lambda b: ABS(b.position.x -
diamond_pos.x) + ABS(b.position.y - diamond_pos.y))
```

get_ranked_robots(bots: list)

```
FUNCTION get_ranked_robots(bots: list) -> list
    -> SORTED(bots, KEY=lambda b:
b.properties.diamonds, REVERSE=True)
```

get_rank(bots: list, bot_id: int)

```
FUNCTION get_rank(bots: list, bot_id: int) -> int
    sorted_bots <- SORTED(bots, KEY=lambda b:
b.properties.diamonds, REVERSE=True)
    FOR i, bot IN ENUMERATE(sorted_bots):
        IF (bot.id == bot_id) THEN
            -> i + 1
```

get_time_to_location(current: Position, dest: Position)

```
FUNCTION get_time_to_location(current: Position,
dest: Position) -> int
    -> 1000 * (ABS(dest.x - current.x) +
ABS(dest.y - current.y))
```

position_equals(a, b)

```
FUNCTION position_equals(a, b) -> bool
    -> a.x == b.x AND a.y == b.y
```

clamp(n, smallest, largest)

```
FUNCTION clamp(n, smallest, largest) -> int
    -> MAX(smallest, MIN(n, largest))
```

killable(my_bot: any, enemies: any, diamonds: any)

```
FUNCTION killable(my_bot: any, enemies: any,
diamonds: any) -> bool
    FOR enemy IN enemies:
        my_bot_closest_diamond <-
get_closest_diamond_position(my_bot.position,
diamonds)
        enemy_closest_diamond <-
get_closest_diamond_position(enemy.position,
diamonds)
        IF (NOT
position_equals(my_bot_closest_diamond,
enemy_closest_diamond)) THEN
            CONTINUE
        ELSE IF (get_time_to_location(my_bot.position,
my_bot_closest_diamond) -
get_time_to_location(enemy.position,
enemy_closest_diamond) == 1000) THEN
            RETURN True
    -> False
```

suicide(my_bot: any, enemies: any, diamonds: any)

```

FUNCTION suicide(my_bot: any, enemies: any,
diamonds: any) -> bool
    FOR enemy IN enemies:
        my_bot_closest_diamond <-
get_closest_diamond_position(my_bot.position,
diamonds)
        enemy_closest_diamond <-
get_closest_diamond_position(enemy.position,
diamonds)
        IF (NOT
position_equals(my_bot_closest_diamond,
enemy_closest_diamond)) THEN
            CONTINUE
        ELSE IF (get_time_to_location(my_bot.position,
enemy_closest_diamond) -
get_time_to_location(enemy.position,
my_bot_closest_diamond) == 1000) THEN
            -> True
        -> False

```

find_diamond_mine(diamonds: list, bot_position: Position)

```

FUNCTION find_diamond_mine(diamonds: list,
bot_position: Position) -> list
    max_tile_diamonds <- 0
    section <- []
    FOR y IN range(11) DO:
        FOR x IN range(11) DO:
            tile_diamonds <-
SUM(diamond.properties.points FOR diamond IN
diamonds IF y <= diamond.position.y < y + 5 AND x <=
diamond.position.x < x + 5)
            IF (tile_diamonds > max_tile_diamonds)
THEN
                section <- []

```

```

        max_tile_diamonds <- tile_diamonds
        FOR diamond IN diamonds DO:
            IF (y <= diamond.position.y < y + 5
AND x <= diamond.position.x < x + 5) THEN
                section.APPEND(diamond)
        -> section

```

get_closest_diamond_in_mine(diamonds: list, bot_position: position)

```

FUNCTION get_closest_diamond_in_mine(diamonds: list,
bot_position: position) -> any
    -> get_closest_diamond_position(bot_position,
find_diamond_mine(diamonds, bot_position))

```

4.1.2. Procedure

init()

```

CLASS Greedy EXTENDS BaseLogic
    PROCEDURE __init__()
        self.directions <- [(1, 0), (0, 1), (-1, 0),
(0, -1)]
        self.goal_position <- NULL
        self.current_direction <- 0

```

next_move()

```

PROCEDURE next_move(board_bot: GameObject, board:
Board)
    diamonds <- board.diamonds
    props <- board_bot.properties

    IF (props.diamonds == 0) THEN
        IF

```

```

(check_if_should_go_for_diamond_button(board_bot,
board.game_objects, find_diamond_mine(diamonds,
board_bot.position))) THEN
    self.goal_position <-
get_diamond_button(board.game_objects).position
    ELSE IF (NOT
(get_time_to_location(board_bot.position,
get_closest_diamond_in_mine(diamonds,
board_bot.position)) -
get_time_to_location(board_bot.position,
get_closest_diamond_position(board_bot.position,
diamonds)) > 5000)) THEN
        self.goal_position <-
get_closest_diamond_in_mine(diamonds,
board_bot.position)
    ELSE
        self.goal_position <-
get_closest_diamond_position(board_bot.position,
diamonds)
        ELSE IF (props.diamonds >= 5 OR
(props.diamonds > 0 AND
abs(get_time_to_location(board_bot.position,
board_bot.properties.base) -
get_time_left(board.game_objects)) <= 2000)) THEN
            base <- board_bot.properties.base
            self.goal_position <- base
        ELSE IF
(get_closest_diamond(board_bot.position,
diamonds).properties.points == 2 AND props.diamonds
== 4) THEN
            self.goal_position <-
get_closest_blue_diamond_position(board_bot.position
, diamonds)
        ELSE
            IF
(check_if_should_go_for_diamond_button(board_bot,

```

```

board.game_objects, diamonds)) THEN
    self.goal_position <-
get_diamond_button(board.game_objects).position
ELSE
    self.goal_position <-
get_closest_diamond_position(board_bot.position,
diamonds)

    current_position <- board_bot.position

    IF (self.goal_position) THEN
        delta_x, delta_y <-
get_direction(current_position.x,
current_position.y, self.goal_position.x,
self.goal_position.y, board)
    ELSE
        delta <-
self.directions[self.current_direction]
        delta_x <- delta[0]
        delta_y <- delta[1]
        IF random.random() > 0.6
            self.current_direction <-
(self.current_direction + 1) %
LENGTH(self.directions)

        OUTPUT("delta_x: ", delta_x, ", delta_y: ",
delta_y)
        OUTPUT("goal", self.goal_position)
        -> delta_x, delta_y

```

4.2 Struktur Data

Kami mengimplementasikan kelas **Greedy** yang merupakan pewarisan dari **BaseLogic**, bertujuan untuk menerapkan strategi algoritma

greedy dalam mengatur pergerakan bot pada permainan. Kelas ini dilengkapi dengan berbagai atribut dan metode untuk mengambil keputusan berdasarkan kondisi permainan saat itu, memanfaatkan fungsi-fungsi pendukung untuk analisis kondisi dan penentuan langkah selanjutnya.

a) Main Method

Method	Deskripsi
<u>next_move(self, board_bot: GameObject, board: Board)</u>	Method ini merupakan method utama yang berfungsi untuk mengalkulasi dan menentukan langkah selanjutnya yang harus diambil bot berdasarkan posisi bot saat ini dan kondisi <i>board</i> permainan. Method ini menganalisis kondisi sekitar, termasuk posisi <i>diamond</i> , posisi bot lain, dan elemen papan lainnya, untuk membuat keputusan strategis yang mengikuti prinsip algoritma <i>greedy</i> .

b) Attributes

Attributes	Deskripsi
<u>directions</u>	Method ini merupakan method utama yang berfungsi untuk mengalkulasi dan menentukan langkah selanjutnya yang harus diambil bot berdasarkan posisi bot saat ini dan kondisi <i>board</i> permainan. Method ini menganalisis kondisi sekitar, termasuk posisi <i>diamond</i> , posisi bot lain,

	dan elemen papan lainnya, untuk membuat keputusan strategis yang mengikuti prinsip algoritma <i>greedy</i> .
<u>goal_position</u>	Variabel ini menyimpan posisi tujuan atau target bot dalam bentuk <i>instance</i> dari <u>Optional[Position]</u> . Nilainya bisa <u>None</u> jika bot tidak memiliki tujuan spesifik pada saat itu, dan bot kemudian dapat menunggu kondisi lainnya terpenuhi.
<u>current_direction</u>	Menyimpan indeks dari <u>directions</u> yang digunakan saat bot bergerak secara roaming atau acak. Indeks ini menentukan arah spesifik bot saat tidak ada tujuan spesifik yang dikejar.

c) Other Functions

Attributes	Deskripsi
<u>is_in_between</u>	Fungsi ini menentukan apakah sebuah nilai tertentu (<u>point2</u>) berada di antara dua nilai lainnya (<u>point1</u> dan <u>point3</u>) dalam garis bilangan. Ini berguna untuk memeriksa posisi relatif dari objek-objek dalam <i>board</i> permainan, seperti apakah

	sebuah objek berada di antara bot dan tujuannya.
<u>get_direction</u>	Fungsi ini bertugas untuk menentukan arah gerak bot dari posisi saat ini menuju tujuan, dengan mempertimbangkan berbagai faktor seperti posisi <i>teleporter</i> , <i>diamond button</i> , dan halangan lainnya di papan permainan. Fungsi ini menghasilkan <i>tuple</i> yang menggambarkan pergerakan bot dalam sumbu x dan y.
<u>get_diamonds_info</u>	Fungsi ini mengumpulkan informasi tentang semua <i>diamond</i> yang ada di papan permainan. Informasi ini kemudian digunakan untuk memandu strategi pengambilan <i>diamond</i> oleh bot.
<u>get_teleport_info</u>	Fungsi ini mengumpulkan informasi tentang semua <i>teleport</i> yang ada di papan permainan. Informasi ini digunakan dalam navigasi bot.
<u>get_time_left</u>	Fungsi ini menghitung waktu yang tersisa untuk bot bergerak dalam satu ronde permainan. Informasi ini kritis dalam strategi pengambilan keputusan, terutama ketika mendekati akhir ronde, di mana bot mungkin perlu mengubah strateginya untuk

	mengoptimalkan skor.
<u>get_closest_diamond</u>	Fungsi ini bertugas untuk menemukan <i>diamond</i> terdekat dari posisi bot saat ini. Menggunakan metode perbandingan jarak, fungsi ini mengiterasi melalui semua <i>diamond</i> yang tersedia dan mengembalikan <i>diamond</i> dengan jarak terdekat ke bot.
<u>check_if_should_go_for_diamond_button</u>	Fungsi ini mengevaluasi keputusan strategis apakah bot harus menuju <i>diamond button</i> berdasarkan perbandingan waktu yang diperlukan untuk mencapai tombol tersebut dibandingkan dengan mendapatkan <i>diamond</i> terdekat. Keputusan ini bergantung pada kondisi saat itu dalam permainan, seperti lokasi <i>diamond</i> dan kondisi <i>board</i> permainan.
<u>get_diamond_button</u>	Fungsi ini mengidentifikasi lokasi <i>diamond button</i> di <i>board</i> permainan. Informasi ini penting bagi bot untuk merencanakan pergerakan, terutama ketika memutuskan untuk mengaktifkan <i>button</i> tersebut guna me-reset posisi <i>diamond</i> , yang dapat menjadi strategi untuk mengoptimalkan pengumpulan <i>diamond</i> atau mengganggu strategi lawan.

<u>get_closest_diamond_position</u>	Fungsi ini menghitung dan mengembalikan posisi dari <i>diamond</i> terdekat berdasarkan lokasi bot saat ini. Fungsi ini memungkinkan bot untuk menargetkan <i>diamond</i> terdekat secara strategis, meminimalkan waktu dan jarak pergerakan untuk mengumpulkan <i>diamond</i> .
<u>get_closest_blue_diamond_position</u>	Fungsi ini khusus mencari posisi <i>diamond</i> biru (<i>blue diamond</i>) terdekat dari posisi bot. <i>Diamond</i> biru memiliki nilai yang lebih rendah dibandingkan dengan <i>diamond</i> merah, namun strategi pengumpulan <i>diamond</i> biru dapat relevan tergantung pada situasi permainan dan kondisi <i>inventory</i> bot.
<u>get_closest_bot</u>	Fungsi ini mengidentifikasi bot lawan terdekat dari posisi bot kita berdasarkan jarak minimum.
<u>get_closest_bot_to_diamond</u>	Fungsi ini bertugas mencari bot lawan yang paling dekat dengan sebuah <i>diamond</i> tertentu. Hal ini membantu dalam strategi pengambilan <i>diamond</i> , memungkinkan bot untuk menilai apakah layak bersaing untuk <i>diamond</i> tertentu atau mencari target lain yang lebih aman.

<u>get_time_to_location</u>	Fungsi ini menghitung waktu estimasi yang dibutuhkan untuk bot bergerak dari posisi saat ini ke lokasi tujuan.
<u>position_equals</u>	Fungsi ini membandingkan dua posisi dan menentukan apakah keduanya identik. Fungsi ini berguna untuk verifikasi lokasi, seperti memastikan apakah bot telah mencapai tujuan atau dalam mengevaluasi kondisi tertentu yang tergantung pada posisi spesifik di <i>board</i> permainan.
<u>clamp(n, smallest, largest)</u>	Fungsi ini digunakan untuk membatasi nilai <i>n</i> agar berada dalam rentang tertentu, yaitu <i>smallest</i> dan <i>largest</i> . Jika <i>n</i> lebih kecil dari <i>smallest</i> , maka fungsi akan me-return <i>smallest</i> . Jika <i>n</i> lebih besar dari <i>largest</i> , fungsi akan me-return <i>largest</i> . Jika tidak, <i>n</i> akan di-return tanpa perubahan.
<u>killable(my_bot: any, enemies: any, diamonds: any)</u>	Fungsi ini mengevaluasi apakah bot pemain memiliki kesempatan untuk men-tackle bot musuh dengan memeriksa posisi relatif antara bot pemain, bot musuh, dan <i>diamond</i> terdekat. Kondisi untuk bisa men-tackle adalah jika bot musuh berada lebih dekat ke

	diamond terdekat dibandingkan bot pemain, sehingga bot pemain dapat mengambil alih posisi dan diamond dari bot musuh tersebut.
<u>suicide(my_bot:</u> <u>any,_____enemies:</u> <u>any,_____diamonds:</u> <u>any)</u>	Fungsi ini menentukan apakah bot pemain berada dalam posisi berisiko tinggi atau <i>suicide</i> yang berpotensi kehilangan <i>diamond</i> yang dikumpulkan jika bot musuh berhasil menaklukkan bot pemain. Fungsi ini melakukan pengecekan berdasarkan posisi relatif antara bot pemain, bot musuh, dan <i>diamond</i> terdekat untuk mengevaluasi risiko.
<u>find_diamond_mine</u> <u>(diamonds:___list,</u> <u>bot_position:</u> <u>Position)</u>	Fungsi ini bertujuan untuk menemukan kumpulan <i>diamond</i> yang paling banyak poinnya dalam area 5x5 di <i>board</i> permainan. Fungsi ini menghitung total poin dari <i>diamond</i> dalam setiap segmen 5x5 dan memilih segmen dengan total poin tertinggi sebagai area tujuan.
<u>get_closest_diamond_in_mine(diamonds:</u> <u>_____list,</u> <u>bot_position:</u> <u>Position)</u>	Setelah menemukan segmen 5x5 yang paling optimal melalui fungsi <u>find_diamond_mine</u> . Fungsi ini menentukan posisi <i>diamond</i> terdekat dalam segmen tersebut dari posisi bot saat ini. Fungsi ini memungkinkan bot untuk

	mengarahkan pergerakannya menuju diamond yang paling menguntungkan untuk dikumpulkan.
--	---

4.3 Analisis Desain Solusi Algoritma Greedy

Analisis desain diukur melalui performa *bot* pada pertandingan melawan *bot* lain. Pada pengujian ini, penulis menggunakan *bot* dua rekan penulis dari kelompok lain dan satu *bot random* sebagai lawan. Permainan *Diamonds* ini mengandalkan faktor keberuntungan yang cukup besar karena penempatan *home base* dan densitas *diamond* di beberapa tempat dapat menjadi keuntungan yang sangat besar kepada satu atau beberapa *bot*. Oleh karena itu, strategi yang sudah baik pun bisa kalah dari strategi yang simpel hanya karena ketidakberuntungan.

Pengujian ini dilakukan sebanyak 10 kali dengan 10 variasi *board*. Berikut adalah hasil pengujiannya.

1. Pengujian 1 Pada *Board 1*



Select board

1

Board 1 players

Name	Diamonds	Score	Time
queen		15	
anita		10	
tesla		5	
random		0	

Select season

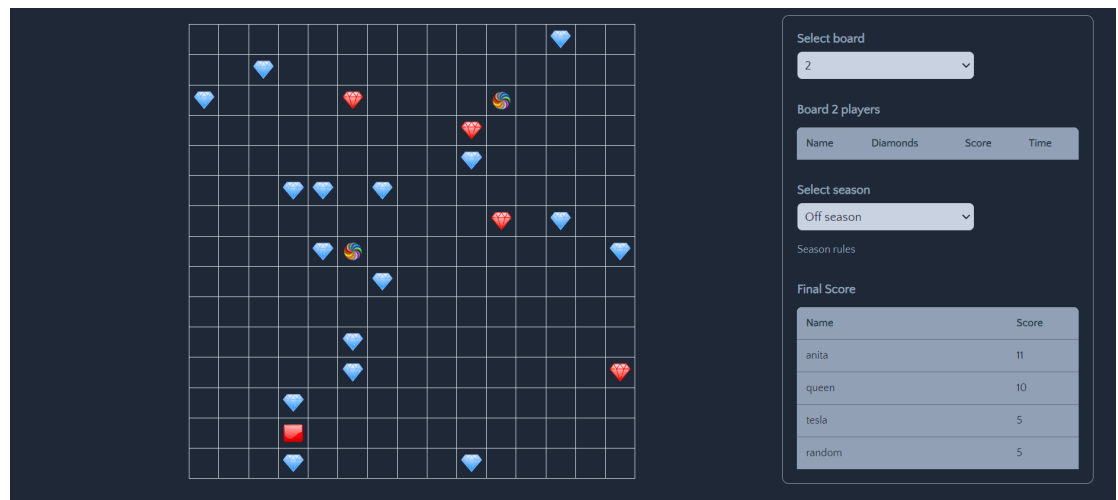
Off season

Season rules

Final Score

Name	Score
queen	15
anita	10
tesla	5
random	0

2. Pengujian 1 Pada *Board 2*



Select board

2

Board 2 players

Name	Diamonds	Score	Time
queen		10	
anita		11	
tesla		5	
random		5	

Select season

Off season

Season rules

Final Score

Name	Score
queen	10
anita	11
tesla	5
random	5

3. Pengujian 1 Pada *Board 3*



Select board

3

Board 3 players

Name	Diamonds	Score	Time
queen		14	
tesla		11	
anita		10	
random		4	

Select season

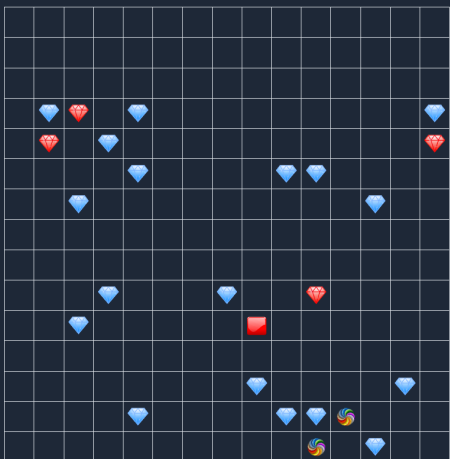
Off season

Season rules

Final Score

Name	Score
queen	14
tesla	11
anita	10
random	4

4. Pengujian 1 Pada *Board 4*



Select board

4

Board 4 players

Name	Diamonds	Score	Time
anita		13	
tesla		12	
queen		10	
random		1	

Select season

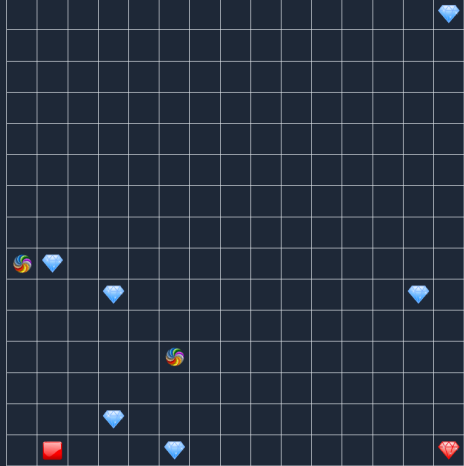
Off season

Season rules

Final Score

Name	Score
anita	13
tesla	12
queen	10
random	1

5. Pengujian 1 Pada *Board 5*



Select board

5

Board 5 players

Name	Diamonds	Score	Time
queen	10		
tesla	5		
anita	5		
random	2		

Select season

Off season

Season rules

Final Score

Name	Score
queen	10
tesla	5
anita	5
random	2

6. Pengujian 1 Pada *Board 6*



Select board

6

Board 6 players

Name	Diamonds	Score	Time
anita	15		
queen	15		
tesla	10		
random	0		

Select season

Off season

Season rules

Final Score

Name	Score
anita	15
queen	15
tesla	10
random	0

7. Pengujian 1 Pada *Board 7*



Select board

7

Board 7 players

Name	Diamonds	Score	Time
queen	18		
anita	5		
tesla	5		
random	0		

Select season

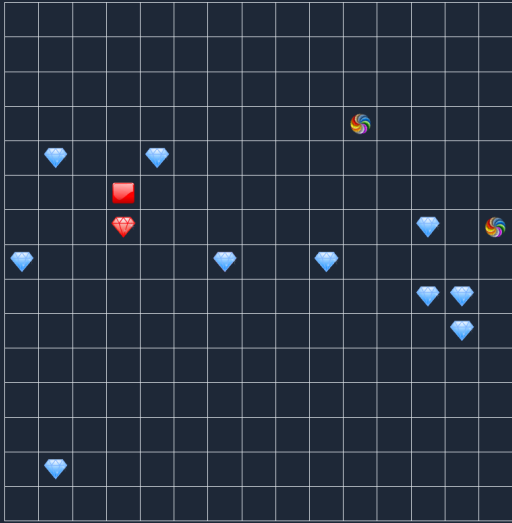
Off season

Season rules

Final Score

Name	Score
queen	18
anita	5
tesla	5
random	0

8. Pengujian 1 Pada *Board 8*



Select board

8

Board 8 players

Name	Diamonds	Score	Time
queen	12		
anita	10		
tesla	8		
random	2		

Select season

Off season

Season rules

Final Score

Name	Score
queen	12
anita	10
tesla	8
random	2

9. Pengujian 1 Pada *Board 9*

Select board

9

Board 9 players

Name	Diamonds	Score	Time
queen		15	
tesla		13	
anita		10	
random		3	

Select season

Off season

Season rules

Final Score

Name	Score
queen	15
tesla	13
anita	10
random	3

10. Pengujian 1 Pada *Board 10* (Tidak sampai benar-benar selesai karena *server* tiba-tiba terputus, tetapi pemenangnya sudah dapat dipastikan)

Select board

10

Board 10 players

Name	Diamonds	Score	Time
tesla	1	9	0s
queen		16	1s
anita	5	15	2s
random	2	1	2s

Select season

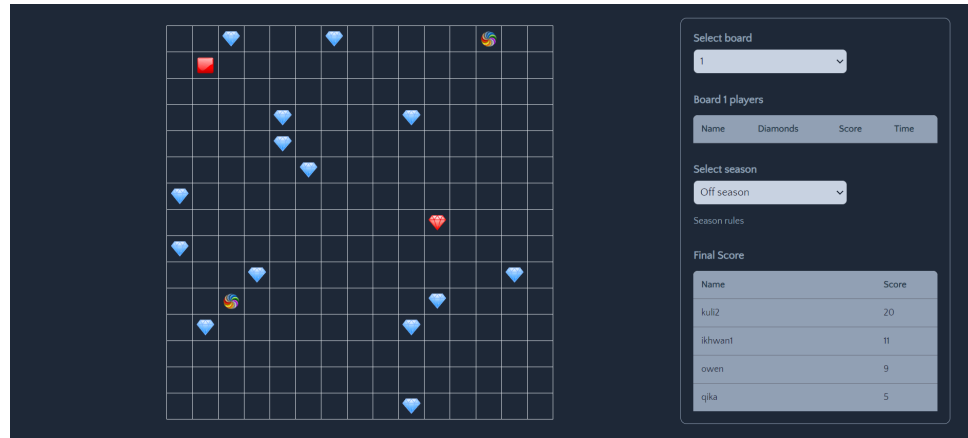
Off season

Season rules

Final Score

Name	Score
queen	16
tesla	9
anita	15
random	1

11. Bonus Pengujian yang Mencapai *High Score* (*bot* kami bernama kuli2 sebelum diganti ke queen)



Dari 10 pengujian tersebut, *bot* penulis berhasil meraih 7 kemenangan, 2 di posisi kedua, dan 1 di posisi ketiga. Hasil yang menurut penulis cukup memuaskan menghitung masih ada beberapa *case* yang belum dapat di-*cover*.

Melalui pengamatan penulis, ada beberapa pergerakan dari *bot* yang bisa dibilang kurang ideal seperti sebagai berikut.

1. Yang bertujuan melakukan *tackle* malah di-*tackle* yang disebabkan adanya *lag* atau *delay* pada server permainan.
2. Bergerak ke kanan lalu ke kiri secara berulang yang disebabkan pergantian lokasi tujuan yang terjadi dalam waktu cepat.
3. Tidak menghindari *obstacles* yang disebabkan *spawn* ulang dari *object* di *board* sehingga *obstacles* tiba-tiba muncul di *tile* yang sedang *bot* tuju.
4. Pergi menuju *diamond* yang jauh walaupun ada kumpulan *diamond* yang lebih dekat yang disebabkan *bot* lebih memprioritaskan area padat *diamond* dibandingkan beberapa *diamond* yang lebih dekat.
5. Tidak kembali ke *base home* dalam waktu yang tepat yang disebabkan pertemuan antara jarak dan waktu permainan yang tidak presisi yang mana kecil kemungkinan terjadinya.

Namun, secara garis besar, *bot* telah memberikan performa sesuai ekspektasi penulis. Beberapa kasus khusus tidak dapat di-*cover* karena keterbatasan waktu, tetapi sudah cukup banyak kasus yang telah ditangani oleh *bot* ini. Masih banyak *room for improvement* untuk *bot* ini, penulis pun merasa masih bisa dioptimalisasi lagi karena banyak sub-algoritma yang masih boros secara waktu.

BAB V

PENUTUP

5.1 Kesimpulan

Permainan *Diamonds* adalah permainan yang membutuhkan pemikiran cepat terkait langkah yang harus diambil pada giliran selanjutnya karena terbatasnya waktu. Hal ini mendorong pemakaian algoritma *greedy* yang terkenal cepat karena hanya mencari solusi yang optimal secara lokal dibandingkan algoritma pada umumnya yang mencari langkah optimal global. Formulasi algoritma *greedy* yang tepat akan membuat *bot* semakin dekat dengan solusi yang optimal secara global.

Banyak hal yang dipelajari dari tugas besar ini, seperti optimasi algoritma, pendekatan algoritma dengan menggunakan rumus fisika, dan fakta bahwa algoritma simpel kadang akan mengalahkan algoritma dengan banyak perhitungan. Penulis merasa tugas ini cukup baik dalam membantu pemahaman terkait algoritma *greedy* yang seringkali disalah artikan oleh orang banyak.

Pada tugas besar ini, penulis telah mengembangkan algoritma *greedy* untuk memenangkan permainan *Diamonds*. Setelah ratusan formulasi dan *testing*, *bot* pun akhirnya menjadi komplit dalam proses penentuan langkah lokal optimal dengan harapan menghasilkan langkah yang optimal secara global.

5.2 Saran

Untuk pengembangan *bot* Permainan *Diamonds* lebih lanjut, kami menyarankan penelitian lebih dalam pada algoritma-algoritma lain yang dapat bekerja sinergis dengan algoritma *greedy*, seperti algoritma *dynamic*

programming atau *backtracking*, untuk situasi di mana perhitungan jangka panjang menjadi penting. Kemudian, memperkaya basis data pengujian dengan skenario permainan yang lebih beragam dan kompleks akan membantu dalam meningkatkan *robustness* dan adaptabilitas bot. Integrasi teknologi kecerdasan buatan lanjutan, seperti *deep learning*, dapat dieksplorasi untuk meningkatkan kemampuan prediksi dan adaptasi bot terhadap strategi lawan yang dinamis dan tidak terduga.

5.3 Komentar atau Tanggapan

Kami terbuka untuk menerima segala jenis masukan dan saran konstruktif dari semua pihak yang berkepentingan, termasuk dosen, rekan-rekan mahasiswa, dan pihak lainnya. Kami menghargai *feedback* yang dapat memperkaya pemahaman dan membuka wawasan baru, khususnya dalam pengembangan bot dan algoritma *greedy*. Kritik dan saran yang konstruktif akan sangat berharga bagi kami untuk memperbaiki kekurangan dan memperkuat aspek-aspek bot yang sudah kami rancang.

5.4 Refleksi terhadap Tugas Besar

Tugas besar ini telah memberikan kami pelajaran tentang pentingnya aplikasi ilmu komputasi dalam menyelesaikan masalah yang nyata. Kami memahami bahwa tidak ada satu solusi yang sempurna. Oleh karena itu, eksplorasi dan inovasi terus-menerus diperlukan. Kerja sama tim, komunikasi yang efektif, dan pengelolaan waktu yang baik telah menjadi kunci dalam menyelesaikan tugas besar ini. Kami juga belajar menghadapi kegagalan sebagai bagian dari proses pembelajaran dan menggunakan kegagalan tersebut sebagai langkah untuk maju lebih jauh.

5.5 Ruang Perbaikan atau Pengembangan

Meskipun bot yang kami kembangkan telah menunjukkan kinerja yang memuaskan, masih banyak ruang untuk perbaikan. Perbaikan algoritma untuk meningkatkan kecepatan dan efisiensi dalam memproses informasi permainan sangat penting. Desain strategi yang lebih fleksibel dan adaptif terhadap perubahan kondisi permainan juga menjadi prioritas.

DAFTAR PUSTAKA

GeeksForGeeks. (2024). *Greedy Algorithms*. Retrieved from GeeksForGeeks:

<https://www.geeksforgeeks.org/greedy-algorithms/>

Munir, R. (2024). *Algoritma Greedy*. Retrieved from Homepage Rinaldi

Munir:

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf)

Sari, A. M. (2023). *Algoritma Greedy: Pengertian ,Jenis dan Contoh*

Program. Retrieved from Fakultas Ilmu Komputer dan Teknologi Informasi

UMSU:

<https://fikti.umsu.ac.id/algoritma-greedy-pengertian-jenis-dan-contoh-program/>

LAMPIRAN

6.1 GitHub Repository (Latest Release)

https://github.com/owenthe10x/Tubes1_Queen

6.2 Video

<https://www.youtube.com/watch?v=ihPMy9OSS6k>