

LAPORAN TUGAS BESAR I
IF2211 STRATEGI ALGORITMA

**PEMANFAATAN ALGORITMA *GREEDY* DALAM
PEMBUATAN BOT PERMAINAN DIAMONDS**



Kelompok Holder RNDR

Anggota :

13522021 Filbert

13522055 Benardo

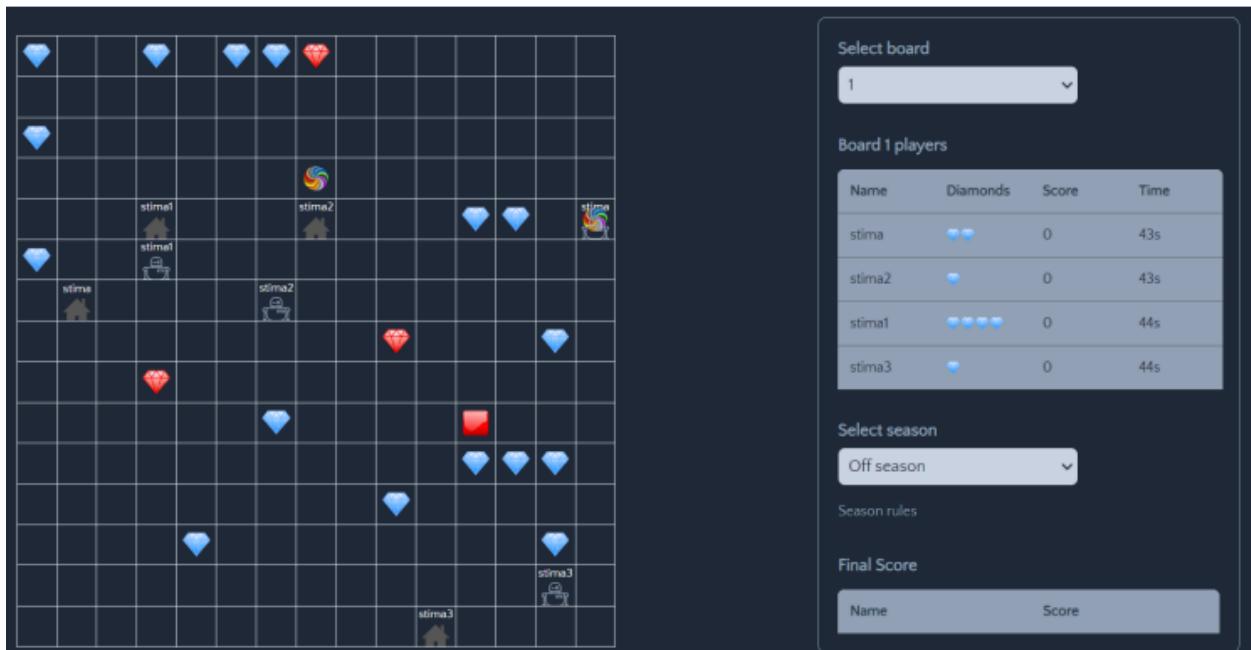
13522113 William Glory Henderson

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG 2023

BAB I

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 bot-nya. Penjelasan lebih lanjut mengenai aturan permainan akan dijelaskan di bawah.



Gambar 1.1. Tampilan permainan Diamonds yang telah dijalankan

Pada tugas pertama Strategi Algoritma ini, mahasiswa diminta untuk membuat sebuah bot yang nantinya akan dipertandingkan satu sama lain. Tentunya mahasiswa harus menggunakan strategi *greedy* dalam membuat bot ini. Program permainan *Diamonds* terdiri atas:

1. Game engine, yang secara umum berisi:

- Kode backend permainan, yang berisi logic permainan secara keseluruhan serta API yang disediakan untuk berkomunikasi dengan frontend dan program bot

- b. Kode frontend permainan, yang berfungsi untuk memvisualisasikan permainan
2. **Bot starter pack**, yang secara umum berisi:
- a. Program untuk memanggil API yang tersedia pada backend
 - b. Program bot logic (bagian ini yang akan kalian implementasikan dengan algoritma *greedy* untuk bot kelompok kalian)
 - c. Program utama (main) dan utilitas lainnya

Untuk mengimplementasikan algoritma pada bot tersebut, mahasiswa dapat menggunakan game engine dan membuat bot dari bot starter pack yang telah tersedia pada pranala berikut.

- Game engine :

<https://github.com/haziqam/tubes1-IF2211-game-engine/releases/tag/v1.1.0>

- Bot starter pack :

<https://github.com/haziqam/tubes1-IF2211-bot-starter-pack/releases/tag/v1.0.1>

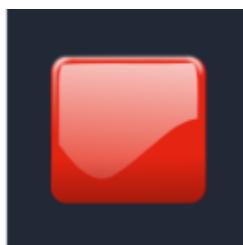
Komponen-komponen dari permainan *Diamonds* antara lain:

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.

Untuk mengetahui *flow* dari game ini, berikut ini adalah 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 menimpa 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.

BAB II

LANDASAN TEORI

2.1. Algoritma *Greedy*

Algoritma greedy merupakan metode pemecahan masalah secara bertahap(step by step), di mana pada tiap langkahnya, pilihan terbaik saat itu diambil tanpa memikirkan dampak jangka panjangnya (mengikuti prinsip ambil apa yang bisa didapat sekarang) dengan harapan bahwa serangkaian pilihan optimum yang dibuat dalam kondisi tersebut akan menghasilkan solusi terbaik secara keseluruhan atau pada akhirnya.

Terdapat elemen-elemen dalam algoritma *greedy* sebagai berikut.

- a. Himpunan kandidat, C : berisi kandidat yang akan dipilih pada setiap langkah (misal : simpul/sisi di dalam graf, job , dsb).
- b. Himpunan solusi, S : berisi kandidat yang sudah dipilih
- c. Fungsi solusi : menentukan apakah himpunan yang dipilih sudah memberikan solusi
- d. Fungsi seleksi (selection function): memilih kandidat berdasarkan strategi greedy tertentu. Strategi greedy ini bersifat heuristik.
- e. Fungsi kelayakan (feasible): memeriksa apakah kandidat yang dipilih layak atau tidak untuk dimasukkan ke dalam himpunan solusi.
- f. Fungsi obyektif: untuk memaksimumkan atau memminimumkan

Dengan menggunakan elemen-elemen tersebut, maka dapat dipastikan bahwa algoritma *greedy* melibatkan pencarian sebuah himpunan bagian S, dari himpunan kandidat C. Maka dari itu, S harus memenuhi beberapa kriteria yang telah ditentukan melalui pemeriksaan fungsi kelayakan, yaitu S menyatakan suatu solusi dan S dioptimasi dengan menggunakan fungsi objektif.

2.2. Game Engine *Diamonds*

Untuk memulai permainan dan mengintegrasikan robot ke dalam game, dibutuhkan *starter pack* permainan Diamonds. Game engine ini dirancang untuk mengoperasikan peta dan menyediakan arena bagi robot untuk berkompetisi, yang telah diintegrasikan ke dalam docker. Oleh karena itu, hanya perlu mengaktifkan lingkungan virtual yang ada di dalam docker untuk menjalankan mesin permainan Diamonds. Karena permainan ini berbasis web, perlu untuk menjalankan antarmuka pengguna Frontend dan juga logika permainan yang terdapat dalam backend yang termasuk dalam *starter pack* bot. Untuk memulai permainan dan mengimplementasikan bot, diperlukan *starter pack* dari game Diamonds, yang dapat diunduh melalui tautan berikut ini:

<https://github.com/haziqam/tubes1-IF2211-game-engine/releases/tag/v1.1.0>.

Ada beberapa komponen penting yang perlu diketahui untuk memahami cara kerja game, yakni:

1. Game Engine: Game Engine merupakan komponen utama dari game Diamonds yang bertanggung jawab atas berbagai aspek penting dalam permainan. Ini termasuk pengendalian logika permainan yang kompleks, pengaturan fisika untuk simulasi gerakan dan interaksi objek, pengelolaan grafis untuk menghasilkan tampilan visual yang menarik, serta mengatur interaksi antara pemain dan elemen-elemen permainan lainnya.
2. Backend: Backend adalah bagian dari game engine yang berfokus pada pemrosesan data dan logika permainan di sisi server. Ini termasuk pengelolaan basis data yang menyimpan informasi tentang pemain, skor, dan status permainan. Backend juga bertanggung jawab atas pemrosesan perintah yang dikirim oleh bot, seperti gerakan atau aksi dalam game, serta menjalankan logika permainan yang tidak langsung terkait dengan aspek visual, seperti perhitungan skor atau penentuan pemenang.
3. Frontend: Frontend adalah bagian dari game engine yang berinteraksi langsung dengan pemain. Ini mencakup semua elemen visual dan antarmuka pengguna seperti rendering grafis , animasi , menu, papan skor, diamond ,papan permainan, dan lain-lain.

4. Karakter dan Objek: Dalam game Diamonds, karakter dan objek adalah entitas penting yang dapat bergerak dan berinteraksi satu sama lain. Karakter biasanya merupakan representasi pemain dalam permainan, seperti robot yang dapat bergerak untuk mengumpulkan diamond. Objek lain, seperti diamond, base, atau teleporter, memiliki peran dalam dinamika permainan, mempengaruhi strategi dan jalannya permainan.
5. Algoritma: Algoritma dalam game Diamonds adalah kumpulan aturan dan instruksi yang digunakan untuk mengatur perilaku dalam permainan. Ini termasuk logika untuk menentukan bagaimana karakter bergerak, bagaimana interaksi antar objek terjadi, serta implementasi kecerdasan buatan untuk bot atau lawan dalam game. Algoritma juga mencakup strategi permainan, seperti cara terbaik untuk mengumpulkan diamond atau menghindari hambatan.
6. Paket Starter: Paket starter adalah sumber daya yang disediakan untuk memudahkan pengembangan atau penggunaan bot dalam game Diamonds. Ini biasanya mencakup kode sumber dasar yang dapat dimodifikasi atau diperluas oleh pengembang, dokumen yang menjelaskan cara kerja game dan API yang tersedia, serta alat-alat pendukung untuk mengaktifkan dan mengelola bot dalam lingkungan permainan.

Berikut adalah garis besar cara kerja program game Diamonds:

1. Inisialisasi Pertandingan: Pada saat awal game engine dijalankan, pertandingan akan dimulai dengan meng-host permainan pada sebuah hostname yang telah ditentukan. Jika program dijalankan di local, engine akan di-host pada localhost:8082
2. Koneksi antara Engine dan Bot: Setelah berhasil menjalankan engine, engine akan melakukan pendaftaran koneksi dengan bot-bot yang dijalankan. Engine menunggu semua bot terkoneksi, lalu melakukan proses logging.
3. Persiapan Jumlah Bot: Jumlah bot yang akan bermain dalam satu pertandingan diatur oleh atribut BotCount yang ada dalam file "appsettings.json" di dalam folder "engine-publish".

4. Memulai Pertandingan: Begitu semua bot pemain terkoneksi dan jumlahnya sesuai dengan *BotCount* yang ditetapkan, pertandingan akan dimulai.
5. Interaksi antara Bot dan Engine: Setelah terkoneksi dengan game engine, bot-bot akan melakukan pengiriman event-event kepada engine , lalu engine juga akan mengirimkan event-event kepada bot.
6. Pelaksanaan Pertandingan: Pertandingan akan berlangsung sampai waktu habis. Pada saat permainan , akan diperlihatkan papan skor yang berupa diamond yang telah berhasil dikumpulkan dan juga log-log yang akan dicatat di file JSON.

Dengan cara kerja seperti ini, program Diamonds mengatur interaksi antara engine dan bot pemain untuk menyelenggarakan pertandingan Diamonds dan mencatat hasilnya.

2.3. Alur Menjalankan Game *Diamonds*

Berikut adalah langkah-langkah untuk menjalankan program:

1. Pastikan requirement yang dibutuhkan telah diinstall seperti Node js, Docker Desktop dan Yarn. Untuk penginstalan yarn dapat mengikuti perintah berikut.

```
npm install --global yarn
```

2. Jalankan game engine dengan cara mengunduh starter pack game engine dalam bentuk file .zip yang terdapat pada tautan yang telah dilampirkan pada bagian 2.2
 - a. Setelah melakukan instalasi, lakukan ekstraksi file .zip tersebut lalu masuk ke root folder dari hasil ekstraksi file tersebut kemudian buka terminal
 - b. Jalankan perintah berikut pada terminal untuk masuk ke root directory dari game engine

```
cd tubes1-IF2110-game-engine-1.1.0
```

- c. Lakukan instalasi dependencies dengan menggunakan yarn.

```
yarn
```

- d. Lakukan setup environment variable dengan menjalankan script berikut untuk OS Windows

```
./scripts/copy-env.bat
```

Untuk Linux / (possibly) macOS

```
chmod +x ./scripts/copy-env.sh  
./scripts/copy-env.sh
```

- e. Lakukan setup local database dengan membuka aplikasi docker desktop terlebih dahulu kemudian jalankan perintah berikut di terminal

```
docker compose up -d database
```

Kemudian jalankan script berikut. Untuk Windows

```
./scripts/setup-db-prisma.bat
```

Untuk Linux / (possibly) macOS

```
chmod +x ./scripts/setup-db-prisma.sh  
./scripts/setup-db-prisma.sh
```

- f. Jalankan perintah berikut untuk melakukan proses build game-engine

```
npm run build
```

- g. Jalankan perintah berikut untuk memulai game-engine

```
npm run start
```

- h. Jika berhasil, tampilan terminal akan terlihat seperti gambar di bawah ini.

```
[0] [nodemon] to restart at any time, enter `rs`
[0] [nodemon] watching dir(s): dist/**/* .env
[0] [nodemon] watching extensions: ts,map,js,json
[0] [nodemon] starting `nest start`
[0] [Nest] 3476 - 02/15/2024, 10:39:58 PM LOG [NestFactory] Starting Nest application...
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [InstanceLoader] AppModule dependencies initialized +106ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] BoardsController {/api/boards}: +100ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/boards, GET} route +3ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/boards/:id, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] BotsController {/api/bots}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/bots/:id, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/bots, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/bots/recover, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/bots/:id/join, POST} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/bots/:id/move, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] HighscoresController {/api/highscores}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/highscores/:seasonId, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] RecordingsController {/api/recordings}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/recordings/seasons/:seasonId, GET} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/recordings/:id, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] SeasonsController {/api/seasons}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/seasons, GET} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/seasons/current, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/seasons/:id, GET} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/seasons/:id/rules, GET} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] SlackController {/api/slack}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/slack/seasons, POST} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/slack/season, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/slack/teams, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/slack/team, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/slack/interact, POST} route +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RoutesResolver] TeamsController {/api/teams}: +0ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [RouterExplorer] Mapped {/api/teams, GET} route +1ms
[0] [Nest] 3476 - 02/15/2024, 10:39:59 PM LOG [NestApplication] Nest application successfully started +50ms
```

3. Jalankan bot starter pack dengan cara mengunduh kit dengan ekstensi .zip yang terdapat pada tautan yang telah dilampirkan pada bab 1.
 - a. Lakukan ekstraksi file zip tersebut, kemudian masuk ke folder hasil ekstrak tersebut dan buka terminal
 - b. Jalankan perintah berikut untuk masuk ke root directory dari project

```
cd tubes1-IF2110-bot-starter-pack-1.0.1
```

- c. Jalankan perintah berikut untuk menginstall dependencies dengan menggunakan pip

```
pip install -r requirements.txt
```

- d. Jalankan program dengan cara menjalankan perintah berikuts.

```
python main.py --logic MyBot --email=your_email@example.com
--name=your_name --password=your_password --team etimo
```

Anda juga bisa menjalankan satu bot saja atau beberapa bot menggunakan .bat atau .sh script.

- Untuk windows

```
./run-bots.bat
```

- Untuk Linux / (possibly) macOS

```
./run-bots.sh
```

Kalian dapat menyesuaikan *script* yang ada pada run-bots.bat atau run-bots.sh dari segi **logic yang digunakan, email, nama, dan password**

```
run-bots.bat
1  @echo off
2  start cmd /c "python main.py --logic Random --email=test@email.com --name=stima --password=123456 --team etimo"
3  start cmd /c "python main.py --logic Random --email=test1@email.com --name=stima1 --password=123456 --team etimo"
4  start cmd /c "python main.py --logic Random --email=test2@email.com --name=stima2 --password=123456 --team etimo"
5  start cmd /c "python main.py --logic Random --email=test3@email.com --name=stima3 --password=123456 --team etimo"
6
```

- Untuk mengimplementasikan bot, buatlah folder baru pada direktori /game/logic (misalnya mybot.py)
- Buatlah kelas yang meng-inherit kelas *BaseLogic*, lalu implementasikan constructor dan method *next_move* pada kelas tersebut

```
mybot.py U X
game > logic > mybot.py > ...
1  from game.logic.base import BaseLogic
2  from game.models import Board, GameObject
3
4
5  class MyBot(BaseLogic):
6      def __init__(self):
7          # Initialize attributes necessary
8          self.my_attribute = 0
9
10     def next_move(self, board_bot: GameObject, board: Board):
11         # Calculate next move
12         delta_x = 1
13         delta_y = 0
14         return delta_x, delta_y
15
```

- Import kelas yang telah dibuat pada main.py dan daftarkan pada dictionary *CONTROLLERS*

```
main.py M X
main.py > ...
1 import argparse
2 from time import sleep
3
4 from colorama import Back, Fore, Style, init
5 from game.api import Api
6 from game.board_handler import BoardHandler
7 from game.bot_handler import BotHandler
8 from game.logic.random import RandomLogic
9 from game.util import *
10 from game.logic.base import BaseLogic
11 from game.logic.mybot import MyBot ←
12
13 init()
14 BASE_URL = "http://localhost:3000/api"
15 DEFAULT_BOARD_ID = 1
16 CONTROLLERS = {"Random": RandomLogic, "MyBot": MyBot} ↓
```

BAB III

Aplikasi Strategi Greedy

3.1. Alternatif *Greedy*

3.1.1. Alternatif *Greedy* by *Shortest Bot Distance*

Greedy by Shortest Bot Distance adalah strategi *greedy* yang mengutamakan jarak terdekat dari bot ke *diamond*. *Diamond* terdekat tidak dipedulikan jumlah poinnya (warna biru atau merah) karena bot hanya mempertimbangkan jarak *diamond* yang terdekat dengan dirinya. Untuk menentukan jarak terdekat antara bot dengan *diamond* dan bot dengan *base*. Jika terdapat *diamond* warna merah dan biru dengan jarak yang sama maka akan mengutamakan *diamond* merah jika total *diamond point* dalam *inventory bot* belum mencapai 4. Algoritma ini juga memperhatikan jarak teleporter ke *diamond* dan ke *base*.

1. Pemetaan Elemen *Greedy*

- a. Himpunan Kandidat: Himpunan *diamond* yang tersedia pada board, yang meliputi *diamond* merah yang bernilai 2 poin dan *diamond* biru yang bernilai 1 poin.
- b. Himpunan Solusi: Himpunan *diamond* yang terpilih.
- c. Fungsi Solusi: Menjumlahkan seluruh poin *diamond* yang telah didapatkan.
- d. Fungsi Seleksi: Memilih *diamond* yang berjarak terdekat dari *bot* tanpa melihat poin dan mengutamakan poin yang lebih besar jika jaraknya sama.
- e. Fungsi kelayakan: Memeriksa jumlah *diamond* yang telah diperoleh tidak melebihi 5 pada setiap iterasi, dan apabila jumlah *diamond* telah mencapai 3-5, *bot* akan diarahkan untuk kembali ke *base* untuk menyimpan *diamond*.
- f. Fungsi obyektif: Jumlah poin *diamond* yang diperoleh maksimum.

2. Analisis Efisiensi Solusi

Pada alternatif ini, akan dilakukan inisialisasi terhadap suatu *array* kosong yang nantinya menampung keseluruhan jarak dan posisi dari keseluruhan *diamond* yang tersedia pada *board*. Kemudian, akan dilakukan iterasi untuk setiap *diamond* yang tersedia pada *board* untuk memasukkan jarak dari *diamond* tersebut pada *array*. Setelah *array* terdefinisi, akan dilakukan sorted dengan fungsi bawaan pada python berdasarkan jarak diamond terdekat dengan koordinat posisi bot pada waktu yang bersangkutan. Kompleksitas algoritma untuk memasukkan jarak dan posisi dari seluruh diamond ke dalam array adalah $O(n)$. Untuk kompleksitas algoritma pengurutan atau *sorting* pada fungsi bawaan python adalah $O(n \log(n))$ sehingga kompleksitas dari keseluruhan algoritmanya adalah $O(n \log(n))$.

3. Analisis Efektivitas Solusi

Strategi ini mengutamakan jarak terdekat (*shortest distance*) dari posisi *bot* dengan *diamond*.

Strategi ini menjadi efektif apabila:

- a. *Bot spawn* dekat pada daerah yang memiliki jumlah *diamond* yang banyak. Karena hanya mempertimbangkan *shortest distance*, maka *bot* dapat mengambil *diamond* dengan cepat dan kembali ke *base* dengan cepat juga.
- b. *Bot* tidak *spawn* di dekat *bot* lainnya, sehingga tidak rentan untuk ditabrak oleh *bot* lainnya.
- c. Terdapat teleporter yang membantu *bot* untuk menuju ke daerah yang mengandung *diamond* dan balik ke *base*.

Strategi ini menjadi tidak efektif apabila:

- a. *Bot spawn* jauh dari *diamond* dan posisi *base* yang jauh juga.
- b. Jarak *bot* dengan *bot* lawan sangatlah dekat sehingga menyebabkan terjadi *collision* / tabrakan.

- c. Terdapat teleporter yang menghalangi jalan *bot* sehingga terdapat dua gerakan tambahan yang harus dilakukan untuk kembali ke jalur pengambilan *diamond* atau balik ke *base*.

3.1.2. Alternatif *Greedy by Highest Density*

Greedy by Highest Density adalah strategi *greedy* yang mengutamakan area dengan konsentrasi *diamond* yang paling tinggi tanpa mempertimbangkan jarak. Dalam strategi *greedy* ini, *bot* akan cenderung menuju ke area di mana terdapat kelompok *diamond* terbanyak dalam radius tertentu. Hal ini berbeda dengan *Greedy by Shortest Distance* yang mengutamakan *diamond* apapun yang berada dekatnya. Algoritma ini juga memperhatikan jarak teleporter ke *diamond* dan ke *base*.

1. Pemetaan Elemen *Greedy*

- a. Himpunan Kandidat: Himpunan kelompok *diamond* yang tersedia pada *board*, yang didefinisikan berdasarkan kepadatan *diamond* dalam radius tertentu.
- b. Himpunan Solusi: Himpunan kelompok *diamond* yang terpilih.
- c. Fungsi Solusi: Menjumlahkan seluruh poin *diamond* yang telah didapatkan dari kelompok *diamond* yang dipilih.
- d. Fungsi Seleksi: Memilih kelompok *diamond* dengan kepadatan (jumlah *diamond* dalam radius tertentu) terbesar pada *board*.
- e. Fungsi kelayakan: Memeriksa jumlah *diamond* yang telah diperoleh tidak melebihi 5 pada setiap iterasi, dan apabila jumlah *diamond* telah mencapai 5, *bot* akan diarahkan untuk kembali ke *base* untuk menyimpan *diamond*.
- f. Fungsi obyektif: Jumlah poin *diamond* yang diperoleh maksimum dengan mempertimbangkan kepadatan *diamond* di suatu area.

2. Analisis Efisiensi Solusi

Dalam alternatif ini, *bot* akan melakukan clustering atau pengelompokan daerah terhadap *diamond* yang tersedia pada *board* berdasarkan kepadatan

tertinggi atau jumlah *diamond* terbanyak dalam radius tertentu. Kemudian, bot akan mengevaluasi setiap daerah untuk menentukan daerah dengan kepadatan tertinggi. Setelah daerah ditemukan, bot akan menentukan posisi dan menuju ke titik tengah dari daerah tersebut sebagai target berikutnya. Kompleksitas algoritma untuk memasukkan jarak dan posisi dari seluruh diamond ke dalam array adalah $O(n)$. Untuk kompleksitas algoritma pengurutan (sorting) pada fungsi bawaan *python* adalah $O(n \log(n))$. Untuk kompleksitas penentuan daerah dengan kepadatan adalah $O(n^2)$ sehingga kompleksitas dari keseluruhan algoritmanya adalah $O(n^2)$.

3. Analisis Efektivitas Solusi

Strategi ini mengutamakan bot untuk bergerak menuju area dengan kepadatan diamond tertinggi.

Strategi ini menjadi efektif apabila:

- a. *Bot* melakukan *spawn* di titik tengah daerah atau di area yang dekat dengan daerah *diamond* yang paling besar.
- b. Papan permainan memiliki distribusi diamond yang tidak merata, sehingga terdapat area dengan kepadatan diamond yang tinggi.
- c. *Bot* tidak *spawn* di dekat *bot* lainnya, sehingga tidak rentan untuk ditabrak oleh *bot* lainnya.
- d. Terdapat teleporter yang membantu *bot* untuk menuju ke daerah yang *mengandung* diamond dan balik ke *base*.

Strategi ini menjadi tidak efektif apabila:

- a. Distribusi *diamond* di papan permainan cukup merata, sehingga tidak terdapat area dengan kepadatan *diamond* yang signifikan.
- b. *Bot* terlalu fokus pada daerah dengan kepadatan tertinggi tanpa mempertimbangkan jarak sehingga dapat menghabiskan waktu yang lebih lama untuk mencapai target.
- c. *Bot spawn* jauh dari *diamond* dan posisi *base* yang jauh juga.

- d. Terdapat teleporter yang menghalangi jalan *bot* sehingga terdapat dua gerakan tambahan yang harus dilakukan untuk kembali ke jalur pengambilan *diamond* atau balik ke *base*.

3.1.3. Alternatif *Greedy by Nearest Base*

Greedy by Nearest Base adalah strategi *greedy* yang mengutamakan pengembalian *diamond* ke *base* sesegera mungkin setelah mencapai jumlah tertentu atau ketika waktu tersisa sudah sedikit. Dalam strategi ini, *bot* akan mengarahkan pergerakannya ke *base* ketika telah mengumpulkan 4 atau lebih *diamond* atau ketika waktu yang tersisa kurang dari nilai tertentu. Algoritma ini juga memperhatikan jarak teleporter ke *diamond* dan ke *base* serta memanfaatkan pencarian *red button* untuk mereset *diamond* jika jarak *diamond* terlalu jauh.

1. Pemetaan *greedy*

- a. Himpunan Kandidat: Himpunan *diamond* yang berada pada *board*, termasuk *diamond* merah yang bernilai 2 point dan *diamond* biru yang bernilai 1 poin
- b. Himpunan Solusi : Himpunan *diamond* yang dipilih
- c. Fungsi Solusi : Menjumlahkan seluruh *diamond* yang telah didapatkan
- d. Fungsi Seleksi : Memilih *diamond* dengan jarak terdekat dengan *base*
- e. Fungsi kelayakan : Memeriksa jumlah *diamond* yang telah diperoleh tidak lebih dari 5 , dan apabila jumlah *diamond* telah mencapai lebih dari 4, maka *bot* akan diarahkan menuju *base* untuk menyimpan *diamond*
- f. Fungsi objektif : Jumlah poin *diamond* yang diperoleh maksimum

2. Analisis Efisiensi Solusi

Pada alternatif ini, diimplementasikan algoritma pengambilan *diamond* dengan jarak terdekat dari *base* dengan uraian sebagai berikut. Algoritma ini akan mencari seluruh *diamond* yang terdapat pada *board*, dan kemudian melakukan iterasi untuk menghitung jarak *diamond* dengan *base*. Kompleksitas algoritma untuk menentukan jarak *diamond* dengan *base* dalam suatu array adalah $O(n)$ dan

penentuan jarak minimum untuk setiap n buah objek diamond adalah $O(n^2)$, sehingga kompleksitas dari keseluruhan algoritmanya adalah $O(n^2)$. Alternatif ini dapat dioptimasi lebih lanjut dengan melakukan pengurutan jarak *diamond* dari *base* dengan kompleksitas $O(n \log(n))$, namun karena ada penerapan strategi *density* sehingga kompleksitas keseluruhan algoritmanya adalah $O(n^2)$.

3. Analisis Efektivitas Solusi

Strategi *greedy* ini mengutamakan *diamond* yang memiliki jarak terdekat dari *base*. *Bot* akan terlebih dahulu mengambil semua *diamond* yang berada dalam radius *base*, lalu jika *diamond* yang berada di sekitar *base* sudah habis, maka akan dilakukan pengambilan *diamond* dengan *greedy by high-density*.

Strategi ini menjadi efektif apabila:

- a. *Base* dari *bot* yang terletak di daerah yang memiliki konsentrasi *diamond* yang tinggi, sehingga *bot* dapat mengambil *diamond* di sekitar *base*, dan langsung pulang ke *base* dengan jarak yang tidak terlalu jauh.
- b. *Bot* tidak *spawn* di tempat yang dekat dengan *bot* lainnya, sehingga tidak terjadi perebutan *diamond* di wilayah sekitar *base*.
- c. *Base* dari *bot* yang terletak ditengah *board*, sehingga membuat jarak ke seluruh wilayah menjadi dekat.
- d. Terdapat teleporter yang dapat memperpendek jarak tempuh dari *bot* menuju *diamond* yang menjadi tujuan.
- e. Tidak berada dekat dengan *bot* lawan, sehingga tidak terjadi tabrakan atau *collision* yang dapat menyebabkan *diamond* dicuri oleh lawan.

Strategi ini menjadi tidak efektif apabila:

- a. *Base* dari *bot* yang terletak di tepi *board*, sehingga jarak *base* dengan *diamond* yang ada di *board* cenderung lebih jauh.
- b. *Base* dari *bot* yang terletak di daerah dengan kepadatan *diamond* yang sedikit.

- c. Jarak antara *bot* dengan *bot* lawan yang dekat sehingga terjadi tabrakan atau *collision*.
- d. *Bot spawn* di daerah yang berdekatan dengan *base* lawan, sehingga diamond di sekitar *base* diperebutkan oleh *bot* lawan.
- e. Terdapat teleporter yang menghalangi jalan *bot*, sehingga harus membuang beberapa waktu dan langkah untuk mencapai tempat tujuan.

3.1.4. Alternatif *greedy by Highest Diamond Value*

Greedy by Highest Diamond Value adalah strategi *greedy* yang mengutamakan jumlah poin dari diamond yang hendak dituju. Dalam strategi *greedy* ini, *bot* akan cenderung menuju ke *diamond* berwarna merah dibandingkan *diamond* berwarna biru, karena *diamond* merah memiliki nilai poin yang lebih tinggi, yaitu 2 poin, dibandingkan dengan *diamond* biru yang hanya bernilai 1 poin. *Bot* akan mengincar *diamond* berwarna merah terdekat, kemudian jika dalam perjalanan *Bot* menuju *diamond* merah, terdapat diamond biru yang berjarak 2 dari *Bot*, maka *bot* akan mengambil *diamond* biru tersebut. Algoritma ini juga memperhatikan jarak teleporter ke *diamond*.

1. Pemetaan Elemen *Greedy*:

- a. Himpunan Kandidat: Himpunan *diamond* yang berada pada *board*, termasuk *diamond* merah yang bernilai 2 point dan *diamond* biru yang bernilai 1 poin
- b. Himpunan Solusi : Himpunan *diamond* yang dipilih
- c. Fungsi Solusi : Menjumlahkan seluruh *diamond* yang telah didapatkan
- d. Fungsi Seleksi : Memilih *diamond* dengan poin 2 terdekat
- e. Fungsi kelayakan : Memeriksa jumlah *diamond* yang telah diperoleh tidak lebih dari 5, dan apabila jumlah *diamond* telah mencapai lebih dari 4, maka *bot* akan diarahkan menuju *base* untuk menyimpan *diamond*
- f. Fungsi objektif : Jumlah poin *diamond* yang diperoleh maksimum

2. Analisis Efisiensi Solusi:

Pada alternatif ini, akan dilakukan pencarian seluruh *diamond* merah yang ada di dalam *board*, kemudian disimpan dalam suatu array. Selain itu, juga akan dicari semua *diamond* biru yang terdapat di dalam *board* permainan. Kemudian, dilakukan iterasi untuk menentukan jarak setiap *diamond* dari *bot*. Setelah mendapatkan jarak setiap *diamond* di kedua array, akan dilakukan pengurutan dari yang terkecil ke terbesar. Kompleksitas algoritma untuk menemukan *diamond* dengan poin terbesar dari suatu array adalah $O(n^2)$. Alternatif ini dapat dioptimasi lebih lanjut dengan mempertimbangkan pengurutan terhadap array secara terurut mengecil dengan fungsi bawaan pada *python* yaitu $O(n \log(n))$, sehingga kompleksitas dari algoritmanya adalah $O(n \log(n))$.

3. Analisis Efektivitas Solusi:

Strategi ini mengutamakan *bot* untuk menuju *diamond* yang mempunyai poin terbanyak dimana di permainan ini *diamond* dengan poin terbanyak adalah *diamond* berwarna merah. *Bot* akan diarahkan untuk mengambil *diamond* merah terdekat, kemudian saat perjalanan menuju *diamond* merah, jika ada *diamond* biru yang berjarak kurang dari 2, maka *bot* akan mengambil *diamond* tersebut.

Strategi ini menjadi efektif apabila :

- a. *Bot spawn* di daerah yang terdapat banyak *diamond* berwarna merah yang berpoin tinggi.
- b. Terdapat banyak *diamond* berwarna merah yang muncul di *board*.
- c. Terdapat *diamond* biru yang berjarak kurang dari 2 saat perjalanan mengambil *diamond* yang berwarna merah.
- d. Terdapat teleporter yang dapat memperpendek jarak tempuh dari *bot* menuju *diamond* yang menjadi tujuan.
- e. Tidak berada dekat dengan *bot* lawan, sehingga tidak terjadi tabrakan atau *collision* yang dapat menyebabkan *diamond* dicuri oleh lawan.

Strategi ini menjadi tidak efektif apabila :

- a. *Diamond* berwarna merah berjarak sangat jauh dengan base, sehingga membuat *bot* harus melakukan perjalanan yang jauh sehingga membuang waktu untuk perjalanan balik ke *base*.
- b. Jika *diamond* berwarna merah habis dalam permainan, *bot* akan mengubah target menjadi *diamond* biru terdekat.
- c. *Bot* lawan yang bermain di pertandingan juga mengincar *diamond* berwarna merah, sehingga akan sering terjadi tabrakan di daerah yang terdapat *diamond* merah.
- d. Terdapat teleporter yang menghalangi jalan *bot*, sehingga harus membuang beberapa waktu dan langkah untuk mencapai tempat tujuan.

3.1.5. Alternatif *greedy by Tackle*

Greedy by Tackle adalah strategi *greedy* untuk mengutamakan menabrak *bot* lawan untuk merebut *diamond* yang dimiliki *bot* lawan. *Bot* akan mentarget *bot* lawan yang berada di dekatnya, namun jika semua *bot* lawan berada jauh dari *bot*, maka *bot* akan berkeliling mencari *diamond* terdekat.

1. Pemetaan Elemen *Greedy*:

- a. Himpunan Kandidat: Himpunan *diamond* yang berada pada board, termasuk *diamond* merah yang bernilai 2 point dan *diamond* biru yang bernilai 1 poin
- b. Himpunan Solusi : Himpunan *diamond* yang dipilih
- c. Fungsi Solusi : Menjumlahkan seluruh *diamond* yang telah didapatkan
- d. Fungsi Seleksi : Mencari lawan terdekat untuk ditabrak
- e. Fungsi kelayakan : Memeriksa jumlah *diamond* yang telah diperoleh tidak lebih dari 5 dan apabila jumlah *diamond* telah mencapai lebih dari 3, maka *bot* akan diarahkan menuju *base* untuk menyimpan *diamond*
- f. Fungsi objektif : Jumlah poin *diamond* yang diperoleh maksimum

2. Analisis Efisiensi Solusi:

Pada alternatif ini, diimplementasikan algoritma yang cukup berbeda dengan algoritma *greedy* lain, dimana pada algoritma ini fokus utama *bot* ini adalah menabrak *bot* lawan terdekat untuk merebut *diamond* yang dimilikinya. Kompleksitas algoritma untuk mencari *bot* lawan di *board* adalah $O(n)$ dan pengurutan posisi *bot* lawan terdekat untuk setiap n *bot* lawan adalah $O(n^2)$. Alternatif ini dapat dioptimasi lebih lanjut dengan melakukan pengurutan array secara terurut, sehingga kompleksitas keseluruhannya adalah $O(n \log(n))$.

3. Analisis Efektivitas Solusi:

Konsep dalam strategi ini cukup berbeda dengan algoritma *greedy* yang lain. *Bot* berfokus untuk mencari lawan terdekat untuk dikejar dan ditabrak. Jika *bot* lawan berada jauh dari *bot*, maka *bot* akan menuju ke posisi *diamond* terdekat.

Strategi ini menjadi efektif apabila :

- a. *Bot* lawan berada di dekat dengan *bot*
- b. Terdapat banyak *diamond* yang berada di dekat *bot*, sehingga memancing *bot* lawan untuk datang di tempat tersebut
- c. *Bot* lawan yang ditabrak memiliki *diamond* di *inventory*-nya

Strategi ini menjadi tidak efektif apabila :

- a. *Bot* lawan terletak berjauhan dengan *bot* kita
- b. *Bot* musuh bergerak menjauh dari *bot*, sehingga terjadi kejadian saling mengejar yang mengakibatkan waktu terbuang sia-sia
- c. *Bot* lawan tidak mempunyai *diamond* di *inventory*.

3.2. Strategi *Greedy* yang diimplementasikan

Berdasarkan alternatif - alternatif *greedy* yang telah dijelaskan pada bagian 3.1, kelompok kami melakukan *testing* berulang-ulang baik secara pertandingan antar *bot* maupun *bot* bermain sendiri dan memutuskan untuk memakai algoritma *Greedy by Nearest Base* yang digabungkan dengan *Greedy by High Density* dikarenakan waktu pada

pertandingan yaitu 60 detik dengan perhitungan 1 gerakan per detik sehingga pengambilan *diamond* harus benar-benar secara efektif dan maksimal. Dengan mengutamakan *diamond* yang berada di sekitar *base* akan menghemat waktu sehingga tidak membuang waktu saat mau balik ke *base* untuk menyimpan *diamond*. Selain itu, jika mengutamakan pergerakan di sekitar *base* akan mengurangi resiko kehilangan *diamond* karena ditabrak oleh bot lawan. Jika di sekitar *base* tidak terdapat *diamond*, maka *bot* akan mengubah strategi menjadi *Greedy by Density*. *Bot* akan mengincar daerah yang terdapat banyak *diamond*, sehingga dengan harapan bahwa setidaknya jika *bot* harus berjalan jauh menuju *diamond*, maka harus menuju posisi yang terdapat banyak *diamond* agar perbandingan *diamond* dan waktu setara. Algoritma ini memanfaatkan teleporter untuk mencari *diamond*, *red button*, dan balik ke *base*. Pencarian *red button* dilakukan ketika jarak *diamond* jauh dan *red button* lebih dekat sehingga akan menuju *red button* dengan harapan *diamond* yang di reset akan muncul dekat dengan *bot* atau *base*.

Alasan alternatif *Greedy by Tackle* tidak dinyatakan sebagai solusi yang paling optimal karena solusi ini sangat bergantung pada bot lawan, sehingga sangat kurang konsisten. Penerapan solusi ini bisa dibilang hanya bergantung pada keberuntungan. Algoritma ini juga tidak optimal karena pergerakan *bot* lawan juga sama yaitu 1 gerakan per detik sehingga kemungkinan terbesar akan terjadi kejar-kejaran dan tidak bertabrakan dengan *bot* lain.

Alasan alternatif *Greedy by Diamond Value* tidak digunakan karena ketika *bot* mengincar *diamond* merah yang memiliki poin 2, terkadang *bot* harus melakukan perjalanan yang cukup jauh sehingga ketika harus balik ke *base* untuk menyimpan *diamond*. Hal ini membuat *bot* kehabisan waktu di perjalanan sehingga tidak ada waktu untuk mencari *diamond* lain.

Alasan alternatif *Greedy by Shortest Bot Distance* tidak dipilih karena jika ada *diamond* yang jaraknya sama dan dimana salah satu *diamond* tersebut lebih dekat ke arah *base*, *bot* belum tentu memilih *diamond* yang lebih dekat tersebut. Sebenarnya, solusi ini cukup seimbang dengan alternatif *Greedy by Nearest Base*, namun karena kami menilai bahwa yang dipertimbangkan oleh solusi ini adalah jarak *bot* dengan *diamond* sehingga terkadang *bot* bisa berjalan cukup jauh dari *base* untuk mengambil *diamond*. *Bot* juga terkadang bisa melakukan pengambilan *diamond* di daerah yang sepi, sehingga jika harus

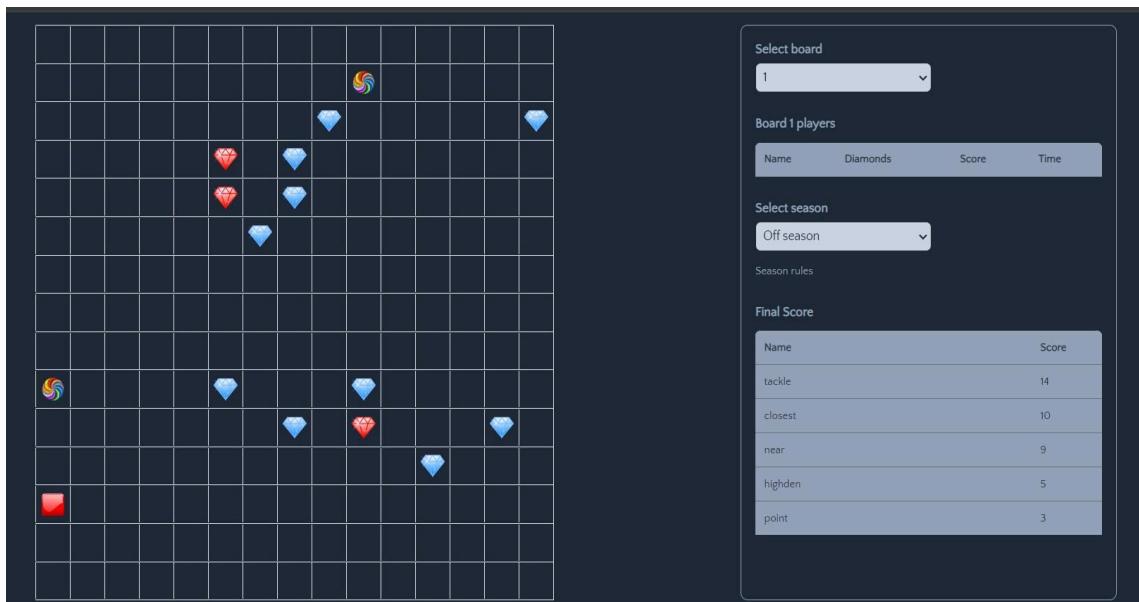
memutar arah untuk mengambil *diamond* lain menjadi membuang-buang langkah dan waktu yang dimiliki.

Alasan alternatif *Greedy by Highest Density* tidak digunakan karena fokus utama dari *bot* ini adalah menuju ke tempat yang memiliki *diamond* yang banyak dan terkadang wilayah ini bisa cukup jauh sehingga memaksa *bot* harus melakukan perjalanan jauh. Selain itu, juga karena *bot* lawan yang cenderung juga mengincar daerah yang terdapat banyak *diamond*, sehingga resiko untuk ditabrak *bot* lawan juga cukup besar.

Berikut merupakan hasil pertandingan dari alternatif tersebut.

1. Laga 1

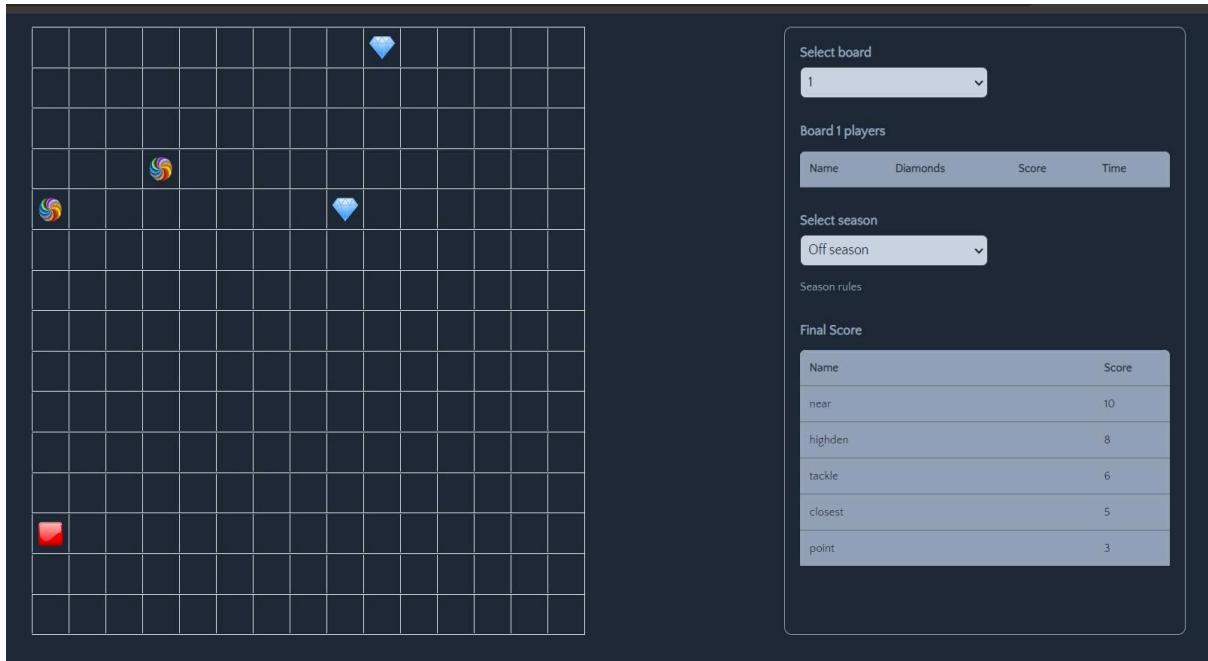
Pemenangnya adalah *Tackle Bot*



Gambar 3.2.1. Hasil pertandingan 1 antar alternatif

2. Laga 2

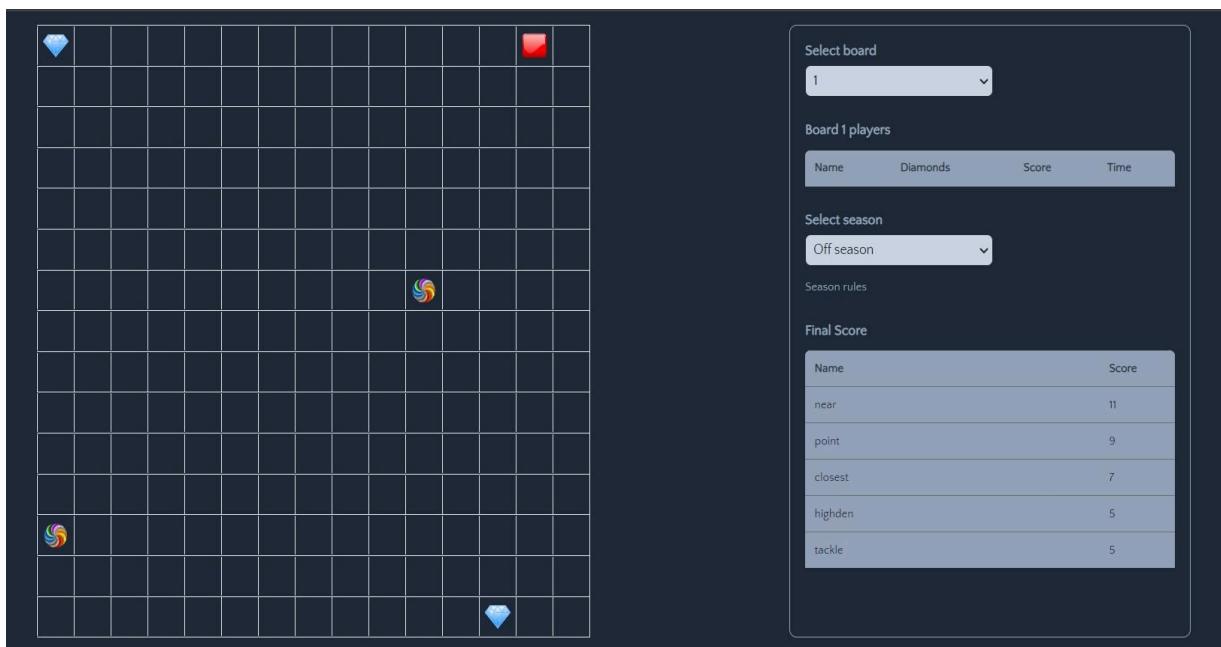
Pemenangnya adalah *Near Base Bot*



Gambar 3.2.2. Hasil pertandingan 2 antar alternatif

3. Laga 3

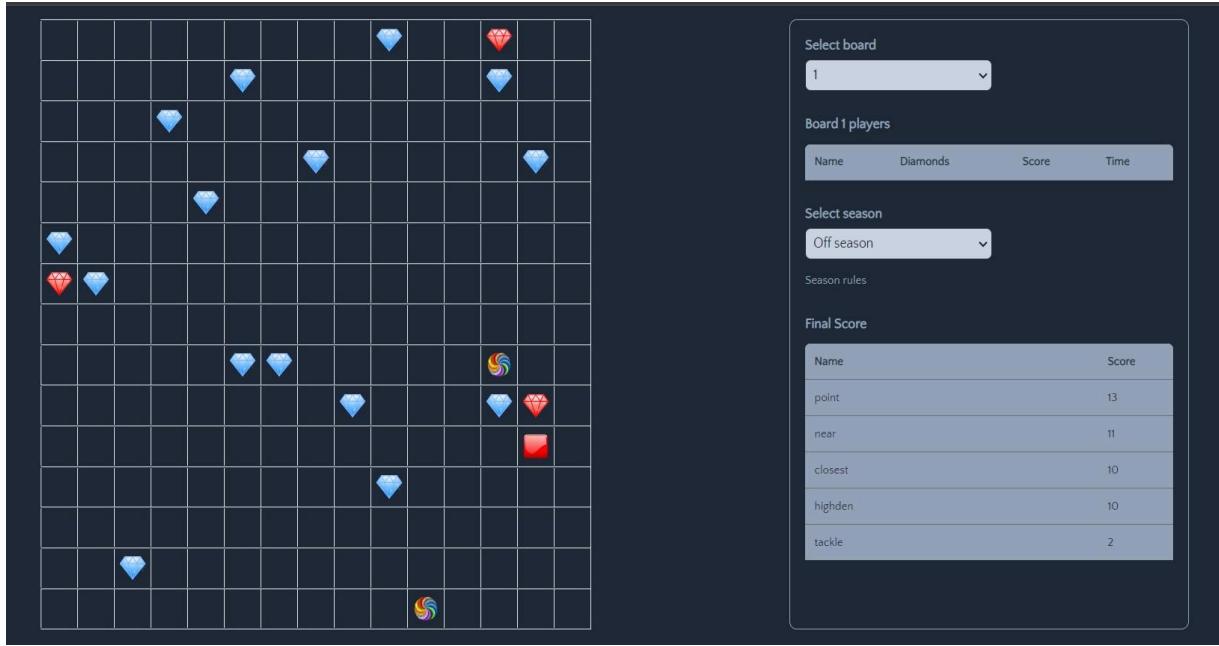
Pemenangnya adalah *Near Base Bot*



Gambar 3.2.3. Hasil pertandingan 3 antar alternatif

4. Laga 4

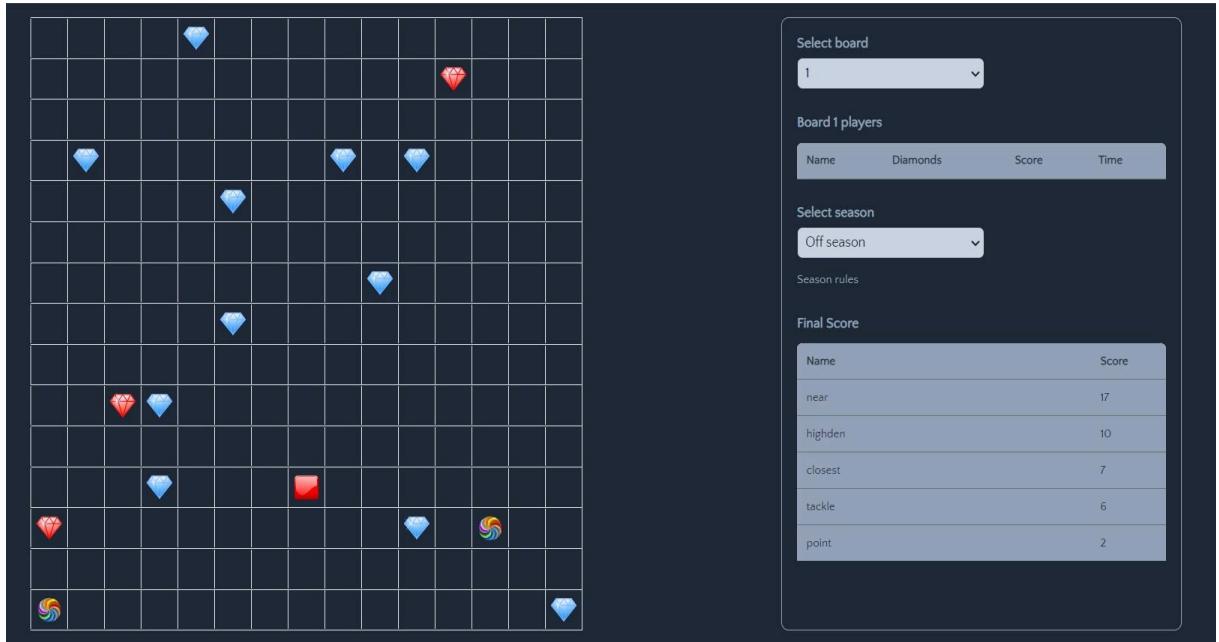
Pemenangnya adalah *Point Diamond Bot*



Gambar 3.2.4. Hasil pertandingan 4 antar alternatif

5. Laga 5

Pemenangnya adalah *Near Base Bot*



Gambar 3.2.5. Hasil pertandingan 5 antar alternatif

Dari hasil banyak *testing* yang dilakukan, *Near Base Bot* memiliki tingkat performa yang bagus karena jumlah *diamond* yang diambil dan persentase juara 1 cukup stabil. Dapat dilihat juga dari 5 data di atas bahwa terjadi perubahan pemenang dari setiap game oleh beberapa alternatif. Dari kelima laga tersebut, *Near Base Bot* memiliki performa yang paling bagus, dimana ditandai dengan juara 1 sebanyak 3 kali , juara 2 sebanyak 1 kali , dan juara 3 sebanyak 1 kali. Sehingga dari hasil uji coba dapat dilihat bahwa *Near Base Bot* (alternatif *Greedy by Nearest Base* yang digabungkan dengan *Greedy by Highest Density*) merupakan *bot* yang paling efisien dan efektif serta bisa dibilang yang paling konsisten karena selalu berada di dalam 3 besar meskipun kondisi board dalam setiap laga berbeda-beda.

Sebenarnya ada juga alternatif *Greedy by Point Diamond* yang sebenarnya berada di juara 1 dan 2, namun *bot* ini tidak konsisten karena pernah 2 kali menjadi juara terakhir. Hal ini karena *bot* ini sangat bergantung kepada kondisi board permainan. Jadi, strategi ini tidak bisa dipilih menjadi strategi utama. Selain *Greedy by Point Diamond*, terdapat juga alternatif *Greedy by Tackle*. Algoritma ini bisa dibilang cukup efektif dan juga sangat meresahkan pihak lawan. *Bot* ini bisa benar-benar mematikan *bot* lawan dari suatu pertandingan dan mendapatkan *diamond* dengan instan tanpa membutuhkan waktu yang lama untuk mengumpulkan *diamond*. Namun, *bot* ini bisa dibilang sangat beresiko tinggi karena *bot* ini bergantung kepada *bot* lawan. Jika tidak ada lawan di sekitar *bot* atau pergerakan *bot* lawan yang tidak searah dengan *bot*, maka *bot* akan membuang-membuang waktu untuk mengejar *bot* lain.

Oleh karena itu, dapat disimpulkan bahwa alternatif *Greedy by Nearest Bot* merupakan alternatif yang paling optimal, sehingga alternatif *greedy* yang diimplementasikan adalah alternatif *Greedy by Nearest Bot* yang digabungkan dengan *Greedy by Highest Density*.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi dalam Pseudocode

Fungsi *findDistanceByBotAndBase*

{ Fungsi ini merupakan fungsi untuk menghasilkan dua buah list , dimana list pertama adalah list jarak setiap diamond dengan base dan list jarak setiap diamond dengan bot.

```
function findDistanceByBotAndBase(object: Game_objects, current_position: Position )  
-> list , list  
    x ← []  
    y ← []  
  
    for position in object do  
        posX ← abs(base.x - position.position.x)  
        posY ← abs(base.y - position.position.y)  
        distance ← posX + posY  
        x.append([distance,position.position,[posX,posY],position.properties.points])  
        posX ← abs(current_position.x - position.position.x)  
        posY ← abs(current_position.y - position.position.y)  
        distance1 ← posX + posY  
        y.append([distance1,position.position,[posX,posY],position.properties.points])  
    → x , y
```

Fungsi *count_distance*

{ Fungsi ini merupakan fungsi untuk menghitung jarak dari suatu lokasi ke lokasi lain.

```
function count_distance (current_x: integer, current_y: integer, dest_x: integer,  
dest_y: integer ) -> list , list  
    delta_x ← dest_x - current_x  
    delta_y ← dest_y - current_y  
    → (abs(delta_x) + abs(delta_y))
```

Fungsi getAllTeleporterSorted

{ Fungsi getAllTeleporterSorted merupakan fungsi untuk mencari semua pasangan teleporter di dalam board , dan kemudian diurutkan dari yang terdekat

```
function getAllTeleporterSorted (board: Board ,current_position: Position) -> List
    teleport ← [x for x in board.game_objects if (x.type =
    "TeleportgameObject")]
    teleporter ← []
    for teleporter in teleport do
        pair_id ← teleporter.properties.pair_id
        if pair_id not in teleporter_groups then
            teleporter_groups[pair_id] ← []
            teleporter_groups[pair_id].append(teleporter)
        sorted_teleporter_groups ← []
        for pair_id, teleporters in teleporter_groups.items() do
            sorted_teleporters ← sorted([(teleporter.position, abs
                (current_position.x - teleporter.position.x) + abs(current_position.y -
                teleporter.position.y)) for teleporter in teleporters], key ← lambda t
                :t[1])
            sorted_teleporter_groups[pair_id] ← sorted_teleporters

        sorted_teleport_groups ← dict(sorted(sorted_teleport_groups.items(), key ←
            lambda item: item[1][0][1]))
    → sorted_teleport_groups
```

Fungsi find_densest

{ Fungsi find_densest merupakan fungsi yang digunakan untuk mengelompokkan diamond terdekat menjadi sebuah cluster berdasarkan kepadatan dan kedekatannya dengan satu sama lain }

```
function neighbors(diamond: GameObject)→ List[GameObject]
    → [other for other in diamonds if count_distance(diamond.position.x,
        diamond.position.y, other.position.x, other.position.y) ≤ eps]

function find_densest(diamonds: List[GameObject], eps: Float, min_samples: Integer)
    → List[List[GameObject]]
    clusters ← []
```

```

visited ← set()
noise ← set()

for diamond in diamonds do
    if diamond.id in visited then
        continue

    visited.add(diamond.id)
    neighbor_diamonds = neighbors(diamond)
    if len(neighbor_diamonds) < min_samples then
        noise.add(diamond.id)
    else
        cluster ← []
        cluster.append(cluster)
        cluster.append(diamond)
        for neighbor in neighbor_diamonds do
            if neighbor.id in noise then
                cluster.append(neighbor)
                noise.remove(neighbor,id)
            if neighbor.id not in noise then
                visited.add(neighbor.id)
                more_neighbors ← neighbors(neighbor)
                if len(more_neighbors) ≥ min_samples then
                    neighbor_diamonds.extend(more_neighbors)
                    if neighbor not in cluster then
                        cluster.append(neighbor)

→ clusters

```

Fungsi next_move

{ Fungsi next_move merupakan fungsi untuk menentukan langkah selanjutnya yang akan dilalui bot berdasarkan algoritma greedy. Fungsi ini mengembalikan tuple of integer yang merupakan (delta_x,delta_y).

```

function next_move(board_bot: GameObject, board: Board) → Tuple of integer
    { Inisialisasi }

    { Terdapat juga inisialisasi goal_position yang merupakan variabel yang
    terdapat pada class HighestDensity ini , yang diinisialisasi pada class dan di

```

```

luar fungsi ini}

props ← board_bot.properties
current_position ← board_bot.position

board_width ← board.width
board_height ← board.height

# Mencari semua diamond dan red button pada board
all_diamonds ← [x for x in board.game_objects if (x.type =
"DiamondGameObject")]
red_button ← [x for x in board.game_objects if (x.type =
"DiamondButtonGameObject")]

# Mencari cluster dengan diamond terbanyak
cluster ← find_densest(all_diamonds)
densest_cluster ← max(cluster, key ← len, default = [])

diamond_distance_base, diamond_distance_bot ←
findDistanceByBotAndBase(all_diamonds, base, current_position)

red_button_base, red_button_bot ← findDistanceByBotAndBase(red_button, base,
current_position)

diamond_sorted_base ← sorted(diamond_distance_base, key ← lambda d: d[0])
diamond_sorted_bot ← sorted(diamond_distance_bot, key ← lambda d: d[0])
red_button_sorted_base ← sorted(red_button_base, key ← lambda d: d[0])
red_button_sorted_bot ← sorted(red_button_bot, key ← lambda d: d[0])
base_distance ← count_distance(current_position.x, current_position.y,
board_bot.properties.base.x, board_bot.properties.base.y)
base ← board_bot.properties.base
Sorted_teleport_groups ← getAllTeleporterSorted(board, current_position)

# Mencari titik tengah dari cluster
if densest_cluster then
    centroid_x ← sum(d.position.x for d in densest_cluster) /
    len(densest_cluster)
    centroid_y ← sum(d.position.y for d in densest_cluster) /
    len(densest_cluster)
    densest_centroid ← Position(x ← int(centroid_x), y ← int(centroid_y))

else

```

```

densest_centroid ← None

# Kondisi untuk tujuan akhir
if props.diamonds ≥ 4 then
    if (diamond_sorted_bot[0][0] ≤ 2 and props.diamonds +
        diamond_sorted_bot[0][3] ≤ 5) then
        self.goal_position ← diamond_sorted_bot[0][1]
    else
        base ← board_bot.properties.base
        self.goal_position ← base
    else
        if (base_distance = 1 and props.diamonds ≥ 2) then
            self.goal_position ← base
        else if (board_bot.properties.milliseconds_left ≤ 8000 and props.diamonds ≥
2) then
            if (diamond_sorted_bot[0][0] ≤ 1) then
                self.goal_position ← diamond_sorted_bot[0][1]
            else
                self.goal_position ← base
        else if (diamond_sorted_bot or diamond_sorted_base) then
            if (props.diamonds ≥ 3 and base_distance ≤ 3) then
                self.goal_position ← base
            else if (diamond_sorted_bot[0][0] ≤ 2) then
                self.goal_position ← diamond_sorted_bot[0][1]
            else if ((diamond_sorted_base[0][2][0] < 0.4 * board_width) and
(diamond_sorted_base[0][2][1] < 0.4 * board_height)) then
                if ((diamond_sorted_bot[0][0] ≤ 2) then
                    self.goal_position ← diamond_sorted_bot[0][1]
                else
                    self.goal_position ← diamond_sorted_base[0][1]
            else if (densest_centroid) then
                self.goal_position ← densest_centroid
            else
                if (red_button_sorted_base ≠ []) then
                    if ((red_button_sorted_base[0][2][0] < 0.4 * board_width) and
(red_button_sorted_base[0][2][1] < 0.4 * board_height)) then
                        self.goal_position ← red_button_sorted_base[0][1]
                    else if (red_button_sorted_bot[0][0] ≤ 3) then
                        self.goal_position ← red_button_sorted_bot[0][1]
                    else
                        self.goal_position ← diamond_sorted_bot[0][1]

```

```

    else
        if (props.diamonds ≥ 3) then
            self.goal_position ← base
        else
            self.goal_position ← red_button

    current_position ← board_bot.position
    if self.goal_position then
        shortest_way ← count_distance(current_position.x, current_position.y,
                                         self.goal_position.x, self.goal_position.y)
        shortest_way_position ← self.goal_position

    for pair_id, teleporters in sorted_teleport_groups.items() do
        closest_teleporter, distance_to_closest_teleporter ← teleporters[0]
        second_teleporter ← teleporters[1][0]
        distance_tele2_goal ← count_distance(second_teleporter.x,
                                               second_teleporter.y, self.goal_position.x, self.goal_position.y)

        way1 ← distance_to_closest_teleporter + distance_tele2_goal
        if way1 < shortest_way then
            shortest_way ← way1
            shortest_way_position ← closest_teleporter

    if (current_position = shortest_way_position) then
        delta_x, delta_y ← random.choice([(1, 0), (0, 1), (-1, 0), (0, -1)])
    else
        self.goal_position ← shortest_way_position
        delta_x, delta_y ← get_direction(current_position.x,
                                         current_position.y, goal_position.x, goal_position.y)
    else
        delta_x ← self.directions[self.current_direction]
        delta_x ← delta[0]
        delta_y ← delta[1]
        if random.random() > 0.6 then
            self.current_direction ← (self.current_direction + 1) mod
                len(self.directions)

    if (delta_x = 0 and delta_y = 0) then
        delta_x, delta_y ← random.choice([(1, 0), (0, 1), (-1, 0), (0, -1)])
→ delta_x, delta_y

```

4.2. Struktur Data Program

Bahasa yang digunakan untuk melakukan implementasi *bot* adalah bahasa pemrograman *Python*. Struktur data dalam program didefinisikan dengan menggunakan *class*. *Class* tersebut didefinisikan di folder *game/* pada *bot* starter pack.

Class yang terdapat pada *models.py* antara lain :

- *Class bot* merupakan kelas yang mendefinisikan *bot* yang digunakan dalam permainan. Dalam *class bot* ini terdapat atribut nama, *email*, dan id.
- *Class Position* merupakan kelas yang digunakan untuk mendefinisikan posisi *game* objek dalam suatu *game* percobaan. Pada *class* ini terdiri dari 2 komponen yaitu x dan y.
- *Class Properties* merupakan kelas yang merepresentasikan berbagai properti yang dimiliki oleh objek dalam game sebagai berikut.
 - *points* merupakan jumlah poin yang dimiliki oleh *gameobject* tersebut, jika *gameobject* tersebut tidak memiliki poin, maka akan poin akan bernilai 0.
 - *pair_id* merupakan informasi yang digunakan untuk melakukan *pairing* pada Teleporter.
 - *diamonds* merupakan total jumlah *diamond* yang sedang dipegang oleh *bot*.
 - *score* merupakan poin total yang telah disimpan oleh *bot* ke dalam *base*-nya.
 - *name* berisi informasi nama dari objek tersebut.
 - *inventory_size* berisi informasi terkait ukuran *inventory* yang dimiliki oleh objeknya, biasanya berukuran 5 untuk objek *bot*.
 - *can_tackle* berisi informasi apakah sebuah objek bisa di tackle atau tidak.
 - *milliseconds_left* berisi informasi waktu sisa yang dimiliki *bot* dalam suatu pertandingan.
 - *time_joined* berisi informasi waktu saat *bot* masuk ke dalam *board* pertandingan.
- *Class GameObject* merupakan kelas yang menampung informasi berupa id, posisi, tipe, dan *properties* daripada sebuah objek. Informasi *properties* telah didefinisikan di *Class* sebelumnya.
- *Class Board* merupakan kelas yang berisi id, lebar, tinggi, *features*, *minimum_delay_between_moves* (jeda antar gerakan pada *bot*), *game_objects* yang berisi

objek-objek permainan yang terdapat pada papan permainan. Pada *class* ini juga terdapat fungsi seperti *is_valid_move* untuk mengecek apakah gerakan *bot* valid atau tidak.

Pada file *util.py* yang terdapat pada folder *game/* juga terdapat beberapa fungsi yang dapat dipakai sebagai berikut

- Fungsi *clamp* untuk mengembalikan nilai *n* yang terletak di antara batas nilai terkecil dan terbesar. Jika nilai *n* lebih kecil dibandingkan batas nilai terkecil, maka akan diambil nilai dari batas terkecil tersebut, dan berlaku sebaliknya.
- Fungsi *get_direction* untuk mengembalikan nilai *delta_x* dan *delta_y* yang merupakan arah gerakan *bot* yang akan dilakukan
- Fungsi *position_equals* untuk menentukan apakah dua buah posisi yang dijadikan sebagai parameter merupakan posisi yang sama atau bukan.
- Fungsi *count_distance* untuk menghitung jarak dari suatu posisi ke posisi lain.
- Fungsi *find_teleport* untuk mencari seluruh teleporter di *board* beserta dengan jaraknya dari *bot*.
- Fungsi *find_red_button* untuk mencari seluruh *red button* di *board* beserta dengan jaraknya dari *bot*.
- Fungsi *find_blue_diamond* untuk mencari seluruh *diamond* biru beserta dengan jaraknya dari *bot*.
- Fungsi *find_diamond* untuk mencari seluruh seluruh *diamond* di dalam *board* beserta dengan jaraknya dari *bot*.
- Fungsi *find_densest* untuk mencari daerah yang mempunyai sekumpulan *diamond*.
- Fungsi *getAllTeleporterSorted* untuk mencari semua teleporter di *board* dan sudah terurut.
- Fungsi *findDistanceByBotAndBase* untuk mencari jarak suatu objek dari *bot* dan dari *base*.

Terdapat juga file-file yang berisi *logic* yang digunakan dalam melakukan implementasi alternatif *greedy* yang digunakan sebagai pengatur gerakan dari *bot*. File ini terdapat pada folder *game/logic/*. File ini berisi fungsi *next_move()* yang merupakan fungsi yang menerapkan alternatif *greedy* utama untuk menentukan *goal_position*, kemudian akan dipanggil fungsi

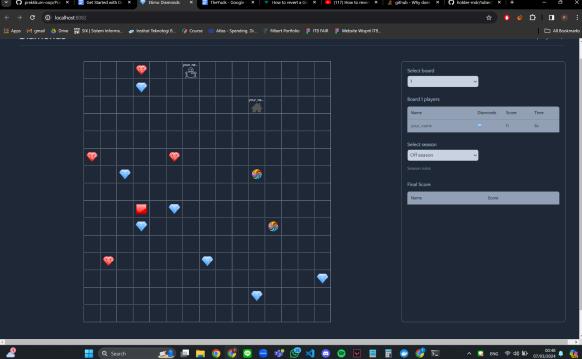
get_direction untuk mendapatkan *delta_x* dan *delta_y*. Terdapat juga folder *otherBot* yang berisi alternatif *greedy* lain yang dibuat.

File *main.py* yang terdapat pada folder *game/* berisi algoritma utama yang digunakan untuk menghubungkan semua yang terdapat pada *bot starter pack* ini. Kita dapat menambahkan logika yang telah diimplementasikan pada folder *game/logic/* dengan melakukan *import* terhadap alternatif tersebut dan kemudian diletakkan pada *CONTROLLERS* yang terdapat pada file ini.

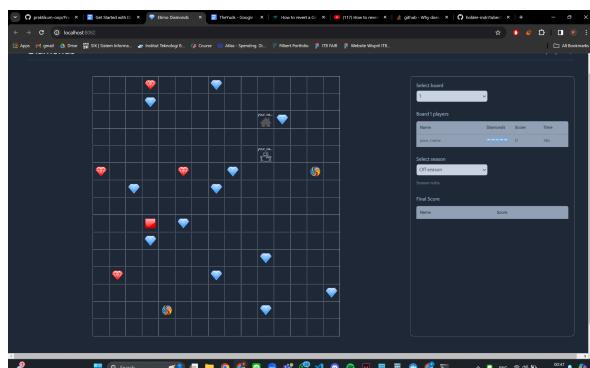
4.3. Analisis dan Pengujian

Pada bagian ini akan dilakukan pengujian terhadap beberapa fitur utama dari *bot* yang akan dipilih, yakni *Greedy by Nearest Base*, kemudian ditambah dengan pencarian diamonds berdasarkan *Highest Density* jika tidak ada diamond dekat dengan base, pergerakan bot menuju base, penggunaan *teleporter* untuk menentukan jarak optimal menuju *diamond* ataupun *base*.

Tabel 4.3.1 Hasil Pengujian

| No | Gambar | Penjelasan |
|----|---|--|
| 1. |  <p><i>Gambar 4.3.1. Kondisi mencari diamond yang dekat base</i></p> | Pada gambar di samping, <i>bot</i> akan melakukan kalkulasi untuk mencari <i>diamond</i> terdekat dengan <i>base</i> , jika terdapat <i>diamond</i> yang dekat dengan base maka <i>bot</i> akan menuju <i>diamond</i> tersebut. Gambar di samping menunjukkan keberhasilan dari <i>greedy</i> dari <i>bot</i> ini yaitu <i>Greedy by Nearest Base</i> . Hal ini menandakan keberhasilan dari strategi utama dari <i>bot</i> ini yaitu mengambil <i>diamond</i> di sekitaran <i>base</i> terlebih dahulu. |

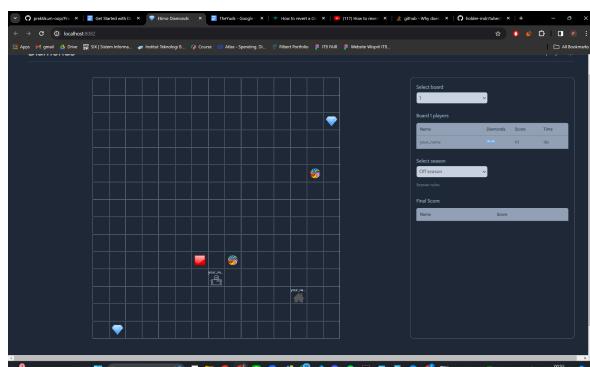
2.



Gambar 4.3.2. Kondisi kembali ke base jika diamond sudah penuh

Pada gambar di samping, dapat diamati bahwa jumlah poin *diamond* yang telah diperoleh *bot* sejumlah 5 poin sehingga *bot* memutuskan untuk kembali menuju *base* untuk menyimpannya sebagai skor. Hal ini menandakan keberhasilan dari fitur pergerakan *bot* menuju *base*.

3.



Gambar 4.3.3.1. Kondisi awal sebelum masuk teleporter

Pada gambar di samping, pada kondisi awal, *bot* memilih untuk menggunakan teleporter karena *bot* sudah mengkalkulasikan jarak terdekat untuk mengambil *diamond* yaitu dengan menggunakan teleporter, maka dari itu pada kondisi akhir, *bot* mengambil *diamond-diamond* yang berada di sekitaran teleport tersebut. Hal ini menandakan keberhasilan dari teleporter yang dipakai untuk menguntungkan *bot* kita dalam mencapai tujuannya yakni mengambil *diamond* ataupun ke *base*.

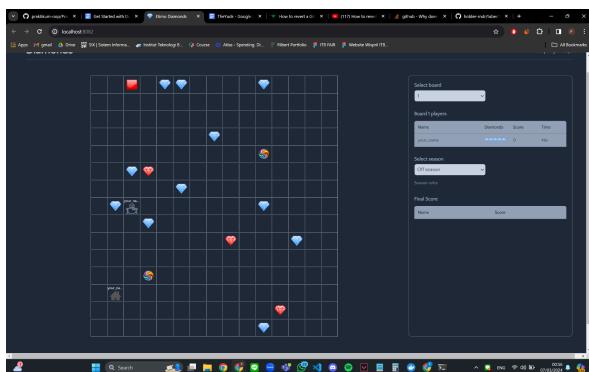


Gambar 4.3.3.2. Kondisi akhir setelah masuk teleporter

4.



Gambar 4.3.4.1. Kondisi awal di sekitar base dan bot tidak ada diamond



Gambar 4.3.4.2. Kondisi akhir mencari daerah yang memiliki diamond terbanyak

Pengujian pada gambar di samping menunjukkan bahwa bot akan mengambil *diamond* yang memiliki kepadatan terbesar (*highest density*) karena tidak ada *diamond* di sekitaran *base*. Hal ini menunjukkan bahwa fitur untuk memprioritaskan *highest density* ketika tidak ada *diamond* di sekitaran *base* telah berhasil.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dalam Tugas Besar IF2211 Strategi Algoritma, kami telah berhasil mengembangkan *bot* untuk permainan *Diamonds* yang menggunakan strategi *greedy* berdasarkan kedekatan *bot* dengan *base* (*Greedy by Nearest Base*). Proses pemilihan strategi dilakukan dengan membandingkan kinerja *bot* yang menggunakan strategi *greedy* yang berbeda berdasarkan data pertandingan antar *bot* dan data *bot* yang bermain sendiri.

Strategi *greedy* berdasarkan kedekatan dengan *base* terbukti lebih efektif dibandingkan dengan alternatif *greedy* lainnya, seperti *Greedy by Shortest Distance* (berdasarkan jarak terdekat dengan *bot*), *Greedy by Highest Density* (berdasarkan kepadatan tertinggi), *Greedy by Tackle* (menabrak *bot* lain), dan *Greedy by Point Diamond* (berdasarkan poin *Diamond*). Berdasarkan hasil statistik pertandingan, strategi ini lebih sering memenangkan pertandingan dibandingkan dengan strategi-strategi lainnya karena strategi tersebut lebih efektif untuk batas waktu dan ketentuan permainan.

Berdasarkan pengamatan dan analisis terhadap pengujian yang telah diimplementasikan, dapat disimpulkan bahwa strategi *Greedy by Nearest Base* yang kami implementasikan adalah yang paling optimal. Strategi ini mempertimbangkan fitur-fitur yang telah dibuat dan diterapkan, seperti pergerakan *bot* menuju *base* ketika *diamond* sudah penuh atau ketika *diamond* selanjutnya lebih jauh dibandingkan dengan *base*, penggunaan teleporter untuk menentukan jarak yang paling optimal menuju *diamond* atau *base*, serta implementasi dari strategi *Greedy by Highest Density*. Hasil yang diperoleh juga mempertimbangkan konfigurasi *board* yang diberikan. Faktor kemenangan pada permainan ini juga ditentukan berdasarkan keberuntungan pada saat *bot spawn* dan jarak *diamond* yang spawn dengan *bot* dan *base*.

Dengan demikian, kami menyimpulkan bahwa strategi *Greedy by Nearest Base*, yang dapat diakses di file *NearestBase.py*, adalah strategi yang paling optimal dengan mempertimbangkan fitur-fitur yang diimplementasikan, konfigurasi permainan, dan juga dari perbandingan statistik kemenangan *bot*.

5.2. Saran

Saran yang dapat diberikan untuk proses pengembangan *bot* untuk memenuhi tugas besar kali ini adalah sebagai berikut.

1. Lebih komunikatif antar anggota kelompok sehingga *progress* yang dibuat *bot* secara asinkron dapat tersampaikan dengan jelas kepada setiap anggota kelompok pada setiap pengembangannya,
2. Melakukan lebih banyak pertandingan atau analisis bersama dengan kelompok lain sehingga lebih mengetahui kelemahan daripada *bot* yang telah dikembangkan dibandingkan dengan hanya melakukan pertandingan antar *bot* yang dibuat sendiri,
3. Implementasi kode dapat dilakukan dengan lebih bersih sehingga tidak akan membutuhkan waktu yang cukup lama untuk melakukan proses *debugging*.
4. Jangan menunda nunda dalam menyelesaikan *bot* dengan berbagai strategi dan laporan.

Link *GitHub* : https://github.com/Benardo07/Tubes1_Holder-RNDR.git

Link video *YouTube* : <https://youtu.be/AZZOr8mV8ag>