

LAPORAN TUGAS BESAR 1 IF2211

STRATEGI ALGORITMA

Pemanfaatan Algoritma Greedy dalam Pembuatan Bot Permainan Diamonds



Dosen Pengampu : Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
Asisten pembimbing : Haziq Abiyyu Mahdy

Disusun oleh:
K02 - Snoopy

Kayla Namira Mariadi (13522050)
Andhita Naura Hariyanto (13522060)
Salsabiila (13522062)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023

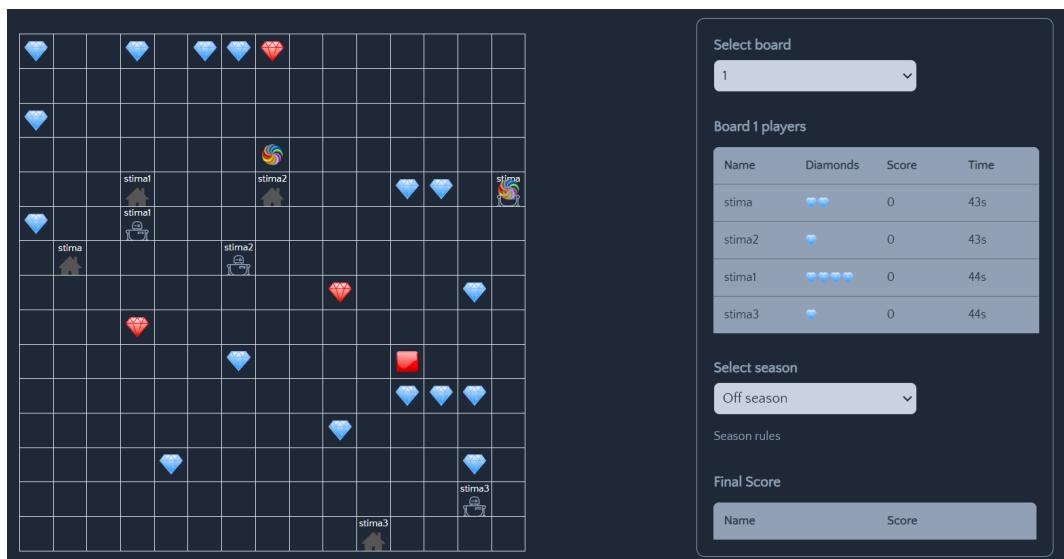
Daftar Isi

Daftar Isi	1
BAB I	2
BAB II	6
2.1. Algoritma Greedy	6
2.2. Cara Kerja Program Etimo Diamond	6
2.3. Alur Menjalankan Game Engine	7
2.4. Alur Menambahkan Logic Bot dan Menjalankan Bot	7
BAB III	9
3.1. Eksplorasi Alternatif Solusi Greedy	9
3.2. Algoritma Greedy yang Dipilih	17
BAB IV	20
4.1. Struktur Data Program	20
4.2. Implementasi dalam Pseudocode	20
4.3. Analisis Desain Solusi Algoritma Greedy	41
BAB V	47
LAMPIRAN	48
DAFTAR PUSTAKA	49

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. Tampilan Game Etimo DIamond

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:
 - a. 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)

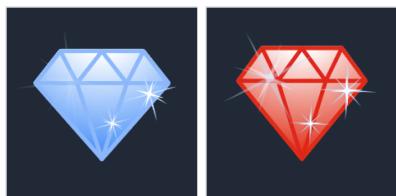
c. Program utama (*main*) dan utilitas lainnya

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>

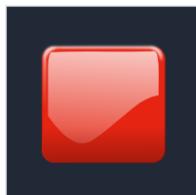
Komponen-komponen dari permainan Diamonds antara lain:

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



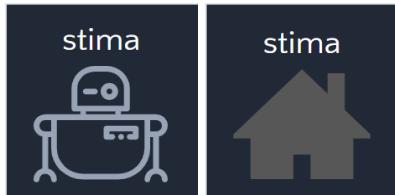
Ketika *red button* ini dilewati/dilangkahi, semua *diamond* (termasuk *red diamond*) akan di-generate kembali pada *board* dengan posisi acak. Posisi *red button* ini juga akan berubah secara acak jika *red button* ini dilangkahi.

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 yang populer digunakan untuk memecahkan persoalan optimasi secara langkah per langkah dengan prinsip “*take what you can get now!*”. Algoritma Greedy mempertimbangkan banyak pilihan dalam setiap langkah. Algoritma Greedy mengambil pilihan terbaik (pilihan yang menjadi optimum lokal pada langkah yang sedang diambil) dengan harapan dapat mengantarkan penyelesaian permasalahan menuju langkah akhir yang bernilai optimum global.

Algoritma Greedy memiliki enam elemen penting, yaitu :

1. Himpunan Kandidat (C)
Himpunan kandidat berisi kandidat yang akan dipilih pada setiap langkah
2. Himpunan Solusi (S)
Himpunan solusi berisi kandidat yang sudah dipilih.
3. Fungsi Solusi
Fungsi solusi menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi.
4. Fungsi Seleksi
Fungsi seleksi memilih kandidat berdasarkan strategi *greedy* tertentu yang bersifat heuristik.
5. Fungsi Kelayakan
Fungsi kelayakan memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi
6. Fungsi Obyektif
Fungsi obyektif memaksimumkan atau meminimumkan hasil dari setiap langkah yang diambil dalam penyelesaian permasalahan.

2.2. Cara Kerja Program Etimo Diamond

Game Etimo Diamond membutuhkan starter pack untuk mulai menjalankan game dan mengimplementasikan bot. Game engine dapat diunduh pada tautan <https://github.com/haziqam/tubes1-IF2211-game-engine/releases/tag/v1.1.0> dengan prasyarat instalasi node.js, docker desktop, dan yarn. Kemudian, bot starter pack dapat diunduh pada tautan <https://github.com/haziqam/tubes1-IF2211-bot-starter-pack/releases/tag/v1.0.1> dengan prasyarat instalasi Python.

Secara garis besar, cara kerja game Etimo Diamond adalah sebagai berikut :

1. 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 dengan diamond merah memiliki 2 poin dan diamond 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. Jika suatu 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 (*tackle*).
7. Apabila waktu seluruh bot telah berakhir, maka permainan berakhir. Score masing-masing pemain akan ditampilkan pada tabel Final Score di sisi kanan layar.

2.3. Alur Menjalankan Game Engine

Langkah-langkah untuk instalasi dan menjalankan *game engine* dari permainan Etimo Diamond :

1. Buka terminal dan masuk ke root directory dari project.
2. Install dependencies menggunakan yarn.
3. Lakukan setup default environment variable.
4. Buka aplikasi docker desktop kemudian lakukan setup local database.
5. Buat direktori build dengan perintah ‘npm run build’.
6. Langkah 1 hingga 4 hanya dilakukan sekali saja.
7. Pada terminal, jalankan runner dengan perintah ‘npm run start’.
8. Klik link localhost yang diberikan di output command dan buka game etimo diamond di platform yang diinginkan.

2.4. Alur Menambahkan Logic Bot dan Menjalankan Bot

Langkah-langkah untuk menambahkan logic bot dan menjalankan bot:

1. Tambahkan file yang berisi logic dari bot yang ingin dimainkan pada folder “tubes1-IF2211-bot-starter-pack-1.0.1/game/logic”
2. Pindah ke file “main.py” yang berada di parent folder “tubes1-IF2211-bot-starter-pack-1.0.1” dan tambahkan logic yang digunakan

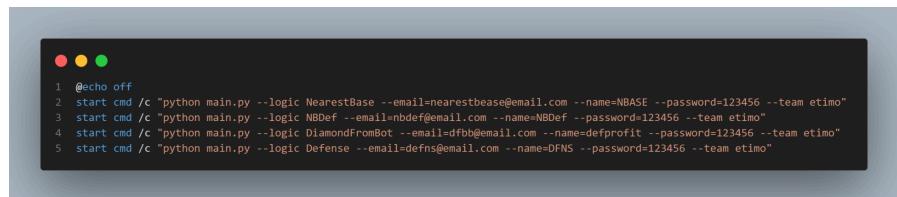
pada bagian CONTROLLERS dengan format “Penamaan Logic : Nama Class Logic”.



```
1  init()
2  BASE_URL = "http://localhost:3000/api"
3  DEFAULT_BOARD_ID = 1
4  CONTROLLERS = {
5      "Random": RandomLogic,
6      "NearestBase": GreedyBase,
7      "DiamondFromBot": DefensiveProfit,
8      "Defense" : defensiveBot,
9      "Attack" : AggresiveBot,
10     "DefProfit" : TryCombine,
11     "NBDef" : GreedyBaseDefense
12     # Tambahkan logic ke CONTROLLERS
13 }
```

Gambar

3. Pindah ke run-bots.bat dan tambahkan bot baru baik dengan menambah jumlah bot yang sudah ada ataupun mengubah identitas bot dan logicnya dengan logic yang baru.



```
1 @echo off
2 start cmd /c "python main.py --logic NearestBase --email=nearestbease@email.com --name=NBASE --password=123456 --team etimo"
3 start cmd /c "python main.py --logic NBDef --email=nbdef@email.com --name=NBDef --password=123456 --team etimo"
4 start cmd /c "python main.py --logic DiamondFromBot --email=dfrb@gmail.com --name=defprofit --password=123456 --team etimo"
5 start cmd /c "python main.py --logic Defense --email=defns@email.com --name=DFNS --password=123456 --team etimo"
```

Gambar

4. Pastikan *game engine* dari Etimo Diamond sudah dijalankan.
5. Buka terminal baru pada folder “tubes1-IF2211-bot-starter-pack-1.0.1”.
6. Pada terminal, jalankan runner dengan perintah “./run-bots.bat”.
7. Program akan menjalankan bot pada board yang tersedia.

BAB III

APLIKASI STRATEGI GREEDY

3.1. Eksplorasi Alternatif Solusi *Greedy*

1. *Greedy by Diamond Nearest Base*

Greedy by Diamond Nearest Base adalah strategi *greedy* yang mengutamakan mengumpulkan *diamond* di dekat *base* milik bot sebanyak-banyaknya dengan tujuan mendapatkan skor secepat mungkin.

a. *Mapping Elemen Greedy*

- Himpunan kandidat : Himpunan *diamond* pada *board*
- Himpunan solusi : Diamond-diamond dengan jarak minimum ke *base* yang memungkinkan terkumpulnya diamond secepat-cepatnya ke *base*.
- Fungsi solusi : Memeriksa apakah diamond-diamond yang dipilih sebagai diamond nearest base berhasil secara efektif mengumpulkan dan mengembalikan diamond ke *base* secepat mungkin.
- Fungsi seleksi : Pilih *diamond* dengan posisi paling dekat dengan *base* milik bot
- Fungsi kelayakan : Bot akan mengambil diamond apabila *inventory*-nya belum penuh (*inventory* berisi kurang dari lima diamond). Untuk bot dengan *inventory* kurang dari lima namun tidak kosong, bot akan mengambil diamond dengan kondisi jarak antara diamond dengan *base* tidak lebih besar dari 1.75 kali jarak antara bot dengan *base*.
- Fungsi objektif: Bot memaksimumkan waktu pengembalian dengan cara mencari diamond yang memiliki jarak terpendek dan menghindari diamond yang posisinya terlalu jauh agar bot dapat mencetak skor secepat-cepatnya dan sebagai mekanisme menghindari risiko tercurinya diamond oleh musuh.

b. Analisis Efisiensi Solusi

Strategi ini efisien dalam mengamankan *diamond* yang berada di *inventory* dari bot untuk mencetak skor dengan cepat, utamanya jika terdapat banyak *diamond* di sekitar *base*. Efisiensi dalam mencetak skor lebih tinggi karena jarak aman yang ditentukan membuat bot lebih mempertimbangkan pengamanan *diamond* dengan memprioritaskan kembali ke bot apabila *diamond* terdekat memiliki jarak yang lebih jauh.

c. Analisis Efektivitas Solusi

Strategi ini efektif pada kondisi masih banyak *diamond* di sekitar *base* seperti pada saat awal permainan atau setelah proses regenerasi *diamond*. Hal

ini disebabkan bot terfokus untuk mengumpulkan *diamond* sebanyak mungkin dan langsung pulang ke *base* sehingga akumulasi skor cepat diraih. Namun, strategi ini kurang efektif pada pertengahan permainan dimana sebaran *diamond* berada jauh dari *base* dan sangat tidak mempertimbangkan faktor lain dalam permainan.

2. *Greedy by Diamond Nearest Bot*

Greedy by Nearest Bot adalah strategi *greedy* yang mengutamakan mengumpulkan *diamond* di dekat bot sebanyak-banyaknya dengan tujuan mendapatkan skor setinggi mungkin.

a. *Mapping Elemen Greedy*

- Himpunan kandidat : Himpunan *diamond* pada *board*
- Himpunan solusi: Diamond-diamond dengan jarak minimum ke bot yang memungkinkan terkumpulnya diamond sebanyak-banyaknya sebelum kembali ke *base*.
- Fungsi solusi: Memeriksa apakah diamond-diamond yang dipilih dapat terkumpul memenuhi inventory bot dalam waktu singkat sebelum kembali ke *base*.
- Fungsi seleksi: Pilih *diamond* terdekat dari bot dari seluruh himpunan diamond pada *board*.
- Fungsi kelayakan: *Diamond* akan diambil oleh bot apabila *inventory* bot belum penuh (*inventory* berisi kurang dari lima diamond).
- Fungsi objektif: Memaksimalkan pengambilan *diamond* dengan jarak minimum dari bot.

b. Analisis Efisiensi Solusi

Strategi *Greedy by Nearest Bot* efisien dalam mengumpulkan banyak diamond dalam waktu singkat. Strategi ini akan menguntungkan apabila bot musuh aksinya tidak condong pada penyerangan terhadap bot lainnya. Namun, bot memiliki kekurangan efisiensi dalam mencetak skor karena adanya kemungkinan *diamond* berada jauh dari *base* sehingga bot harus memiliki *effort* untuk kembali ke *base* dengan ancaman dari bot lain juga

c. Analisis Efektivitas Solusi

Strategi *Greedy by Nearest Bot* efektif apabila jumlah *diamond* di sekitar bot banyak dan berdekatan. Bot akan mendeteksi banyaknya diamond yang tersebar di sekitarnya dan mengambil *diamond* dengan jarak minimum. Bot akan mengambil *diamond* hingga kapasitas *inventory* terpenuhi kemudian kembali ke *base* untuk mencetak skor. Namun, strategi ini kurang efektif apabila bot berada di tengah-tengah bot lainnya yang mengoptimalkan mode penyerangan. Bot cenderung mengalami kerugian karena besarnya potensi bot mengalami kehilangan banyak *diamond* yang sudah dikumpulkan sebelum kembali ke *base*.

3. *Greedy by Defense*

Greedy by Defense adalah strategi *greedy* yang mengutamakan untuk menjauhi bot lawan dengan tujuan bertahan dan menghindari *diamond* di-*tackle*. Pada saat eksplorasi, implementasi *Greedy by Defense* pertama kami menggunakan 12 titik koordinat di sekitar bot untuk mengecek apakah terdapat bot lawan atau tidak. Selain itu, terdapat array yang menyimpan variabel bertipe boolean representasi arah gerak yang dapat dilakukan oleh bot. Dalam menentukan arah gerak akhir yang dipilih oleh bot, kami masih menggunakan *looping* pada array untuk mengecek nilainya satu per satu.

a. *Mapping Elemen Greedy*

- Himpunan kandidat: Himpunan posisi pada board.
- Himpunan solusi: Posisi-posisi pada board yang membuat bot menghindari bot musuh.
- Fungsi solusi: Memeriksa apakah posisi-posisi yang dipilih dapat berhasil membuat bot menghindari bot musuh.
- Fungsi seleksi: Memilih kandidat posisi yang aman dari keberadaan bot musuh sebagai posisi untuk pengambilan langkah berikutnya.
- Fungsi kelayakan: Posisi tidak melebihi batas panjang maupun lebar dari board (posisi valid).
- Fungsi objektif: Meminimumkan kejadian terserangnya bot oleh bot musuh.

b. Analisis Efisiensi Solusi

Strategi ini dinilai kurang efisien karena iterasi *looping* yang terlalu banyak dan dapat dipersingkat, serta pengecekan titik koordinat yang banyak sehingga bot akan menjauhi bot lawan yang masih dianggap tidak terlalu berbahaya.

c. Analisis Efektivitas Solusi

Strategi ini dinilai kurang efektif karena hanya berfokus dalam menghindari bot lain, tidak berfokus mengejar *diamond* yang dekat dengan bot, jarak bot dengan *base*, dan keberadaan teleporter.

4. *Greedy by Attack with Most Profitable Diamond*

Greedy by Attack with Most Profitable Diamond adalah strategi *greedy* yang mengutamakan untuk memakan bot lawan terdekat dan jika tidak akan berfokus dalam mengambil *diamond*. Algoritma yang digunakan dalam strategi ini menggunakan proses yang sama dengan *Greedy by Defense* dipadukan dengan *Greedy by Most Profitable Diamond*, tetapi dengan perbedaan di mana saat terdeteksi musuh pada *Greedy by Defense* bot memilih untuk menjauh, pada strategi ini dibuat agar bot menyerang atau men-*tackle* musuh.

a. *Mapping Elemen Greedy*

- Himpunan kandidat: Himpunan posisi pada board.
- Himpunan solusi: Posisi-posisi pada board yang membuat bot mendekati bot musuh.
- Fungsi solusi: Memeriksa apakah posisi yang akan diambil dapat membuat bot berhasil mendekati bot musuh untuk melakukan penyerangan.
- Fungsi seleksi: Pilih posisi yang terdeteksi sebagai posisi ditemukannya keberadaan bot musuh.
- Fungsi kelayakan: Posisi tidak melebihi batas panjang maupun lebar dari board (posisi valid).
- Fungsi objektif: Memaksimumkan kejadian perlakuan penyerangan pada bot musuh.

b. Analisis Efisiensi Solusi

Strategi ini dinilai kurang efisien karena bot akan menyerang bot lawan manapun tanpa adanya konsiderasi lain sehingga berpotensi kehilangan *diamond* ketimbang mengambil *diamond* milik bot lawan.

c. Analisis Efektivitas Solusi

Strategi ini dinilai kurang efektif karena terlalu berfokus dalam pengejaran bot lawan sehingga besar potensi bot untuk terus mengejar bot lawan hingga bot lawan berada di luar radius yang sudah ditentukan dan mengakibatkan bot kehilangan banyak kesempatan untuk mengambil *diamond*. Namun, strategi ini diuntungkan ketika *diamond* pada *inventory* bot mengungguli bot lain atau jarak dengan bot lain cukup dekat.

5. *Greedy by Diamond Nearest Base with Defense*

Greedy by Nearest Base with Defense adalah strategi *greedy* yang mengutamakan mengumpulkan *diamond* di dekat *base* milik bot sebanyak-banyaknya dengan tujuan mendapatkan skor secepat mungkin. Strategi ini merupakan modifikasi dari *Greedy by Diamond Nearest Base* berupa tambahan metode pertahanan dari ancaman bot lawan.

a. *Mapping Elemen Greedy*

- Himpunan kandidat : Himpunan *diamond* pada *board*.
- Himpunan solusi: Diamond-diamond dengan jarak minimum ke *base* yang memungkinkan terkumpulnya diamond secepat-cepatnya ke *base*.
- Fungsi solusi: Memeriksa apakah diamond-diamond yang dipilih sebagai diamond nearest base berhasil secara efektif mengumpulkan dan mengembalikan diamond ke *base* secepat mungkin.
- Fungsi seleksi: Pilih *diamond* dengan posisi paling dekat dengan *base* milik bot dan jarak terjauh dari musuh.

- Fungsi kelayakan: Diamond akan diambil oleh bot apabila *inventory*-nya belum penuh (*inventory* berisi kurang dari lima diamond). Untuk bot dengan *inventory* kurang dari lima namun tidak kosong, bot akan mengambil diamond dengan kondisi jarak antara diamond dengan base tidak lebih besar dari 1.75 kali jarak antara bot dengan base.
- Fungsi objektif: Memaksimalkan posisi aman bot dan memaksimalkan efisiensi pengembalian diamond ke base untuk mencetak skor.

b. Analisis Efisiensi Solusi

Strategi ini efisien dalam mengamankan diamond pada kondisi banyaknya *diamond* di sekitar *base* dengan lebih aman. Bot dapat mengembalikan diamond secepat-cepatnya ke base untuk mencetak skor. Namun, dengan penambahan mekanisme *defense*, efisiensi bot dalam mengumpulkan diamond pada kondisi sudah tidak adanya *diamond* di sekitar *base* akan berkurang karena selain mempertimbangkan jauhnya jarak *diamond* dari base, bot juga cenderung punya fokus untuk menjauh dari bot lain dan sulit menjangkau *diamond* terdekat. *Diamond* juga dikembalikan dalam jumlah yang lebih sedikit di kondisi ini.

c. Analisis Efektivitas Solusi

Sama seperti *Greedy by Diamond Nearest Base with Defense*, strategi ini efektif dan lebih cepat mendapat skor pada kondisi masih banyak *diamond* di sekitar *base*. Walaupun sudah sedikit lebih baik dengan mengimplementasikan strategi *defense*, strategi ini dinilai masih kurang efektif dalam mendapatkan *diamond* secara *general*. Hal ini disebabkan pengumpulan *diamond* diprioritaskan pada *range* sekitar *base*, tanpa mempertimbangkan faktor kedekatan *diamond* dengan bot pemain.

6. *Greedy by Most Profitable Diamond*

Greedy by Most Profitable Diamond adalah strategi *greedy* yang mengutamakan mengumpulkan *diamond* dengan skor “keuntungan” tertinggi. Keuntungan tersebut dikalkulasi dengan mempertimbangkan berbagai aspek, di antara lain:

- Poin *diamond* : *diamond* biru memiliki nilai poin 1, *diamond* merah bernilai 2. Semakin besar nilai *diamond*, semakin tinggi skor keuntungannya
- Jarak *diamond* dengan bot : Semakin kecil jarak *diamond* dengan bot, semakin tinggi skor keuntungannya
- Jarak *diamond* dengan *base* : Semakin kecil jarak *diamond* dengan *base*, semakin tinggi skor keuntungannya karena bot akan lebih cepat mencapai *base* dan meraih skor

- Jarak *diamond* dengan bot lawan : Semakin besar jarak bot dengan bot lawan, semakin berkurang ancaman dalam meraih *diamond* tersebut sehingga semakin tinggi skor keuntungan *diamond*-nya

Dengan demikian, dapat ditentukan rumus untuk menghitung keuntungan *diamond* sebagai berikut:

$$\text{profit} = (\text{distToEnemy} + \text{diamond.properties.points}) / (\text{distToBot} + \text{distToBase})$$

a. *Mapping Elemen Greedy*

- Himpunan kandidat: Himpunan *diamond* pada *board*.
- Himpunan solusi: Diamond-diamond dengan *profit* terbesar yang memungkinkan terkumpulnya diamond secara aman dan terkumpulnya skor dengan cepat.
- Fungsi solusi: Memeriksa apakah diamond-diamond yang dipilih sebagai most profitable diamond berhasil mengumpulkan diamond sebanyak dan secepat mungkin ke base serta relatif aman dari potensi serangan musuh akibat dekatnya letak diamond dengan bot musuh.
- Fungsi seleksi: Pilih *diamond* yang memiliki nilai *profit* terbesar berdasarkan perhitungan rumus *profit* yang mempertimbangkan jarak diamond ke bot musuh, besar poin yang dimiliki *diamond*, jarak *diamond* ke bot, dan jarak *diamond* ke base.
- Fungsi kelayakan: *Diamond* akan diambil oleh bot apabila inventory bot belum penuh.
- Fungsi objektif: Bot memaksimalkan kondisi pengambilan diamond dengan mempertimbangkan besar *profit* yang maksimal.

b. Analisis Efisiensi Solusi

Strategi ini tepat digunakan untuk mengumpulkan banyak diamond bernilai besar yang relatif aman dari keberadaan bot musuh serta dekat dengan base.

c. Analisis Efektivitas Solusi

Strategi ini cukup efektif karena telah mempertimbangkan banyak faktor. Bot ini bahkan mengungguli permainan ketika diadu dengan mayoritas bot lain yang menerapkan mekanisme pertahanan (karena bot lawan tersebut akan menghindar dari bot ini dan “membuka jalan” bagi bot untuk mengambil *diamond* yang diincar sehingga lebih cepat meraih skor). Namun, ketika diadu dengan bot yang tidak dominan strategi pertahanannya (dominan penyerangan, misalnya), bot akan rentan dimakan. Kemudian, ketika penempatan *base* pada *board* terpusat pada suatu titik, bot ini akan kesulitan untuk mencari celah dan meraih targetnya serta sangat rentan atas bot-bot di sekitarnya. Strategi ini juga kurang efektif ketika masih ada *diamond* dengan *range* jarak yang cukup dekat dari *base* dan juga terdapat bot lawan di sekitarnya. Dalam beberapa kasus, bot akan “enggan” mengambil *diamond* di dekat *base* karena terancam bot lawan dalam radius dekat. Padahal *diamond*

tersebut bisa saja mempercepat bot dalam meraih skor karena bot semakin cepat pulang ke *base*.

7. *Greedy by Most Profitable Diamond with Defense*

Greedy by Most Profitable Diamond with Defense adalah modifikasi dari strategi *Greedy by Most Profitable Diamond* dengan tambahan fitur pertahanan dari bot lawan agar menghindari *diamond* pada *inventory* di-*tackle*. Kami juga menambahkan fitur deteksi *teleporter* sehingga bot bisa meraih suatu *diamond* lebih cepat dengan jalur alternatif melalui *teleporter*.

Dengan dukungan sistem defense ini, konsiderasi jarak antara diamond ke enemy dapat diabaikan saat melakukan penghitungan *profit* dari suatu diamond. Oleh karena itu, formula *profit* mengalami modifikasi menjadi :

$$\text{profit} = (\text{diamond.properties.points}) / (\text{distToBot} + \text{distToBase})$$

Dari hasil eksplorasi, ditemukan tiga cara implementasi konsep *defense* dalam mencari cara yang efisien dan efektif untuk menentukan arah gerak yang aman untuk bot. Implementasi pertama, merupakan implementasi yang masih sama dengan implementasi strategi *Greedy by Defense* dan hanya menambahkan aspek perhitungan *profit* dari strategi *Greedy by Most Profitable Diamond*. Setelah menganalisis kembali dan melakukan *testing*, ditemukan bahwa menyimpan koordinat pada suatu array dan representasi arah jalan yang akan diambil dengan variabel lain (misalnya boolean atau integer) kurang efisien dan memperlambat *reaction time* bot yang selanjutnya diperbaiki di implementasi kedua.

Pada implementasi kedua, *logic* langsung menyimpan arah gerak dalam array dan mempersempit *range* koordinat yang diperiksa dari 12 kotak di sekitar bot menjadi 8 kotak. Selain itu, mulai digunakannya fungsi *remove()* untuk menggantikan *filter()* dalam mendapatkan array dari arah gerak akhir yang akan dipakai. Permasalahan yang masih belum di-*handle* pada implementasi kedua ini adalah saat bot berputar-putar di *track* yang sama akibat posisi *teleporter* serta potensi penggunaan *teleporter* yang belum diimplementasikan. Oleh sebab itu, kami melakukan implementasi baru untuk meningkatkan efisiensi dan efektifitas strategi ini.

Implementasi ketiga merupakan implementasi final sebelum ditambahkannya kapabilitas *attacking* pada strategi *Greedy by Most Profitable Diamond with Defense and Attack* yang merupakan modifikasi lebih lanjut dari strategi ini. Pada implementasi ketiga, potensi *teleporter* diperhitungkan untuk membantu bot sampai ke *diamond* maupun *base*, serta memastikan bahwa bot akan menunggu dibandingkan terus berputar-putar saat *teleporter* membentuk *track* atau menghalangi arah jalannya bot. Selain itu, proses pengecekan apakah bot musuh terdapat dalam *range* atau tidak dilakukan perubahan, karena terdapat lebih sedikit musuh dibandingkan jumlah kotak di sekitar bot. Pengecekan diubah ke metode perhitungan jarak secara langsung dan menentukan arah mana yang tidak diambil oleh bot. Implementasi ini merupakan implementasi paling efisien jika dibandingkan sebelumnya.

a. *Mapping Elemen Greedy*

- Himpunan kandidat: Himpunan *diamond* pada *board*.
- Himpunan solusi: Diamond-diamond dengan *profit* terbesar yang memungkinkan terkumpulnya banyak diamond secara aman dan cepat ke base.
- Fungsi solusi: Memeriksa apakah *diamond-diamond* yang dipilih sebagai *most profitable diamond* berhasil mengumpulkan diamond sebanyak dan secepat mungkin ke base serta relatif aman dari potensi serangan musuh akibat dekatnya letak diamond dengan bot musuh.
- Fungsi seleksi: Pilih *diamond* yang memiliki nilai *profit* terbesar berdasarkan perhitungan rumus *profit* yang mempertimbangkan besar poin yang dimiliki *diamond*, jarak *diamond* ke bot, dan jarak *diamond* ke base. Pada konsiderasi pemilihan *diamond*, keberadaan musuh berakibat pada menjauhnya bot dari keberadaan bot musuh pada saat itu.
- Fungsi kelayakan: Diamond akan diambil oleh bot apabila *inventory* bot belum penuh.
- Fungsi objektif: Memaksimumkan pengambilan *diamond* dengan mengambil *diamond* berdasarkan *profit* terbesar diamond yang mempertimbangkan

b. Analisis Efisiensi Solusi

Strategi ini cukup efisien dalam mengumpulkan *diamond*, tetapi membutuhkan banyak langkah untuk melakukan berbagai pengecekan, baik itu posisi *teleporter*, posisi bot lawan, posisi target, serta jalan menuju target yang dipilih. Hal ini tentunya cukup berpengaruh terhadap *reaction time* bot yang menggunakan strategi ini.

c. Analisis Efektivitas Solusi

Strategi ini cukup efektif dalam mengumpulkan *diamond* berdasarkan faktor point diamond, jarak diamond ke base, dan jarak diamond ke bot (baik jarak bot secara langsung dengan *diamond* maupun jarak bila melalui *teleporter*). Kelemahan dari strategi ini adalah ketika terdapat sepasang *teleporter* yang berdekatan dengan *diamond* target, bot mungkin terjebak di antara 2 *teleporter* sambil menunggu hingga konfigurasi *board* berubah. Hal ini disebabkan bot akan menghindari memasuki *teleporter* jika bot tidak ingin mengejar *diamond* lewat *teleporter*. Ketika 2 *teleporter* berdekatan, bot akan menghindari keduanya dan terjebak. Kelemahan ini diminimalisir dengan memberi persentase yang lebih kecil untuk perhitungan jarak *teleporter* agar bot tidak terjebak. Pertimbangan lain untuk memperkecil persentase perhitungan bagi *teleporter* yaitu jumlah *teleporter* di *board* yang relatif sedikit(biasanya hanya 1 pasang) sehingga dinilai tidak begitu signifikan efeknya dibandingkan faktor lain.

8. Greedy by Most Profitable Diamond with Defense and Attack

Greedy by Most Profitable Diamond with Defense and Attack adalah strategi *greedy* yang mengutamakan untuk mengambil *diamond* dengan profit tertinggi sekaligus menerapkan strategi pertahanan dan penyerangan. Strategi ini merupakan gabungan dan optimasi dari strategi *greedy* yang telah disebutkan sebelumnya. Penjelasan mengenai strategi ini akan dijelaskan pada bagian 3.2.

3.2. Algoritma *Greedy* yang Dipilih

Berdasarkan berbagai algoritma *greedy* yang terdapat pada bagian 3.1, diputuskan untuk menggunakan algoritma *greedy* yang dinilai paling stabil dan mempertimbangkan berbagai komponen pada permainan, seperti jarak *diamond* dengan bot, jarak *diamond* dengan *teleporter*, dan jarak *diamond* dengan *base* dengan pertahanan, dan penyerangan juga, yaitu Algoritma *Greedy by Most Profitable Diamond with Defense and Attack*. Adapun alur algoritma tersebut adalah sebagai berikut:

1. Cegah pemilihan target di luar range *board*
2. Deteksi *diamond*, *teleporter*, *base* pada permainan
3. Pilih *base* sebagai target jika *diamond* pada *inventory* sudah 5, atau *diamond* pada *inventory* ada 4 namun *diamond* target adalah red, atau *diamond* pada *inventory* ada 4 namun bot berada pada radius 24 kotak di sekitar *base* atau berada pada radius $\sqrt{8}$ dari *base* (sehingga bot tidak berkeliling membuang waktu untuk mengejar 1 *diamond* yang lebih jauh)
4. Jika tidak memenuhi kondisi pada nomor 3, pilih target berupa *diamond* dengan profit terbesar, bisa langsung atau bisa melalui *teleporter* (bila target akan dikejar secara langsung, hindari *teleporter*)
5. Jika terdapat bot lawan pada radius 8 kotak sekitar bot (jarak $1,5 = \sqrt{2}$ mendekati $\sqrt{8}$) dan lawan memiliki lebih sedikit *diamond*, tabrak bot lawan, jika tidak hindari bot tersebut.

a. Mapping Elemen *Greedy*

- Himpunan kandidat : Himpunan posisi yang dapat memberikan *diamond* bagi bot.
- Himpunan solusi : Posisi yang didapat dari perhitungan profit *diamond* atau posisi musuh untuk bot melakukan attack hingga terpenuhi kondisi bot harus kembali ke *base* (*inventory* sudah penuh atau *inventory* sudah berisi 4 *diamond* namun *diamond* yang menjadi target berikutnya adalah red *diamond* atau *inventory* sudah berisi 4 *diamond* namun bot berada dekat *base*).
- Fungsi solusi : Memeriksa apakah posisi *diamond* yang dipilih sebagai *most profitable diamond* ataupun posisi bot musuh untuk diberlakukan attack berhasil mengumpulkan *diamond* hingga bot memenuhi kondisi harus kembali ke *base* dengan seefisien mungkin.
- Fungsi seleksi : Pilih posisi dari *diamond* yang memiliki nilai *profit* terbesar berdasarkan perhitungan rumus *profit* yang mempertimbangkan

besar poin yang dimiliki *diamond*, jarak *diamond* ke bot, dan jarak *diamond* ke base. Apabila didapatkan keberadaan musuh di posisi dekat bot dengan *inventory* yang besarnya di bawah bot, pilih posisi bot musuh tersebut untuk melakukan mekanisme attack agar didapatkan *diamond* sebanyak-banyaknya secara efisien.

- Fungsi kelayakan : Diamond akan diambil oleh bot apabila kondisi bot belum memenuhi kondisi bot harus kembali ke base.
- Fungsi objektif : Memaksimumkan peluang didapatkannya *diamond-diamond* oleh bot, baik itu berdasarkan pengambilan *diamond* yang mengkonsiderasi profit maupun berdasarkan pengambilan *diamond* dari bot lain dengan menggunakan mekanisme attack.

b. Analisis Efisiensi Solusi

Tahap-tahap algoritma yang diimplementasikan pada method *next_move* dimulai dari iterasi array *direction* yang merepresentasikan 4 kemungkinan pergerakan bot, tujuannya untuk mencegah pergerakan bot ke luar *board*. Kompleksitas untuk tahap ini adalah $O(n)$. Kemudian, objek *diamond* dan *teleporter* pada *board* dideteksi dan dimasukkan ke dalam list dan map objeknya masing-masing, kompleksitas waktunya $O(n)$ dengan n merupakan jumlah objek dalam permainan.

Kompleksitas waktu kalkulasi jarak antara 2 posisi dengan Euclidean Distance menggunakan method *math.dist* membutuhkan kompleksitas $O(1)$. pada method *calculateDistance*, *teleporter* terdekat dari bot kemudian ditentukan dengan kompleksitas waktu $O(n)$ dengan n merupakan jumlah *teleporter*. Posisi *base* dari bot dan jumlah *diamond* pada *inventory* ditentukan dengan kompleksitas $O(1)$.

Method *getMostProfitable* yang menentukan *diamond* paling profit dimulai dengan proses penentuan jarak *diamond-bot*, *diamond-teleporter*, *teleporter-bot*, *diamond-base*, dan kalkulasi profit, kompleksitas waktunya masing-masing $O(1)$. Proses ini diulang untuk n objek *diamond* pada *board*, sehingga kompleksitas *getMostProfitable* adalah $O(n)$.

Deteksi bot musuh membutuhkan kompleksitas $O(n)$ dengan n adalah jumlah bot pada *board*. Penentuan arah gerak untuk menghindari *teleporter* secara situasional dilakukan dengan kompleksitas $O(n)$ dengan n adalah jumlah arah pada array *direction*. Kemudian, penentuan arah gerak menuju *goal position* dengan function *clamp* adalah $O(1)$ karena hanya membandingkan nilai integer. Pengecekan posisi bot musuh serta mekanisme penyerangan musuh membutuhkan kompleksitas $O(n)$ dengan n jumlah bot musuh. Kompleksitas waktu total untuk strategi ini adalah $O(n)$.

c. Analisis Efektivitas Solusi

Strategi ini efektif dan relatif stabil dibandingkan strategi lainnya karena telah mempertimbangkan berbagai faktor seperti posisi *teleporter*, posisi dan poin *diamond*, posisi *base*, pertahanan dari lawan dengan *diamond* lebih banyak, penyerangan lawan dengan *diamond* lebih sedikit pada radius tertentu,

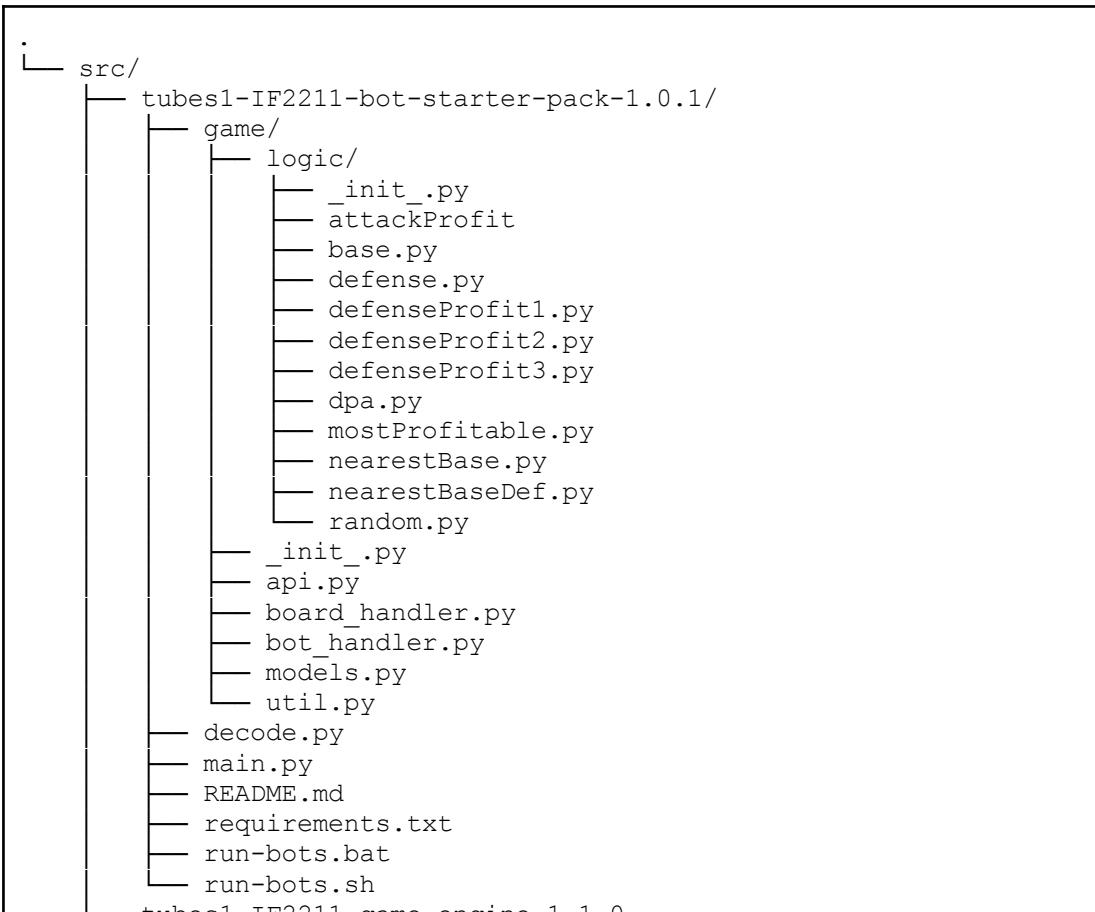
menghindar dari sudut *board*, dan pertimbangan lainnya. Namun, strategi *greedy* tidak selalu memberikan hasil optimal. Detail mengenai hal ini akan dijelaskan pada bab 4

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Struktur Data Program

Struktur folder program dalam permainan Diamonds adalah sebagai berikut:



Bot yang kami bangun berada pada direktori
`src\tubes1-IF2211-bot-starter-pack-1.0.1\game\logic`.

4.2. Implementasi dalam Pseudocode

4.2.1. dpa.py

```
class DPA(BaseLogic):
    goal_position <- None
    aim_for_teleporter <- False
    diamondtargetpoint <- None

    function next_move(board_bot: GameObject, board: Board) -> Tuple[int, int]:
        props <- board_bot.properties
        current_position <- board_bot.position

        direction <- [(1,0), (0,1), (-1,0), (0,-1)] # East,
        South, West, North
```

```

# handling edge positions
if current_position.x == 0:
    direction.remove((-1,0))
elif current_position.x == board.width-1:
    direction.remove((1,0))
if current_position.y == 0:
    direction.remove((0,-1))
elif current_position.y == board.height-1:
    direction.remove((0,1))

diamonds <- []
teleporters <- {}

for object in board.game_objects:
    if object.type == "DiamondGameObject":
        diamonds.append(object)
    elif object.type == "TeleportGameObject":
        if object.properties.pair_id not in
teleporters:
            teleporters[object.properties.pair_id] <-
[]

teleporters[object.properties.pair_id].append(object)

nearestTeleporter <- None
minDist <- inf

for key in teleporters:
    for teleporter in teleporters[key]:
        distTeleporterToBot <-
calculateDistance(board_bot.position,teleporter.position)
        if (distTeleporterToBot < minDist):
            nearestTeleporter <- teleporter
            minDist <- distTeleporterToBot

nearestTeleporterIdx <-
teleporters[nearestTeleporter.properties.pair_id].index(nearestTeleporter)

base <- board_bot.properties.base

if (props.diamonds == 5 or (props.diamonds == 4 and
(diamondtargetpoint == 2 or dist([base.x, base.y],
[current_position.x, current_position.y])<2.9))) and
current_position != board_bot.properties.base:
    goal_position <- base
else:
    if (nearestTeleporterIdx == 0):
        nearestTeleporterPair <-
teleporters[nearestTeleporter.properties.pair_id][1]
    else:
        nearestTeleporterPair <-
teleporters[nearestTeleporter.properties.pair_id][0]

target <- getMostProfitable(board_bot, diamonds,
nearestTeleporter, nearestTeleporterPair)
    goal_position <- target.position if not
aim_for_teleporter else nearestTeleporter.position

enemies <- [bot for bot in board.bots if

```

```

bot.properties.name != board_bot.properties.name]

        if not aim_for_teleporter:
            if current_position.x+1 ==
nearestTeleporter.position.x and current_position.y ==
nearestTeleporter.position.y:
                direction.remove((1,0))
            elif current_position.x-1 ==
nearestTeleporter.position.x and current_position.y ==
nearestTeleporter.position.y:
                direction.remove((-1,0))
            if current_position.y+1 ==
nearestTeleporter.position.y and current_position.x ==
nearestTeleporter.position.x:
                direction.remove((0,1))
            elif current_position.y-1 ==
nearestTeleporter.position.y and current_position.x ==
nearestTeleporter.position.x:
                direction.remove((0,-1))

            x_clamp <- clamp(goal_position.x-current_position.x,
-1, 1)
            y_clamp <- clamp(goal_position.y-current_position.y,
-1, 1)

            for enemy in enemies:
                if dist([enemy.position.x, enemy.position.y],
[current_position.x, current_position.y]) <= 1.5:
                    enemy_x <-
clamp(enemy.position.x-current_position.x, -1, 1)
                    enemy_y <-
clamp(enemy.position.y-current_position.y, -1, 1)

                    if props.diamonds > enemy.properties.diamonds:
                        if enemy.position.x == current_position.x
and enemy_y == y_clamp:
                            return 0, y_clamp
                        elif enemy.position.y ==
current_position.y and enemy_x == x_clamp:
                            return x_clamp, 0
                        else:
                            if enemy_x != 0 and (enemy_x, 0) in
direction:
                                direction.remove((enemy_x, 0))
                            if enemy_y != 0 and (0, enemy_y) in
direction:
                                direction.remove((0, enemy_y))

                    if len(direction) == 0:
                        if x_clamp != 0:
                            y_clamp <- 0
                        return x_clamp, y_clamp
                    else:
                        if (x_clamp,0) in direction:
                            return x_clamp, 0
                        elif (0,y_clamp) in direction:
                            return 0, y_clamp
                        else:
                            return direction[0]

```

```

        function calculateDistance(position1: Position, position2:
Position):
            return dist([position2.x,
position2.y], [position1.x,position1.y])

        function getMostProfitable(bot: GameObject, diamonds:
List[GameObject], nearestTeleporter:GameObject,
nearestTeleporterPair:GameObject) -> GameObject:
            maxProfit <- 0
            selectedDiamond <- None

            for diamond in diamonds:
                realdistToBot <-
calculateDistance(bot.position,diamond.position)
                altDistToBot <-
calculateDistance(nearestTeleporter.position,bot.position) +
calculateDistance(nearestTeleporterPair.position,diamond.position)

                if (realdistToBot > 1.6 * altDistToBot):
                    aim_for_teleporter <- True
                    distToBot <- altDistToBot
                else:
                    aim_for_teleporter <- False
                    distToBot <- realdistToBot

                distToBase <-
calculateDistance(bot.properties.base,diamond.position)
                profit <- ((diamond.properties.points)/(distToBot
+ distToBase))

                if (profit >= maxProfit):
                    selectedDiamond <- diamond
                    maxProfit <- profit
                    diamondtargetpoint <-
diamond.properties.points

            return selectedDiamond

```

4.2.2. defense.py

```

class DefensiveLogic(BaseLogic):
    directions <- [(1, 0), (0, 1), (-1, 0), (0, -1)]
    goal_position <- None
    current_direction <- 0

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

        if props.diamonds == 5:
            base <- board_bot.properties.base
            goal_position <- base
            print("Moving to base")
        else:
            goal_position <- None

```

```

current_position <- board_bot.position
if goal_position:
    print("Moving to goal")
    enemies_coordinate <- [bot.position for bot in
board.bots if bot.properties.name !=
board_bot.properties.name]
    for enemy in enemies_coordinate:
        print(f"Enemy at {enemy.x}, {enemy.y}")

dangerous_coordinate <- []

for i in range(-2, 3):
    for j in range(-2, 3):
        if abs(i)+abs(j) <= 2 and i != 0 and j != 0:
            dangerous_coordinate.append(Position(current_position.x+i,
current_position.y+j))
        else:
            continue

threat <- list(filter(lambda x: x in
enemies_coordinate, dangerous_coordinate))
if len(threat) != 0:
    move_direction <- [True, True, True, True] #
right, up, left, down
    for i in range(len(threat)):
        print("x", i, "=", threat[i].x)
        print("y", i, "=", threat[i].y)
        if threat[i].x > current_position.x:
            move_direction[0] <- False
        elif threat[i].x < current_position.x:
            move_direction[2] <- False
        if threat[i].y > current_position.y:
            move_direction[1] <- False
        elif threat[i].y < current_position.y:
            move_direction[3] <- False

if True in move_direction:
    move <- move_direction.index(True)
    if move == 0:
        delta_x, delta_y <- 1, 0
    elif move == 1:
        delta_x, delta_y <- 0, 1
    elif move == 2:
        delta_x, delta_y <- -1, 0
    elif move == 3:
        delta_x, delta_y <- 0, -1
    else:
        delta_x, delta_y <- get_direction(
            current_position.x,
            current_position.y,
            goal_position.x,
            goal_position.y,
        )
else:
    delta_x, delta_y <- get_direction(
        current_position.x,
        current_position.y,

```

```

        goal_position.x,
        goal_position.y,
    )

else:
    print("Roaming around")
    enemies_coordinate <- [bot.position for bot in
board.bots if bot.properties.name !=
board_bot.properties.name]
    for enemy in enemies_coordinate:
        print(f"Enemy at {enemy.x}, {enemy.y}")

dangerous_coordinate <- []

for i in range(-2, 3):
    for j in range(-2, 3):
        if abs(i)+abs(j) <= 2 and i != 0 and j != 0:
            dangerous_coordinate.append(Position(current_position.x+i,
current_position.y+j))
        else:
            continue

threat <- list(filter(lambda x: x in
enemies_coordinate, dangerous_coordinate))
if len(threat) != 0:
    move_direction <- [True, True, True, True] # right, up, left, down
    for i in range(len(threat)):
        if threat[i].x > current_position.x:
            move_direction[0] <- False
        elif threat[i].x < current_position.x:
            move_direction[2] <- False
        if threat[i].y > current_position.y:
            move_direction[1] <- False
        elif threat[i].y < current_position.y:
            move_direction[3] <- False

if True in move_direction:
    move <- move_direction.index(True)
    if move == 0:
        delta_x, delta_y <- 1, 0
    elif move == 1:
        delta_x, delta_y <- 0, 1
    elif move == 2:
        delta_x, delta_y <- -1, 0
    elif move == 3:
        delta_x, delta_y <- 0, -1
    else:
        # Roam around
        delta <- directions[current_direction]
        delta_x <- delta[0]
        delta_y <- delta[1]
        if random.random() > 0.6:
            current_direction <-
(current_direction + 1) % len(
            directions
        )

```

```
    return delta_x, delta_y
```

4.2.3. attackProfit.py

```
class AttackProfitLogic(BaseLogic):
    goal_position <- None
    aim_for_teleporter <- False
    diamondtargetpoint <- None

    function next_move(board_bot: GameObject, board: Board) ->
        Tuple[int, int]:
        props <- board_bot.properties
        current_position <- board_bot.position

        direction <- [(1,0), (-1,0), (0,1), (0,-1)]
        # handling edge
        if current_position.x == 0:
            direction.remove((-1,0))
        elif current_position.x == board.width-1:
            direction.remove((1,0))
        if current_position.y == 0:
            direction.remove((0,-1))
        elif current_position.y == board.height-1:
            direction.remove((0,1))

        diamonds <- []
        teleporters <- {}
        for object in board.game_objects:
            if object.type == "DiamondGameObject":
                diamonds.append(object)
            elif object.type == "TeleportGameObject":
                if object.properties.pair_id not in
                    teleporters:
                    teleporters[object.properties.pair_id] =
                    []
            teleporters[object.properties.pair_id].append(object)

            nearestTeleporter <- None
            minDist <- inf
            for key in teleporters:
                for teleporter in teleporters[key]:
                    distTeleporterToBot <-
                        calculateDistance(board_bot.position,teleporter.position)
                    if (distTeleporterToBot < minDist):
                        nearestTeleporter <- teleporter
                        minDist <- distTeleporterToBot
            nearestTeleporterIdx <-
                teleporters[nearestTeleporter.properties.pair_id].index(nearestTeleporter)

            if (props.diamonds == 5 or (props.diamonds == 4 and
                diamondtargetpoint == 2)) and current_position !=
                board_bot.properties.base:
                base <- board_bot.properties.base
                goal_position <- base

            else:
```

```

        if (nearestTeleporterIdx == 0):
            nearestTeleporterPair <-
teleporters[nearestTeleporter.properties.pair_id][1]
        else:
            nearestTeleporterPair <-
teleporters[nearestTeleporter.properties.pair_id][0]

            target <- getMostProfitable(board_bot, diamonds,
nearestTeleporter, nearestTeleporterPair)
            goal_position <- target.position if not
aim_for_teleporter else nearestTeleporter.position

            enemies_coordinate <- [bot.position for bot in
board.bots if bot.properties.name !=
board_bot.properties.name]

            if not aim_for_teleporter:
                if current_position.x+1 ==
nearestTeleporter.position.x and current_position.y ==
nearestTeleporter.position.y:
                    direction.remove((1,0))
                elif current_position.x-1 ==
nearestTeleporter.position.x and current_position.y ==
nearestTeleporter.position.y:
                    direction.remove((-1,0))
                if current_position.y+1 ==
nearestTeleporter.position.y and current_position.x ==
nearestTeleporter.position.x:
                    direction.remove((0,1))
                elif current_position.y-1 ==
nearestTeleporter.position.y and current_position.x ==
nearestTeleporter.position.x:
                    direction.remove((0,-1))

                for coord in enemies_coordinate:
                    if dist([coord.x, coord.y], [current_position.x,
current_position.y]) <= 1.5: # sqrt(2) = 1.414
                        x <- clamp(coord.x-current_position.x, -1, 1)
                        if x != 0 and (x, 0) in direction:
                            return x, 0
                        y <- clamp(coord.y-current_position.y, -1, 1)
                        if y != 0 and (0, y) in direction:
                            return 0, y

                        x <- clamp(goal_position.x-current_position.x, -1, 1)
                        y <- clamp(goal_position.y-current_position.y, -1, 1)
                        if (x,0) in direction:
                            delta_x <- x
                            delta_y <- 0
                        elif (0,y) in direction:
                            delta_x <- 0
                            delta_y <- y
                        else:
                            delta_x, delta_y <- direction[0]

                        return delta_x, delta_y

        function calculateDistance(position1: Position, position2:
Position):

```

```

        return dist([position2.x,
position2.y], [position1.x,position1.y])

    function getMostProfitable(bot: GameObject, diamonds:
List[GameObject], nearestTeleporter:GameObject,
nearestTeleporterPair:GameObject) -> GameObject:
        maxProfit <- 0

        for diamond in diamonds:
            realdistToBot <-
calculateDistance(bot.position,diamond.position)
            altDistToBot <-
calculateDistance(nearestTeleporter.position,bot.position) +
calculateDistance(nearestTeleporterPair.position,diamond.posit
ion)
            if (realdistToBot > 1.5 * altDistToBot):
                aim_for_teleporter <- True
                distToBot <- altDistToBot
            else:
                aim_for_teleporter <- False
                distToBot <- realdistToBot
            distToBase <-
calculateDistance(bot.properties.base,diamond.position)
            profit <- (diamond.properties.points)/(distToBot +
distToBase)

            if (profit >= maxProfit):
                selectedDiamond <- diamond
                maxProfit <- profit
                diamondtargetpoint <-
diamond.properties.points
            return selectedDiamond

```

4.2.4. nearestBase.py

```

import math
from typing import Optional
from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position
from ..util import get_direction

function distance(object1position, object2position) :
    distance <- math.sqrt(math.pow(abs(object1position.x -
object2position.x), 2) + math.pow(abs(object1position.y -
object2position.y), 2))
    return distance

class NearestBaseLogic(BaseLogic):
    directions <- [(1, 0), (0, 1), (-1, 0), (0, -1)] # valid
position
    goal_position <- None # position ini defaultnya adalah
None jika bot belum mengetahui position si bot, value atribut
ini None,
                                # tapi kalau sudah ada update
pergerakan, value atributnya terunion dengan current position
bot
    current_direction <- 0

```

```

        function next_move(board_bot: GameObject, board: Board) ->
    Tuple[int, int]: # akan mengembalikan data ke mana bot akan
    bergerak
            props <- board_bot.properties
            base <- board_bot.properties.base
            # Analyze new state

            diamond_base_distances <- []
            diamond_list <- board.diamonds
            for diamond in diamond_list :

diamond_base_distances.append(distance(diamond.position,
base))

            min_distance <- diamond_base_distances[0]
            min_distance_index <- 0
            for i, distance_info in
enumerate(diamond_base_distances) :
                if distance_info <= min_distance :
                    min_distance <- distance_info
                    min_distance_index <- i

            current_position <- board_bot.position
            # Proses menentukan goal position
            if props.diamonds == 5 :
                # Move to base
                # base <- board_bot.properties.base # base <-
rumah tempat bot nge-store hasil diamond ambilannya
                goal_position <- base # ketika diamond sudah full,
dia akan mengubah value goal_position menjadi menuju base
                elif props.diamonds < 5 :
                    if props.diamonds == 0 :
                        goal_position <-
diamond_list[min_distance_index].position # else masih
ngumpulin diamond

                else :
                    if
distance(diamond_list[min_distance_index].position, base) >=
1.75 * distance(current_position, base) :
                        goal_position <- base
                    else :
                        goal_position <-
diamond_list[min_distance_index].position
                    else :
                        goal_position <- base

                # if goal_position: # Ketika punya goal position, bot
akan menerima tujuan pergerakan dari get_direction
                #      # We are aiming for a specific position,
calculate delta

                delta_x, delta_y <- get_direction(
                    current_position.x,
                    current_position.y,
                    goal_position.x,
                    goal_position.y,
)
            return delta_x, delta_y

```

4.2.5. nearestBaseDef.py

```
import math
from typing import Optional

from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position
from ..util import *

function distance(object1position, object2position):
    distance <- math.sqrt(math.pow(abs(object1position.x - object2position.x), 2) + math.pow(abs(object1position.y - object2position.y), 2))
    return distance

class NearestBaseDefLogic(BaseLogic):
    goal_position <- None # position defaultnya adalah None
    jika bot belum mengetahui position bot
        # jika sudah ada update pergerakan,
    value atributnya terunion dengan current position bot

    function next_move(board_bot: GameObject, board: Board) -> Tuple[int, int]:
        props <- board_bot.properties
        base <- board_bot.properties.base
        # Analyze new state

        diamond_base_distances <- []
        diamond_list <- board.diamonds
        for diamond in diamond_list:

            diamond_base_distances.append(distance(diamond.position, base))

            min_distance <- diamond_base_distances[0]
            min_distance_index <- 0
            for i, distance_info in enumerate(diamond_base_distances):
                if distance_info <= min_distance:
                    min_distance <- distance_info
                    min_distance_index <- i

            current_position <- board_bot.position

            # Proses menentukan goal position
            if props.diamonds == 5:
                # Move to base
                goal_position <- base # ketika diamond sudah full,
            ubah value goal_position menjadi menuju base
            elif props.diamonds < 5:
                if props.diamonds == 0:
                    goal_position <-
                diamond_list[min_distance_index].position # else masih
            ngumpulin diamond
            else:
                if
            distance(diamond_list[min_distance_index].position, base) >=
            1.75 * distance(current_position, base):
```

```

        # Bergerak ke base jika jarak diamond ke
base >= 1.75 * jarak bot ke base
            goal_position <- base
        else:
            # Bergerak ke diamond terdekat jika jarak
diamond ke base < 1.75 * jarak bot ke base
            goal_position <-
diamond_list[min_distance_index].position
        else:
            goal_position <- base

        current_position <- board_bot.position
        enemies_coordinate <- [bot.position for bot in
board.bots if bot.properties.name !=

board_bot.properties.name]
        dangerous_coordinate <- []

        for i in range(-2, 3):
            for j in range(-2, 3):
                if abs(i)+abs(j) <= 2 and (i != 0 or j != 0)
and (0 <= current_position.x+i < board.width) and (0 <=
current_position.y+j < board.height):

dangerous_coordinate.append(Position(current_position.y+j,
current_position.x+i))
            else:
                continue

        threat <- list(filter(lambda x: x in
enemies_coordinate, dangerous_coordinate))

        if len(threat) != 0:
            print("Threat detected")
            move_direction <- [1, 1, 1, 1] # right, left, up,
down
            for i in range(len(threat)):
                if threat[i].x > current_position.x or
current_position.x == board.width-1:
                    move_direction[0] <- 0
                elif threat[i].x < current_position.x or
current_position.x == 0:
                    move_direction[1] <- 0
                if threat[i].y > current_position.y or
current_position.y == board.height-1:
                    move_direction[2] <- 0
                elif threat[i].y < current_position.y or
current_position.y == 0:
                    move_direction[3] <- 0

            if 1 in move_direction:
                print("Moving to safety")
                goal_direction <- [0, 0, 0, 0] # right, left,
up, down
                if goal_position.x > current_position.x or
current_position.x < board.width-1:
                    goal_direction[0] <- 1
                elif goal_position.x < current_position.x or
current_position.x > 0:
                    goal_direction[1] <- 1
                if goal_position.y > current_position.y or

```

```

current_position.y < board.height-1:
    goal_direction[2] <- 1
elif goal_position.y < current_position.y or
current_position.y > 0:
    goal_direction[3] <- 1

    result <- [i for i, (x, y) in
enumerate(zip(move_direction, goal_direction)) if x == y == 1]

    if len(result) != 0:
        print("Intersection detected")
        move <- result[0]
    else:
        # No intersection between goal_position
and move_direction
        print("No intersection detected")
        move <- move_direction.index(1)

    if move == 0:
        delta_x, delta_y <- 1, 0
    elif move == 1:
        delta_x, delta_y <- -1, 0
    elif move == 2:
        delta_x, delta_y <- 0, 1
    elif move == 3:
        delta_x, delta_y <- 0, -1

    else:
        # if the bot is surrounded by enemies ;
move_direction = [0, 0, 0, 0]
        delta_x, delta_y <- get_direction(
            current_position.x,
            current_position.y,
            goal_position.x,
            goal_position.y,
        )
else:
    # No threat detected; move to targeted diamond
    print("No threat detected")
    delta_x, delta_y <- get_direction(
        current_position.x,
        current_position.y,
        goal_position.x,
        goal_position.y,
    )

return delta_x, delta_y

```

4.2.6. mostProfitable.py

```

from typing import Tuple, Optional, List
from math import dist, inf
from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position
from ..util import *

# enemy nonactive
class ProfitLogic(BaseLogic):

```

```

directions <- [(1, 0), (0, 1), (-1, 0), (0, -1)]
goal_position <- None
current_direction <- 0

function next_move(board_bot: GameObject, board: Board) -> Tuple[int, int]:
    props <- board_bot.properties

    if props.diamonds >= 4 and board_bot.position != board_bot.properties.base:
        # Move to base if inventory full
        base <- board_bot.properties.base
        goal_position <- base
    else:
        # Detect a list of diamond in board
        diamonds <- board.diamonds

        # Detect other bots in board
        enemies <- [bot for bot in board.bots if bot.properties.name != board_bot.properties.name]

        # Aim for the nearest diamond
        target <- getMostProfitable(board_bot, diamonds, enemies)
        goal_position <- target.position

    current_position <- board_bot.position
    if goal_position:
        # We are aiming for a specific position, calculate delta
        delta_x, delta_y <- get_direction(
            current_position.x,
            current_position.y,
            goal_position.x,
            goal_position.y,
        )
    else:
        # Roam around
        delta <- directions[current_direction]
        delta_x <- delta[0]
        delta_y <- delta[1]

    current_direction <- (current_direction + 1) % len(directions)

    return delta_x, delta_y

function calculateDistance(position1: Position, position2: Position) -> float:
    # Calculate distance between 2 Position
    return dist([position2.x, position2.y], [position1.x, position1.y])

function getMostProfitable(bot: GameObject, diamonds: List[GameObject], bots: List[GameObject]) -> GameObject:
    # Find the nearest diamond with most profit
    maxProfit <- 0
    for diamond in diamonds:
        nearestEnemy <- min(bots, key=lambda x:

```

```

calculateDistance(x.position, diamond.position)) # nearest
enemy from diamond
    distToBot <- calculateDistance(bot.position,
diamond.position)
    distToBase <-
calculateDistance(bot.properties.base, diamond.position)
    distToEnemy <-
calculateDistance(nearestEnemy.position, diamond.position)
    profit <- (distToEnemy +
diamond.properties.points) / (distToBot + distToBase)

    if profit >= maxProfit:
        selectedDiamond <- diamond
        maxProfit <- profit
    return selectedDiamond

```

4.2.7. defenseProfit1.py

```

from typing import Tuple, Optional, List
from math import dist
from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position
from ..util import *

class DefensiveProfitLogic1(BaseLogic):
    directions <- [(1, 0), (0, 1), (-1, 0), (0, -1)]
    goal_position <- None
    current_direction <- 0

    function next_move(board_bot: GameObject, board: Board) ->
Tuple[int, int]:
    props <- board_bot.properties

    if props.diamonds >= 5:
        # Move to base if inventory full
        base <- board_bot.properties.base
        goal_position <- base
    else:
        # Detect a list of diamond in board
        diamonds <- board.diamonds

        # Aim for the most profitable diamond
        target <- getMostProfitable(board_bot, diamonds)
        goal_position <- target.position

        current_position <- board_bot.position
        enemies_coordinate <- [bot.position for bot in
board.bots if bot.properties.name !=
board_bot.properties.name]
        dangerous_coordinate <- []

        for i <- range(-2, 3):
            for j <- range(-2, 3):
                if abs(i) + abs(j) <= 2 and (i != 0 or j != 0)
and (0 <= current_position.x + i < board.width) and (0 <=
current_position.y + j < board.height):

            dangerous_coordinate.append(Position(current_position.y + j,

```

```

        current_position.x + i))
            else:
                continue

            threat <- list(filter(lambda x: x in
enemies_coordinate, dangerous_coordinate))

            if len(threat) != 0:
                move_direction <- [1, 1, 1, 1] # right, left, up,
down
                for i <- range(len(threat)):
                    if threat[i].x > current_position.x or
current_position.x == board.width - 1:
                        move_direction[0] <- 0
                    elif threat[i].x < current_position.x or
current_position.x == 0:
                        move_direction[1] <- 0
                    if threat[i].y > current_position.y or
current_position.y == board.height - 1:
                        move_direction[2] <- 0
                    elif threat[i].y < current_position.y or
current_position.y == 0:
                        move_direction[3] <- 0

                    if 1 in move_direction:
                        goal_direction <- [0, 0, 0, 0] # right, left,
up, down
                        if goal_position.x > current_position.x or
current_position.x < board.width - 1:
                            goal_direction[0] <- 1
                        elif goal_position.x < current_position.x or
current_position.x > 0:
                            goal_direction[1] <- 1
                        if goal_position.y > current_position.y or
current_position.y < board.height - 1:
                            goal_direction[2] <- 1
                        elif goal_position.y < current_position.y or
current_position.y > 0:
                            goal_direction[3] <- 1

                        result <- [i for i, (x, y) in
enumerate(zip(move_direction, goal_direction)) if x == y == 1]

                        if len(result) != 0:
                            move <- result[0]
                        else:
                            move <- move_direction.index(1)

                        if move == 0:
                            delta_x, delta_y <- 1, 0
                        elif move == 1:
                            delta_x, delta_y <- -1, 0
                        elif move == 2:
                            delta_x, delta_y <- 0, 1
                        elif move == 3:
                            delta_x, delta_y <- 0, -1

                    else:
                        delta_x, delta_y <- get_direction(
current_position.x,

```

```

        current_position.y,
        goal_position.x,
        goal_position.y,
    )
else:
    delta_x, delta_y <- get_direction(
        current_position.x,
        current_position.y,
        goal_position.x,
        goal_position.y,
    )

return delta_x, delta_y

function calculateDistance(position1: Position, position2: Position) -> float:
    # Calculate distance between 2 Position
    return dist([position2.x, position2.y], [position1.x, position1.y])

function getMostProfitable(bot: GameObject, diamonds: List[GameObject]) -> GameObject:
    # Find the nearest diamond with most profit
    maxProfit <- 0
    for diamond in diamonds:
        distToBot <- calculateDistance(bot.position,
        diamond.position)
        distToBase <-
        calculateDistance(bot.properties.base, diamond.position)
        profit <- (diamond.properties.points) / (distToBot
        + distToBase)

        if profit >= maxProfit:
            selectedDiamond <- diamond
            maxProfit <- profit
    return selectedDiamond

```

4.2.8. defenseProfit2.py

```

from typing import Tuple, Optional, List
from math import dist
from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position
from ..util import clamp, get_direction

# Defensive clamp + most profitable(no enemy)
class DefensiveProfitLogic2(BaseLogic):
    goal_position <- None

    # Calculate next move based on the nearest diamond from
    bot
    function next_move(board_bot: GameObject, board: Board) ->
    Tuple[int, int]:
        props <- board_bot.properties
        current_position <- board_bot.position

        if (props.diamonds == 5 or (props.diamonds == 4 and

```

```

self.diamondtargetpoint == 2)) and current_position != board_bot.properties.base:
    # Move to base if inventory is full or inventory is 4 and the target diamond is 2 points and not already in base
    base <- board_bot.properties.base
    goal_position <- base
else:
    # Detect a list of diamond in board
    diamonds <- board.diamonds

    # Aim for the most profitable diamond
    target <- getMostProfitable(board_bot, diamonds)
    goal_position <- target.position

    enemies_coordinate <- [bot.position for bot in board.bots if bot.properties.name != board_bot.properties.name]

    # Plotting dangerous coordinate
    dangerous_coordinate <- []

    for i in range(-1, 2):
        for j in range(-1, 2):
            if (i != 0 or j != 0) and (0 <= current_position.x+i < board.width) and (0 <= current_position.y+j < board.height):

                dangerous_coordinate.append(Position(current_position.y+j, current_position.x+i))
            else:
                continue

    # Find the intersection between dangerous coordinate and enemies coordinate
    threat <- list(filter(lambda x: x in enemies_coordinate, dangerous_coordinate))

    if len(threat) != 0:
        directions <- [(1,0), (-1,0), (0,1), (0,-1)]
        enemy_direction <- []

        for coordinate in threat:
            x <- clamp(coordinate.x-current_position.x, -1, 1)
            if x != 0 and (x, 0) not in enemy_direction:
                enemy_direction.append((x, 0))
            y <- clamp(coordinate.y-current_position.y, -1, 1)
            if y != 0 and (0, y) not in enemy_direction:
                enemy_direction.append((0, y))

        # List all potential directions to move
        directions <- list(filter(lambda x: x not in enemy_direction, directions))

    # Handling edge
    if current_position.x == 0:
        directions.remove((-1,0))
    elif current_position.x == board.width-1:

```

```

        directions.remove((1, 0))
        if current_position.y == 0:
            directions.remove((0, -1))
        elif current_position.y == board.height-1:
            directions.remove((0, 1))

        if len(directions) == 0:
            # No safe direction to move, just move to the
            nearest diamond
            delta_x, delta_y <- get_direction(
                current_position.x,
                current_position.y,
                goal_position.x,
                goal_position.y,
            )

        else:
            # Move to the nearest safe direction
            x <- clamp(goal_position.x-current_position.x,
            -1, 1)
            y <- clamp(goal_position.y-current_position.y,
            -1, 1)

            if (x,0) in directions:
                delta_x <- x
                delta_y <- 0
            elif (0,y) in directions:
                delta_x <- 0
                delta_y <- y
            else:
                delta_x, delta_y <- directions[0][0],
                directions[0][1]

            else:
                # Move safely to the nearest diamond
                delta_x, delta_y <- get_direction(
                    current_position.x,
                    current_position.y,
                    goal_position.x,
                    goal_position.y,
                )

        return delta_x, delta_y

    function calculateDistance(position1: Position, position2:
Position) -> float:
    # Calculate distance between 2 Position
    return dist([position2.x, position2.y], [position1.x,
position1.y])

    function getMostProfitable(bot: GameObject, diamonds:
List[GameObject]) -> GameObject:
        # Find the nearest diamond with the most profit
        maxProfit <- 0
        for diamond in diamonds:
            distToBot <- calculateDistance(bot.position,
diamond.position)
            distToBase <-
calculateDistance(bot.properties.base, diamond.position)
            profit <- (diamond.properties.points) / (distToBot

```

```

+ distToBase)

        if profit >= maxProfit:
            selectedDiamond <- diamond
            maxProfit <- profit
            self.diamondtargetpoint <-
diamond.properties.points
        return selectedDiamond

```

4.2.9. defenseProfit3.py

```

class DefensiveProfitLogic3:
    initialize():
        goal_position = None
        aim_for_teleporter = False
        diamondtargetpoint = None

    next_move(board_bot, board):
        props = board_bot.properties
        current_position = board_bot.position
        direction = [(1, 0), (0, 1), (-1, 0), (0, -1)]

        # handling edge
        if current_position.x == 0:
            remove((-1, 0)) from direction
        elif current_position.x == board.width-1:
            remove((1, 0)) from direction
        if current_position.y == 0:
            remove((0, -1)) from direction
        elif current_position.y == board.height-1:
            remove((0, 1)) from direction

        diamonds = []
        teleporters = {}

        # Detect diamonds and teleporters in the board
        for object in board.game_objects:
            if object.type == "DiamondGameObject":
                add object to diamonds
            elif object.type == "TeleportGameObject":
                if object.properties.pair_id not in
teleporters:
                    teleporters[object.properties.pair_id] =
[]
                    add object to
teleporters[object.properties.pair_id]

                nearestTeleporter = find_nearest_teleporter(board_bot,
teleporters)

            # Check conditions for moving to base
            if (props.diamonds == 5 or (props.diamonds == 4 and
diamondtargetpoint == 2)) and current_position !=
board_bot.properties.base:
                goal_position = board_bot.properties.base
            else:
                nearestTeleporterPair =

```

```

find_nearest_teleporter_pair(nearestTeleporter)
    target = find_most_profitable_diamond(board_bot,
diamonds, nearestTeleporter, nearestTeleporterPair)
    goal_position = target.position if not
aim_for_teleporter else nearestTeleporter.position

    enemies_coordinate = get_enemies_coordinates(board,
board_bot)

    update_direction_for_teleporter(enemies_coordinate,
current_position, nearestTeleporter, direction)

    if len(direction) == 0:
        delta_x, delta_y = get_direction(current_position,
goal_position)
    else:
        x = clamp(goal_position.x - current_position.x,
-1, 1)
        y = clamp(goal_position.y - current_position.y,
-1, 1)
        if (x, 0) in direction:
            delta_x = x
            delta_y = 0
        elif (0, y) in direction:
            delta_x = 0
            delta_y = y
        else:
            delta_x, delta_y = direction[0]

    return delta_x, delta_y

find_nearest_teleporter(board_bot, teleporters):
    # Find the nearest teleporter to the bot
    nearestTeleporter = None
    minDist = infinity
    for key in teleporters:
        for teleporter in teleporters[key]:
            distTeleporterToBot =
calculate_distance(board_bot.position, teleporter.position)
            if distTeleporterToBot < minDist:
                nearestTeleporter = teleporter
                minDist = distTeleporterToBot
    return nearestTeleporter

find_nearest_teleporter_pair(nearestTeleporter):
    # Find the pair of the nearest teleporter
    nearestTeleporterIdx = index_of(nearestTeleporter) in
teleporters[nearestTeleporter.properties.pair_id]
    return
teleporters[nearestTeleporter.properties.pair_id][1 -
nearestTeleporterIdx]

find_most_profitable_diamond(board_bot, diamonds,
nearestTeleporter, nearestTeleporterPair):
    # Find the most profitable diamond
    maxProfit = 0
    selectedDiamond = None

    for diamond in diamonds:
        realdistToBot =

```

```

calculate_distance(board_bot.position, diamond.position)
    altDistToBot =
calculate_distance(nearestTeleporter.position,
board_bot.position) +
calculate_distance(nearestTeleporterPair.position,
diamond.position)
    if realdistToBot > 1.6 * altDistToBot:
        aim_for_teleporter = True
        distToBot = altDistToBot
    else:
        aim_for_teleporter = False
        distToBot = realdistToBot
    distToBase =
calculate_distance(board_bot.properties.base,
diamond.position)
    profit = diamond.properties.points / (distToBot +
distToBase)

    if profit >= maxProfit:
        selectedDiamond = diamond
        maxProfit = profit
        diamondtargetpoint = diamond.properties.points

return selectedDiamond

get_enemies_coordinates(board, board_bot):
    # Get coordinates of other bots in the board
    return [bot.position for bot in board.bots if
bot.properties.name != board_bot.properties.name]

update_direction_for_teleporter(enemies_coordinate,
current_position, nearestTeleporter, direction):
    # Update direction to avoid teleporter if enemies are
nearby
    if not aim_for_teleporter:
        for coord in enemies_coordinate:
            if distance([coord.x, coord.y],
[current_position.x, current_position.y]) <= 1.5: # sqrt(2) =
1.414
                x = clamp(coord.x - current_position.x,
-1, 1)
                if x != 0 and (x, 0) in direction:
                    remove (x, 0) from direction
                y = clamp(coord.y - current_position.y,
-1, 1)
                if y != 0 and (0, y) in direction:
                    remove (0, y) from direction

```

4.3. Analisis Desain Solusi Algoritma Greedy

Aspek yang Diuji	Implementasi Strategi	Hasil
Penentuan diamond dengan <i>profit</i> terbesar	1. Dilakukan pengecekan terhadap seluruh diamond yang ada dalam board	Strategi berjalan baik pada banyak kasus secara umum dapat menentukan <i>diamond</i> yang memiliki

	<p>(seluruh diamond terdapat pada list diamonds).</p> <ol style="list-style-type: none"> 2. Pada setiap diamond dilakukan pengecekan terhadap informasi poin dari diamond, jarak ke base, dan jarak ke bot. 3. Pada pengecekan informasi jarak diamond ke bot, akan dihitung terlebih dahulu jarak antara diamond dengan bot berdasarkan jarak sesungguhnya pada board (jarak yang terpisahkan oleh blok-blok board). 4. Kemudian, akan dilakukan perhitungan jarak antara bot dengan teleporter terdekat ditambah dengan jarak antara pasangannya dengan diamond yang sedang ditelaah untuk kemudian disimpan dalam sebuah variabel yang merepresentasikan alternatif besarnya jarak antara diamond dengan bot. 5. Apabila jarak bot dengan diamond berdasarkan blok lebih besar 1.6 kali daripada jarak bot ke teleporter yang ditambahkan dengan jarak teleporter ke diamond, maka jarak yang akan diambil sebagai nilai jarak bot untuk perhitungan profit suatu diamond 	<p><i>profit</i> untuk bot terbesar, tetapi terdapat hal yang dapat ditingkatkan, yakni:</p> <ul style="list-style-type: none"> - Pada beberapa kasus, terdapat opsi yang lebih optimal untuk diambil. Namun mengingat implementasi perhitungan <i>teleporter</i> yang ditambahkan di akhir, dibutuhkan evaluasi lebih lanjut untuk rumus yang digunakan dalam kalkulasi <i>diamond's profit</i>
--	--	---

	<p>adalah jarak bot ke diamond dengan teleporter.</p> <p>6. Kemudian, perhitungan profit akan dilakukan dengan rumus profit yaitu besar poin diamond dibagi dengan jumlah dari jarak diamond ke bot dengan jarak diamond ke base.</p> <p>7. Jika nilai profit yang didapat dari hasil perhitungan rumus profit nilainya lebih besar dari maxProfit, maka diamond yang dijadikan target adalah diamond yang sedang dijadikan target.</p>	
Menentukan <i>goal</i> yang diambil oleh bot	<p>1. Dilakukan pengecekan terhadap jumlah <i>diamond</i> yang sedang dibawa oleh bot</p> <p>2. Jika jumlah <i>diamond</i> sudah maksimal atau jumlah <i>diamond</i> max-1 dan bot ingin mengambil <i>diamond</i> dengan value 2 atau bot berada dalam radius 24 kotak di sekitar base, maka <i>base</i> akan dijadikan <i>goal</i> bot</p> <p>3. Jika kondisi sebelumnya tidak terpenuhi, akan dilakukan kalkulasi <i>most profitable diamond</i></p> <p>4. Jika saat kalkulasi didapatkan bahwa bot akan menggunakan</p>	Strategi berjalan baik pada banyak kasus secara umum dan dapat menentukan posisi <i>goal</i> yang tepat

	<i>teleporter</i> , maka <i>teleporter</i> akan dijadikan <i>goal</i> dan jika tidak maka <i>goal</i> dari bot adalah <i>most profitable diamond</i>	
Pemilihan arah menuju <i>goal</i> dengan atau tanpa dengan menggunakan <i>teleporter</i>	<ol style="list-style-type: none"> 1. Pada kalkulasi <i>most profitable diamond</i>, dilakukan perbandingan antara jarak bot langsung menuju <i>diamond</i> dan melewati <i>teleporter</i> 2. Jika dengan melalui <i>teleporter</i> dihasilkan jarak yang lebih pendek, maka variabel <i>self.aim_for_teleporter</i> akan dijadikan bernilai True dan <i>goal</i> bot adalah <i>teleporter</i> 3. Jika tidak, maka <i>goal</i> bot akan dijadikan <i>most profitable diamond</i> 	<p>Strategi berjalan baik pada banyak kasus secara umum dan dapat menentukan apakah melewati <i>teleporter</i> atau tidak, tetapi masih terdapat <i>handling</i> beberapa kasus yang bisa ditingkatkan, yakni:</p> <ul style="list-style-type: none"> - Terdapat beberapa kasus di mana bot memilih untuk berputar menggunakan <i>teleporter</i> dibandingkan langsung menuju <i>goal</i>
Pengambilan langkah defense	<ol style="list-style-type: none"> 1. Dilakukan pengecekan terhadap jarak dari setiap bot musuh ke posisi bot saat ini 2. Untuk setiap bot yang terdeteksi terdapat dalam <i>range</i> yang ditentukan, maka opsi pergerakan ke arah bot tersebut akan dihindarkan 3. Jika hasil akhir dari arah yang mungkin bot ambil beririsan dengan arah <i>goal</i>, maka arah yang beririsan akan menjadi arah akhir yang diambil 4. Jika tidak ada, maka bot akan mengambil arah pertama yang 	<p>Strategi berjalan baik pada banyak kasus secara umum, tetapi masih terdapat beberapa permasalahan yang belum di-<i>handle</i>, yakni:</p> <ul style="list-style-type: none"> - Akibat proses konsiderasi yang lama, bot terkadang tidak mendapatkan posisi ril dari bot lawan sehingga hasil akhir gerakan dapat terlihat seakan-akan bot mendekati lawan. Hal ini dapat dilihat dengan melakukan printing info koordinat yang digunakan bot - Bot membutuhkan waktu yang lebih lama untuk <i>processing</i>

	<p>terdapat dari opsi pergerakan yang tersisa</p> <p>5. Jika bot terkepung, maka bot tidak akan mempertimbangkan apapun kecuali menuju target</p>	<p><i>time</i> sehingga dapat di-tackle terlebih dahulu oleh bot lawan</p> <p>- Bot tidak melakukan kalkulasi lebih lanjut untuk menghindari pinggir dari board sehingga terdapat kemungkinan untuk bot memojokkan dirinya sendiri</p>
Pengambilan langkah attack	<ol style="list-style-type: none"> 1. Dilakukan pengecekan terhadap jarak dari setiap bot musuh ke posisi bot saat ini 2. Untuk setiap bot yang terdeteksi membawa jumlah <i>diamond</i> yang lebih sedikit akan dilakukan pengecekan lebih lanjut 3. Jika bot lawan tersebut tepat berada di samping atas, kanan, kiri, atau bawah bot, serta sejalan dengan arah <i>goal</i> maka bot akan melakukan penyerangan 	<p>Strategi berjalan baik pada banyak kasus secara umum dan bot tidak mengejar-ngejar bot lawan dengan persisten, tetapi masih terdapat beberapa permasalahan yang belum di-handle, yakni:</p> <ul style="list-style-type: none"> - Bot membutuhkan waktu yang lebih lama untuk <i>processing time</i> sehingga dapat memperlambat proses penyerangan
Penghindaran dari teleporter yang membentuk looping track menuju base	<ol style="list-style-type: none"> 1. Dilakukan pengecekan apakah arah menggunakan <i>teleporter</i> atau tidak 2. Jika tidak menggunakan <i>teleporter</i>, maka akan dicek apakah <i>teleporter</i> tepat berada di samping atas, kanan, kiri, atau bawah bot 3. Jika terdapat <i>teleporter</i> maka arah gerakan menuju <i>teleporter</i> tersebut akan dihilangkan dari 	<p>Strategi berjalan baik pada banyak kasus secara umum, tetapi masih terdapat beberapa permasalahan yang belum selesai diperbaiki, yakni:</p> <ul style="list-style-type: none"> - Terdapat inkonsistensi di mana terkadang bot dapat ‘going around’ <i>teleporter</i> dan terkadang bot ‘terhalang’ <i>teleporter</i> - Bot yang terhalang <i>teleporter</i> biasanya akan bergerak di arah kanan-kiri-kanan-kiri hingga <i>teleporter</i>

	opsi	berpindah tempat. Hal ini meningkatkan vulnerabilitas bot terhadap serangan lawan
--	------	---

BAB V

KESIMPULAN DAN SARAN

Dari tugas besar IF2211 Strategi Algoritma ini, kami telah berhasil membuat bot permainan “Diamonds” menggunakan strategi Greedy. Strategi yang digunakan tidak hanya satu, melainkan kombinasi dari beberapa strategi yang memiliki prioritasnya masing-masing sehingga terciptalah sebuah bot dengan strategi terbaik versi kami.

Algoritma greedy terbukti dapat digunakan untuk membuat bot yang cukup baik pada permainan Diamonds karena solusi yang diambil pada setiap langkahnya merupakan optimum lokal. Bot akan mengutamakan pengambilan diamond dengan mempertimbangkan profit dari tiap diamond yang terlebih dahulu dikalkulasi dengan mempertimbangkan besar poin yang dimiliki diamond, jarak antara diamond dengan base, dan jarak antara diamond dengan bot.

Saran pengembangan untuk tugas besar ini adalah:

1. Pengeksplorasiyan yang lebih maksimal untuk mendapatkan solusi greedy yang dapat mengantarkan pengguna menuju solusi yang mencapai optimum global.

LAMPIRAN

Link repository dari Tugas Besar 01 IF Kelompok “Snoopy” adalah sebagai berikut :

https://github.com/kaylanamira/Tubes1_Snoopy

Video Tugas Besar 01 IF Kelompok “Snoopy” adalah sebagai berikut :

https://youtu.be/-cy5HEH_v-o

DAFTAR PUSTAKA

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-B
ag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf) diakses pada 2 Maret 2024 pukul 13.38 WIB.