LAPORAN TUGAS BESAR 1

IF2211 STRATEGI ALGORITMA

Dipersiapkan oleh:

Kelompok 33 - K03

Maulvi Ziadinda M (13522122)

Satriadhikara Panji Yudhistira (13522125)

Valentino Chryslie Triadi (13522164)

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
DESKRIPSI TUGAS

Diamonds adalah tantangan pemrograman yang menantang para peserta untuk bertanding menggunakan bot yang mereka kembangkan melawan bot peserta lain. Setiap peserta akan memiliki bot dengan tujuan utama mengumpulkan sebanyak mungkin diamond. Namun, tantangan ini tidak akan mudah karena berbagai hambatan akan menambah keseruan dan kerumitan dalam permainan. Untuk memenangkan kompetisi, peserta harus menerapkan strategi khusus pada bot mereka masing-masing. Penjelasan lebih lanjut mengenai aturan permainan akan dijelaskan di bawah.

Gambar 1.1 Layar permainan Diamonds

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

Komponen-komponen dari permainan Diamonds antara lain:

1. Diamonds

    Untuk meraih kemenangan dalam pertandingan, pemain harus mengumpulkan sejumlah diamond dengan cara melewati atau mengambilnya. Ada dua varian diamond dalam permainan: diamond merah dan biru. Setiap diamond merah memiliki nilai dua poin, sementara diamond biru bernilai satu poin. Diamond akan terus muncul kembali secara periodik, dan proporsi antara diamond merah dan biru akan berubah-ubah setiap kali terjadi regenerasi.

Gambar 1.2 Varian Diamond Biru

Gambar 1.3 Varian Diamond Merah

2. Red Button / Diamond Button

   Saat tombol merah dilewati atau dilangkahi, semua diamond, diamond merah maupun diamond biru, akan muncul kembali di papan permainan dengan posisi yang acak. Lokasi tombol merah itu sendiri juga akan berubah ke posisi acak setelah tombol tersebut diaktifkan.

Gambar 1.4 Red Button / Diamond Button

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.

Gambar 1.5 Teleporters

4. Bots

   Pada game ini kita akan menggerakkan bot untuk mendapatkan *diamond* sebanyak banyaknya. Bot ini bisa men-*tackle* bot lainnya.

Gambar 1.6 Bot dengan name “stima”

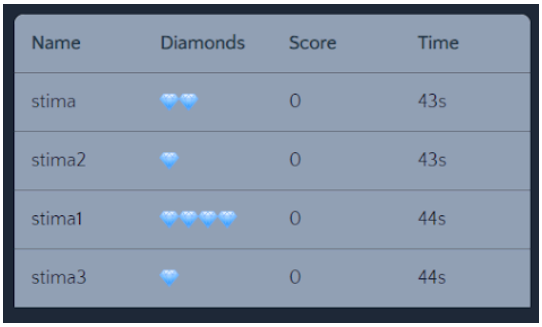
5. Bases

   Setiap bot dilengkapi dengan sebuah Base, yang berfungsi sebagai tempat untuk menampung diamond yang dikumpulkan. Ketika diamond disimpan di Base, skor dari bot tersebut akan meningkat sesuai dengan nilai diamond yang disimpan, dan inventaris bot (yang akan diuraikan lebih lanjut) akan dikosongkan.

Gambar 1.7 Base dengan name “stima”

6. Inventory

   Bot dilengkapi dengan sebuah inventaris, yang berperan sebagai lokasi penyimpanan sementara untuk diamond yang telah dikumpulkan. Inventaris ini dibatasi oleh sebuah kapasitas maksimal, yang artinya dapat terisi penuh. Untuk menghindari keadaan penuh tersebut, bot dapat mentransfer isi inventaris mereka ke Base, sehingga memungkinkan inventaris tersebut untuk dikosongkan kembali.



Gambar 1.8 Layar Inventory

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.

Dalam tugas besar ini, mahasiswa diminta untuk bekerja dalam kelompok minimal dua orang dan maksimal tiga orang, dengan lintas kelas dan lintas kampus diperbolehkan. Tujuan dari tugas ini adalah untuk mengimplementasikan algoritma Greedy pada bot permainan Diamonds dengan strategi yang dirancang untuk memenangkan permainan. Setiap kelompok diminta untuk mengembangkan strategi Greedy terbaik yang berkaitan dengan fungsi objektif permainan, yaitu mengumpulkan diamond sebanyak mungkin dan mencegah diamond tersebut diambil oleh bot lain.

Strategi Greedy yang diterapkan oleh setiap kelompok harus dijelaskan secara eksplisit dalam laporan, dan kode program yang sesuai dengan strategi tersebut harus disertakan. Mahasiswa dilarang menggunakan kode program yang diunduh dari internet; setiap kelompok diharapkan untuk membuat program mereka sendiri dengan menggunakan

kreativitas mereka. Program harus dapat dijalankan pada game engine yang ditentukan dan dapat bersaing dengan bot dari kelompok lain.

Setiap implementasi strategi Greedy harus dijelaskan dengan komentar yang jelas dalam kode program. Selain itu, terdapat bonus poin untuk kelompok yang membuat video tentang aplikasi Greedy pada bot dan simulasinya dalam permainan, dengan menampilkan wajah dari setiap anggota kelompok. Asistensi tugas besar bersifat opsional, namun jika diperlukan, mahasiswa dapat meminta bimbingan dari asisten yang dipilih.

Kelompok yang telah membuat bot akan berkompetisi dengan kelompok lain dalam kompetisi yang disaksikan oleh seluruh peserta kuliah. Hadiah menarik akan diberikan kepada kelompok yang memenangkan kompetisi. Informasi terkait pendataan kelompok, asistensi, demo, dan pengumpulan laporan telah disediakan melalui tautan yang telah disediakan.

Seluruh laporan tugas besar harus dikumpulkan dalam format PDF melalui tautan yang telah disediakan paling lambat pada tanggal 9 Maret 2024. Isi dari repository yang digunakan untuk menyimpan program harus mengikuti struktur yang telah ditentukan, termasuk folder src untuk source code, folder doc untuk laporan, dan README yang berisi penjelasan singkat algoritma Greedy yang diimplementasikan, persyaratan program, langkah-langkah untuk penggunaan, dan informasi tentang pembuat program.

BAB II
LANDASAN TEORI

A. Algoritma Greedy

Algoritma greedy adalah algoritma yang memecahkan persoalan secara langkah per langkah (step by step) sedemikian sehingga, pada setiap langkah:

1. Mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan (prinsip“take what you can get now!”)
2. Berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global.

Dalam menjalankan algoritma greedy ada beberapa elemen yang diperlukan agar proses pemecahan masalah secara greedy lebih mudah untuk dilakukan. Elemen - elemen tersebut adalah sebagai berikut.

1. Himpunan kandidat (**C**), berisi kandidat yang akan dipilih pada setiap Langkah. Contohnya adalah simpul atau sisi di dalam graf, job, task, koin, benda, dan karakter.
2. Himpunan solusi (**S**), berisi kandidat yang sudah dipilih untuk menjadi solusi.
3. Fungsi solusi yang digunakan menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi atau belum.
4. Fungsi seleksi (selection function) untuk memilih kandidat berdasarkan strategi greedy tertentu. Strategi greedy ini bersifat heuristik yang tentunya berbeda untuk setiap masalah yang dihadapi.
5. Fungsi kelayakan (feasible) untuk memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi atau tidak.
6. Fungsi objektif untuk memaksimumkan atau meminimumkan kandidat yang dipilih.

Setelah semua elemen - elemen tersebut lengkap, maka proses pencarian solusi akan bisa dilakukan. Algoritma greedy melibatkan pencarian sebuah himpunan bagian (**S**) dari himpunan kandidat (**C**) yang dalam hal ini, himpunan **S** harus memenuhi beberapa kriteria

yang ditentukan, yaitu **S** menyatakan suatu solusi dan **S** di optimisasi oleh fungsi objektif. Algoritma greedy akan menghasilkan beberapa solusi optimum lokal dan dari semua solusi lokal tersebut akan dipilih solusi terbaik yang menjadi solusi optimum global.

Akan tetapi, solusi optimum global yang dihasilkan algoritma greedy belum tentu merupakan solusi terbaik karena sangat mungkin solusi tersebut merupakan solusi sub-optimum atau pseudo-optimum. Ada dua alasan kenapa hal tersebut bisa terjadi, yaitu sebagai berikut.

1. Algoritma greedy tidak beroperasi secara menyeluruh terhadap semua kemungkinan solusi yang ada (sebagaimana pada metode exhaustive search).
2. Terdapat beberapa fungsi seleksi yang berbeda, sehingga perlu memilih fungsi yang tepat jika algoritma diinginkan untuk menghasilkan solusi optimal.

Sebagai kesimpulan, pada sebagian persoalan, algoritma greedy tidak selalu berhasil memberi solusi terbaik. Jika solusi terbaik mutlak tidak terlalu diperlukan, maka algoritma greedy dapat digunakan untuk menghasilkan solusi hampiran (*approximation*), daripada menggunakan algoritma yang kebutuhan waktunya eksponensial untuk menghasilkan solusi yang terbaik. Contohnya adalah permasalahan mencari lintasan dengan bobot minimal pada persoalan Traveling Salesman Problem pada grafik dengan jumlah simpul *n*. Apabila jumlah simpul sangat banyak, maka penyelesaian dengan algoritma brute force akan membutuhkan waktu komputasi yang lama untuk menemukannya. Namun dengan algoritma greedy, meskipun tur dengan berbobot minimal tidak dapat ditemukan, namun solusi dengan algoritma greedy dianggap sebagai hampiran solusi optimal. Secara matematis, pembuktian bahwa algoritma greedy tidak selalu menghasilkan solusi yang optimal biasanya dilakukan dengan menggunakan *counterexample* bahwa ada solusi yang lebih optimal.

Dalam penerapannya, ada cukup banyak permasalahan yang bisa diselesaikan dengan pendekatan greedy, yaitu sebagai berikut.

1. Persoalan penukaran uang (coin exchange problem)
2. Persoalan memilih aktivitas (activity selection problem)
3. Minimisasi waktu di dalam sistem

4. Persoalan knapsack (knapsack problem)
5. Penjadwalan job dengan tenggat waktu (job schedulling with deadlines)
6. Pohon merentang minimum (minimum spanning tree)
7. Lintasan terpendek (shortest path)
8. Kode Huffman (Huffman code)
9. Pecahan Mesir (Egyptian fraction)

B. Cara Kerja Program

1. Struktur Direktori Program

Program Bot yang dibuat dalam tugas besar 1 memiliki struktur direktori sebagai berikut.

```
Root Directory
├── game
│   ├── logic
│   │   ├── __init__.py
│   │   ├── base.py
│   │   ├── ...
│   │   └── random.py
│   ├── __init__.py
│   ├── board_handler.py
│   ├── bot_handler.py
│   ├── models.py
│   ├── util.py
│   └── api.py
├── decode.py
├── main.py
├── requirements.txt
├── run-bots.bat
└── run-bots.sh
```

Secara garis besar, proses pembuatan algoritma hanya akan dilakukan pada folder `/game/logic`. Semua file diluar folder logic adalah file yang berfungsi sebagai komponen-komponen untuk menjalankan bot dan untuk berkomunikasi dengan website game.

2. Proses Pelaksanaan Aksi oleh Bot

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

a. Program bot akan melakukan pengecekan apakah bot sudah terdaftar atau belum dengan mengirimkan POST request terhadap endpoint `/api/bots/recover` dengan body berisi email dan password bot. Jika bot sudah terdaftar, maka backend akan memberikan response code 200 dengan body berisi id dari bot tersebut. Jika tidak, backend akan memberikan response code 404.
b. Jika bot belum terdaftar, maka program bot akan mengirimkan POST request terhadap endpoint `/api/bots` dengan body berisi email, name, password, dan team. Jika berhasil, maka backend akan memberikan response code 200 dengan body berisi id dari bot tersebut.
c. Ketika id bot sudah diketahui, bot dapat bergabung ke board dengan mengirimkan POST request terhadap endpoint `/api/bots/{id}/join` dengan body berisi board id yang diinginkan (preferredBoardId). Apabila bot berhasil bergabung, maka backend akan memberikan response code 200 dengan body berisi informasi dari board.
d. Program bot akan mengkalkulasikan move selanjutnya secara berkala berdasarkan kondisi board yang diketahui, dan mengirimkan POST request terhadap endpoint `/api/bots/{id}/move` dengan body berisi direction yang akan ditempuh selanjutnya (“NORTH”, “SOUTH”, “EAST”, atau “WEST”). Apabila berhasil, maka backend akan memberikan response code 200 dengan body berisi kondisi board setelah move tersebut. Langkah ini dilakukan terus-menerus hingga waktu bot habis. Jika waktu bot habis, bot secara otomatis akan dikeluarkan dari board.
e. Program frontend secara periodik juga akan mengirimkan GET request terhadap endpoint `/api/boards/{id}` untuk mendapatkan kondisi board terbaru, sehingga tampilan board pada frontend akan selalu ter-update.

3. Implementasi Algoritma Greedy ke Dalam Bot

Untuk mengimplementasikan algoritma yang telah dibuat ke dalam bot, perlu dibuat terlebih dahulu sebuah folder baru pada direktori */game/logic*, misalnya *MyBot.py*. Lalu Selanjut akan kelas baru yang meng-inherit kelas *BaseLogic*, lalu implementasikan *constructor* dan method *next_move* pada kelas tersebut. Berikut adalah contoh dari pembuatan kelas.

```
from game.logic.base import BaseLogic
from game.models import Board, GameObject

class MyBot(BaseLogic):
    def _init_(self):
          # Initialize attributes necessary
          self.my_attribute = 0
    def next_moveself, board_bot: GameObject, board: Board):
          # Calculate next move
          delta_x = 1
          delta_y = 0

          return delta_x, delta_y
```

Fungsi next_move pada kutipan di atas akan mengembalikan nilai delta_x dan delta_y, di mana nilai yang diperbolehkan hanyalah (1, 0), (0, 1), (-1, 0), (0, -1). Apabila nilai ilegal atau di-luar range board, maka move akan diabaikan oleh program dan akan mucul error Invalid Move.

Langkah selanjutnya adalah melakukan Import kelas yang telah dibuat pada main.py dan mendaftarkannya pada dictionary CONTROLLERS. Berikut adalah contoh dari proses Import kelas.

```
from game.logic.mybot import MyBot

init()
BASE_URL = "http://localhost:3000/api"
DEFAULT_BOARD_ID = 1
CONTROLLERS = {"Random": RandomLogic, "MyBot": MyBot}
```

Bot yang telah dibuat dapat dijalankan satu per satu atau dapat dijalankan secara bersamaan dengan bot lain dengan menggunakan .bat atau .sh script. Proses menjalankan program akan dijelaskan lebih lanjut pada poin 4.

4. Menjalankan Program Bot

Program bot yang telah dibuat dapat dijalankan dengan menggunakan aplikasi *cmd* atau *terminal.* Untuk menjalankan satu bot dengan logic yang terdapat pada file *game/logic/random.py* diperlukan perintah seperti di bawah ini. Argumen logic pada *command/script* tersebut perlu disesuaikan menjadi nama bot yang telah terdaftar pada *CONTROLLERS.*

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

Untuk menjalankan beberapa bot sekaligus, misalnya menjalankan 4 bot dengan logic yang sama, yaitu *game/logic/random.py*. Perlu di buat terlebih dahulu file *run-bots.bat* pada Windows atau *run-bots.sh* pada Linux dan MacOS. File tersebut akan berisi perintah - perintah yaitu sebagai berikut.

```
@echo off
start cmd /c "python main.py --logic Random --email=your_email@example.com
--name=your_name --password=your_password --team etimo"
start cmd /c "python main.py --logic Random --email=your_email@example.com
--name=your_name --password=your_password --team etimo"
start cmd /c "python main.py --logic Random --email=your_email@example.com
--name=your_name --password=your_password --team etimo"
start cmd /c "python main.py --logic Random --email=your_email@example.com
--name=your_name --password=your_password --team etimo"
```

Script yang ada pada *run-bots.bat* atau *run-bots.sh* dapat disesuaikan sesuai dengan *logic* file yang digunakan, email, nama, ataupun password.

BAB III
APLIKASI STRATEGI GREEDY

A. Mapping Persoalan Diamonds

Berdasarkan studi pustaka yang telah dilakukan pada landasan teori, dikatakan algoritma greedy melibatkan pencarian sebuah himpunan bagian (**S**) dari himpunan kandidat (**C**) yang dalam hal ini, himpunan **S** harus memenuhi beberapa kriteria yang ditentukan, yaitu **S** menyatakan suatu solusi dan **S** di optimisasi oleh fungsi objektif. Oleh karena itu, untuk bisa menerapkan algoritma greedy dalam permasalahan ini, maka perlu diidentifikasi terlebih dahulu elemen - elemen yang ada di dalam *game* Diamonds menjadi elemen - elemen algoritma greedy, yaitu sebagai berikut.

1. Himpunan kandidat (**C**)

   Himpunan kandidat dalam permasalahan ini akan berisi koordinat - koordinat objek yang ada di dalam *game*. Objek - objek tersebut adalah *blue diamond*, *red diamond*, *red button, teleporter, base bot, bot* musuh, dan *base* musuh. Selain itu, ada juga elemen penting yang perlu diperhatikan dalam *game*, yaitu jarak *bot* ke koordinat tujuan.
2. Himpunan solusi (**S**)

   Himpunan solusi berisi koordinat yang akan memberikan keuntungan paling banyak ketika didatangi oleh *bot*. Berdasarkan pendekatan *greedy* yang dilakukan, himpunan solusi bisa hanya berisi satu koordinat atau beberapa koordinat yang harus dilalui secara berurutan.
3. Fungsi solusi

   Fungsi solusi berguna untuk menentukan apakah kandidat yang dipilih sudah memberikan solusi. Secara umum, fungsi ini hanya akan digunakan ketika permainan akan berakhir. Fungsi ini akan melakukan perhitungan apakah waktu yang tersisa cukup untuk menempuh suatu koordinat dalam *map* dan untuk kembali ke base. Alasan kenapa fungsi ini hanya dilakukan di akhir permainan adalah karena fungsi-fungsi untuk

menentukan kandidat solusi pada fungsi seleksi sudah menghilangkan kandidat yang tidak bisa menjadi solusi.

4. Fungsi Seleksi (*Selection)*

   Fungsi seleksi berguna untuk memilih kandidat berdasarkan strategi greedy tertentu. Fungsi seleksi akan dijelaskan lebih lanjut pada bagian alternatif solusi karena pendekatan greedy yang berbeda akan memiliki fungsi seleksi yang berbeda.

5. Fungsi Kelayakan (*Feasible)*

   Fungsi kelayakan berguna untuk memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi. Dalam *game* Diamonds, fungsi kelayakan akan digunakan untuk menentukan apakah objek dalam *game* bisa menjadi tujuan berikutnya atau tidak. Contoh yang paling sederhana adalah ketika *red diamond* menjadi tujuan selanjutnya, namun *bot* sudah memiliki 4 *diamonds* dalam *inventory* sehingga *red diamond* tidak bisa diambil. Fungsi kelayakan akan mencegah *bot* untuk menuju koordinat tersebut, karena apabila bot tetap menuju koordinat tersebut ada kemungkinan terjadi *bug* yaitu *bot* akan berputar - putar di sekitar *red diamond* atau bahkan memberikan *invalid move.*

6. Fungsi objektif

   Fungsi objektif digunakan untuk memaksimalkan atau meminimumkan dari himpunan kandidat menjadi solusi. Dalam permasalahan ini himpunan kandidat akan melalui fungsi objektif untuk ditentukan mana kandidat yang memiliki keuntungan terbesar. Jadi, fungsi objektif akan diterapkan untuk memaksimalkan keuntungan yang didapat dari himpunan kandidat.

   Keuntungan dalam fungsi objektif akan sangat berkaitan dengan jarak antara *bot* dengan koordinat tujuan. Semakin dekat jaraknya dan semakin tinggi poin dari objek tujuan, maka keuntungannya akan semakin besar. Karena *bot* hanya bisa bergerak ke atas, bawah, kanan, atau kiri, maka jarak antara *bot* dengan koordinat tujuan akan dihitung menggunakan *Manhattan Distance.* Misalkan posisi *bot* saat ini adalah (x_1, y_1) dan koordinat tujuan adalah (x_2, y_2), maka *Manhattan Distance* akan dihitung melalui persamaan sebagai berikut.

$$Distance = |x_1 - x_2| + |y_1 - y_2|$$

Jarak yang telah didapatkan hanyalah salah satu komponen untuk menghitung keuntungan dari gerakan yang dilakukan oleh *bot*. Namun, secara sederhana keuntungan yang didapatkan dapat dihitung menggunakan rumus:

$$Keuntungan = \frac{Poin\ Objek}{Jarak}$$

Namun, untuk setiap alternatif solusi akan memiliki fungsi objektif yang berbeda. Fungsi objektif untuk setiap alternatif solusi akan dijelaskan lebih lanjut pada bagian berikutnya.

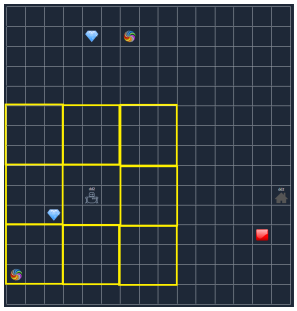
B. Alternatif Solusi

1. Algortima Greedy Blok

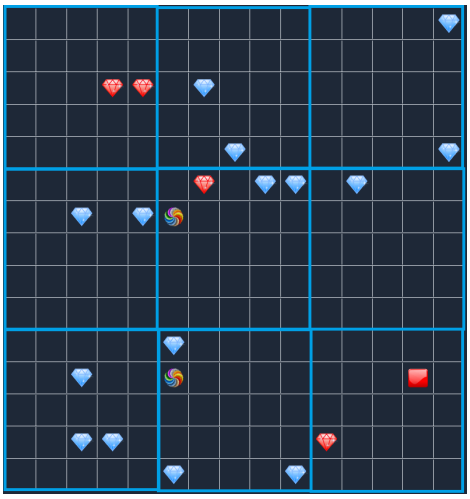
Ide dari strategi ini adalah pencarian *diamonds* akan dilakukan dari area di sekitar *bot* terlebih dahulu dan diikuti dengan pencarian *diamonds* pada satu papan permainan. Pencarian tersebut dilakukan dengan membagi-bagi lokasi pencarian menjadi beberapa blok. Tujuan dari pembagian lokasi pencarian menjadi blok adalah agar *diamond-diamond* yang berkelompok dapat diprioritaskan untuk diambil terlebih dahulu. Selain itu, apabila terdapat beberapa blok dengan nilai keuntungan yang sama, maka akan diprioritaskan blok terdekat terlebih dahulu.

a. Himpunan kandidat (C)

Himpunan kandidat dalam strategi ini akan berisi blok-blok berukuran 3x3 dan 5x5 yang nantinya akan dihitung nilai terbaiknya berdasarkan *diamonds* di dalamnya. Terdapat 9 blok 3x3 dan 9 blok 5x5 yang dapat dilihat lebih jelas pada ilustrasi berikut.



Gambar 3.2.1 Ilustrasi blok 3x3



Gambar 3.2.2. Ilustrasi blok 5x5

b. Himpunan solusi (S)

Himpunan solusi berisi sekumpulan koordinat (satu atau lebih koordinat) dalam satu blok yang memberikan keuntungan paling besar ketika didatangi oleh *bot*.

c. Fungsi solusi

Fungsi solusi pada strategi ini akan melakukan pengecekan terlebih dahulu apakah di dalam blok 3x3 masih terdapat *diamonds* yang bisa diambil oleh *bot* atau tidak. Jika ternyata tidak ditemukan satupun *diamonds* pada blok 3x3, maka akan dilakukan pengecekan selanjutnya pada blok 5x5. Dalam hal ini, tidak mungkin ada kejadian dimana dalam seluruh blok 3x3 maupun blok 5x5 tidak ditemukan satupun *diamonds*.

d. Fungsi seleksi

Fungsi seleksi pada strategi ini akan terbagi menjadi dua tahap, yakni pencarian terhadap blok 3x3 dan pencarian terhadap blok 5x5. Pencarian terhadap blok 3x3 akan diprioritaskan berdasarkan strategi heuristik yang kami gunakan. Dalam melakukan pencarian nilai terbaik dalam sebuah blok baik blok 3x3 maupun blok 5x5, strategi yang kami gunakan adalah menjumlahkan semua nilai *diamonds* yang ada pada blok tersebut dibagi dengan jarak *diamonds* tersebut ke *bot*. Hal ini sama saja seperti menjumlahkan semua nilai keuntungan *diamonds* yang berada dalam satu blok. Setelah itu, akan diambil blok yang memiliki jumlah

nilai keuntungan paling besar dan solusi terbaik berupa kumpulan koordinat *diamonds* dalam blok terbaik berhasil didapatkan.

e. Fungsi kelayakan

Fungsi kelayakan pada strategi ini adalah melakukan pengecekan terhadap nilai *diamonds* yang hendak dituju. Apabila nilai *diamond* yang sedang dituju bernilai dua sedangkan *inventory bot* yang tersisa hanya satu *diamond*, maka kandidat tersebut tidak akan diterima sebagai solusi.

f. Fungsi objektif

Fungsi objektif pada strategi ini berfungsi untuk memaksimalkan nilai keuntungan pada kumpulan koordinat solusi. Hal tersebut dilakukan dengan ikut memperhitungkan jarak *bot* ke koordinat solusi selain memperhitungkan nilai keuntungan blok. Hal tersebut dapat mengoptimalkan solusi terbaik yang dapat diraih.

2. Algoritma Greedy Direct Diamond

Ide dari strategi ini adalah kita melakukan pencarian ke seluruh *diamonds* yang ada pada papan permainan dan menghitung nilai keuntungan terbaik yang dapat diambil.

a. Himpunan kandidat (**C**)

Himpunan kandidat dalam strategi ini akan berisi kumpulan koordinat semua *diamonds* yang ada pada papan permainan. Selain itu himpunan ini juga akan berisi posisi bot musuh lain.

b. Himpunan solusi (**S**)

Himpunan solusi berisi koordinat *diamond* dalam papan permainan yang dapat memberikan keuntungan paling besar ketika didatangi oleh *bot*.

c. Fungsi solusi

Fungsi solusi pada strategi ini adalah dengan melakukan pengecekan terhadap *inventory* dan nilai suatu *diamond*. Apabila sebuah *diamond* memiliki nilai dua dan *inventory bot* hanya tersisa satu buah *diamond*, maka *diamond* tersebut tidak dapat dianggap sebagai salah satu solusi.

d. Fungsi seleksi

Fungsi seleksi pada algoritma ini adalah dengan menghitung nilai

keuntungan terbaik dari seluruh *diamond* dengan cara membagi nilai sebuah *diamond* dengan jarak antara *diamond* tersebut dengan *bot*. *Diamond* dengan nilai keuntungan terbesar lah yang akan menjadi solusi terbaik pada strategi ini.

e. Fungsi kelayakan

Fungsi kelayakan pada strategi ini adalah melakukan pengecekan terhadap nilai *diamonds* yang hendak dituju dan *inventory* yang tersisa. Apabila nilai suatu *diamond* adalah dua sedangkan *inventory* yang tersisa hanya ada satu, maka *diamond* tersebut dianggap tidak layak menjadi sebuah solusi.

f. Fungsi objektif

Fungsi objektif pada strategi ini berfungsi untuk memaksimalkan nilai keuntungan pada koordinat solusi. Hal tersebut dilakukan dengan ikut memperhitungkan jarak *bot* ke koordinat solusi selain memperhitungkan nilai keuntungan blok. Selain itu, dalam mengoptimasi solusi juga dapat dilakukan dengan menerapkan strategi *teleporter*, yakni mencari jarak sebuah *diamond* ke *bot* melalui *teleporter.*

3. Algoritma Greedy Tackle

Ide dari algoritma ini muncul karena ada fitur *tackle* yang bisa dilakukan. *Tackle* bisa dilakukan dengan menempati koordinat musuh. Ketika *tackle* berhasil, maka *diamonds* musuh akan terambil hingga *inventory bot* menjadi penuh, dan *bot* musuh akan kembali ke *base*-nya. Tentu gerakan ini akan memberikan keuntungan sangat besar, mulai dari bisa mengambil diamond dari musuh, maksimal 5 *diamonds* hanya dalam 1 gerakan, dan akan membuat musuh bergerak kembali dari base sehingga memerlukan banyak waktu untuk mengumpulkan *diamonds* kembali. Lalu dari analisis tersebut muncullah pertanyaan, kenapa bot harus repot - repot mencari *diamonds*, jika musuh bisa mencarikan *diamonds*, dan *bot* tinggal men-*tackle*-nya? Berikut adalah elemen - elemen *greedy* dari pendekatan ini.

a. Himpunan Kandidat (C)

Himpunan kandidat dalam pendekatan ini akan berisi koordinat *base* - *base* musuh yang ada di dalam *board*. *Base* musuh dipilih karena musuh yang

base-nya masing - masing. Jadi yang perlu dilakukan hanyalah menunggu di *base* musuh hingga musuh mendekat. Alasan lainnya adalah ketika menjadikan posisi musuh sebagai tujuan, maka ada kemungkinan *bot* tidak bisa mengejar musuh karena posisi musuh selalu berubah - ubah dan sulit untuk diprediksi.

b. Himpunan Solusi (S)

Himpunan solusi akan berisi koordinat *base* musuh yang paling mungkin untuk dilakukan *tackle*.

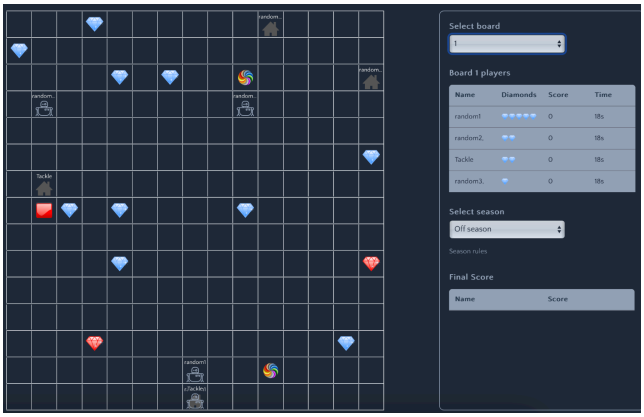
c. Fungsi Solusi

Pendekatan ini tidak memiliki fungsi solusi karena koordinat *base* musuh yang telah dipilih melalui fungsi seleksi, kelayakan, dan objektif dipastikan valid.

d. Fungsi Seleksi

Fungsi seleksi akan dilakukan untuk menentukan *base* musuh yang paling menguntungkan. Seleksi akan dilakukan dengan melihat musuh dengan *diamonds* paling banyak dan menjadikan *base* musuh tersebut sebagai target. Semakin banyak *diamonds* yang dimiliki musuh, maka semakin menguntungkan ketika *tackle* dilakukan.

Sebagai contoh, situasi gambar di bawah adalah ketika bot "tackle" sudah di base "random1" dikarenakan bot "random1" sudah mempunyai banyak *diamonds,* oleh karena itu *base* tersebut jadi target dan siap untuk men-*tackle*.



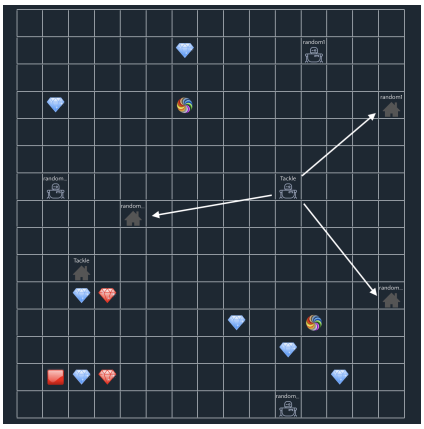
Gambar 3.2.3 Ilustrasi Situasi Fungsi Seleksi

e. Fungsi Kelayakan

Fungsi kelayakan akan digunakan untuk menentukan apakah *base* musuh

bisa menjadi target selanjutnya. Fungsi ini akan menghitung jarak *bot* saat ini ke *base* musuh dan membandingkannya dengan jarak musuh ke *base*-nya sendiri. Apabila *bot* lebih dekat ke *base* musuh daripada musuh pemilik *base* itu sendiri, maka koordinat tersebut layak untuk menjadi solusi.

f. Fungsi objektif



Gambar 3.2.4 Ilustrasi Objektif Titik Tengah

Fungsi ini berguna untuk meminimalkan jarak *bot* saat ini ke semua *base* musuh dengan cara menghitung titik tengah dari semua *base* musuh. Hal ini dilakukan karena ketika permainan dimulai, tentu semua musuh perlu waktu untuk mencari *diamonds* terlebih dahulu. Waktu ini akan digunakan *bot* untuk ke titik tengah *base* musuh dengan harapan *bot* bisa segera ke *base* musuh dengan *diamonds* terbanyak, entah dimanapun posisi *base*-nya. Titik tengah dari *base* - *base* musuh akan dihitung dengan rumus:

$$x_{mid} = \frac{x_1 + x_2 + \ldots + x_n}{n}, \quad y_{mid} = \frac{y_1 + y_2 + \ldots + y_n}{n}.$$

4. Algoritma Tambahan

Ada beberapa algoritma tambahan yang penulis pertimbangkan untuk membuat *bot* bekerja menjadi lebih efisien dan menghindari beberapa kemungkinan *bug* ataupun *loop* yang bisa terjadi. Algoritma tambahan harus diterapkan bersamaan dengan algoritma *greedy* yang lain karena algoritma ini hanya berfungsi untuk mengefisienkan gerakan. Berikut adalah beberapa algoritma tambahan yang dibuat:

a. Pemanfaatan Teleport

Algoritma dibuat karena ada objek *teleport* di dalam game. Algoritma ini akan memanfaatkan objek teleport untuk mencari jarak yang lebih dekat untuk mendapatkan *diamond* atau untuk kembali ke *base*. Algoritma bekerja dengan terlebih dahulu mencari posisi teleport terdekat dan terjauh,. Kemudian, algoritma melakukan iterasi pada setiap *diamond* dalam *board*. Pada setiap iterasi, jarak antara *diamond* dan posisi *teleport* terjauh, posisi *teleport* terdekat, serta posisi bot dihitung untuk dihasilkan *score* berdasarkan jumlah poin yang dimiliki oleh *diamond* tersebut. Hasil akhir yang dikembalikan oleh fungsi ini adalah jarak terpendek, posisi *diamond* terdekat, dan posisi *teleport* terdekat. Dengan demikian, algoritma ini membantu bot dalam menentukan gerakan yang akan selanjutnya dilakukan.

b. Menghindari Object dalam Path

Algoritma ini bekerja untuk menentukan apakah terdapat *obstacle* di jalur yang menghubungkan posisi bot saat ini dengan posisi *goal* dalam game. Setelah itu, algoritma melakukan iterasi pada objek-objek *obstacle* dan memeriksa apakah posisi objek tersebut berada dalam lintasan bot ke *goal*. Jika berada dalam lintasan, algoritma menentukan posisi tujuan alternatif yang dapat dijangkau oleh bot untuk menghindari rintangan tersebut. Tujuan alternatif ini akan berada di atas, bawah, kanan, atau kiri dari posisi *goal*. Posisi tujuan alternatif ini kemudian disimpan dan diperbarui dalam variabel goal_position. Algoritma ini akan membantu bot dalam melewati *obstacle* yang mungkin ada dalam permainan sehingga tidak menimbulkan perubahan *goal* dan menghindari error.

C. Analisis Efisiensi dan Efektivitas Alternatif Solusi

Berdasarkan rancangan algoritma yang telah dibuat, telah dilakukan proses pengetesan untuk menilai seberapa baik rancangan algoritma bekerja. Proses pengetesan akan dilakukan dengan proses - proses sebagai berikut.

1. Bot dijalankan *single player* pada board sebanyak 5 kali dan dihitung rata-rata diamonds yang didapatkan. Game akan dijalankan selama 60 detik dengan delay tiap gerakannya adalah 100 ms. Khusus untuk bot *tackle*, tes ini akan dilakukan 1 vs 1 dengan bot lain karena bot *tackle* tidak akan bisa berjalan jika tidak ada musuh.
2. Ketiga bot akan dijalankan bersamaan dan akan dinilai berdasarkan *leaderboard* yang tersedia di dalam *game*. Tes ini akah dilakukan sebanyak 5 kali.

Dengan proses diatas, diharapkan efektivitas dari setiap *bot* dapat dinilai secara objektif. Berikut adalah hasil pengetesan efisiensi yang telah dilakukan terhadap algoritma *greedy block* dan algoritma *greedy direct* yang dijalankan secara *single player*.

Table 3.3.1 Hasil Tes Algoritma Greedy Blok

Percobaan	Diamonds yang diperoleh
1	80
2	93
3	79
4	67
5	77
Rata-rata	79,2

Tabel 3.3.2 Hasil Tes Algoritma Greedy Direct Diamond

Percobaan	Diamonds yang diperoleh
1	86
2	93
3	105
4	113
5	106
Rata-rata	100,6

Berikut adalah hasil pengetesan yang telah dilakukan terhadap algoritma *greedy tackle* dengan sistem 1 vs 1 terhadap bot lain, dalam tes ini bot yang digunakan adalah bot *block*.

Tabel 3.3.3 Hasil Tes Algoritma Greedy Tackle

Percobaan	Diamonds yang diperoleh
1	45
2	40
3	60
4	35
5	45
Rata-rata	45

Berikut adalah hasil pengetesan yang telah dilakukan terhadap algoritma *greedy block, direct,* dan *tackle* yang dijalankan secara bersamaan.

Tabel 3.3.4 Hasil Tes Algoritma Greedy Block, Direct, dan Tackle Bersamaan

Percobaan	Diamonds yang diperoleh		
	Block	Direct	Takle
1	70	99	10
2	61	98	5
3	63	87	0
4	51	96	0
5	51	100	0
Rata-rata	59,2	96	3

D. Strategi Greedy Terpilih

Berdasarkan hasil pengetesan yang telah dilakukan, strategi *greedy* yang dipilih adalah *Greedy Direct* Diamond. Alasan dipilihnya algoritma ini adalah sebagai berikut.

1. Dalam pengetesan *single player* yang telah dilakukan, algoritma ini mendapatkan score yang lebih banyak daripada algoritma lain dengan jarak yang cukup jauh dengan rata - rata 100,6. Walaupun algoritma *tackle* sangat baik digunakan untuk 1 vs 1, namun sebagian besar game akan dilakukan dengan pemain lebih dari 2 bot karena kurang efektif ketika digunakan untuk bermain *multiplayer* karena bot hanya bisa mentarget 1 musuh dan musuh lain akan bebas untuk mengumpulkan *diamond*
2. Algoritma ini telah memanfaatkan hampir semua elemen yang ada di dalam game Diamonds. Elemen - elemen tersebut termasuk *red button* dan *teleporter*. *Red button* akan banyak digunakan oleh bot dengan tujuan untuk merugikan lawan dan *teleporter* digunakan untuk mempersingkat jarak untuk mendapatkan *diamonds* atau jarak untuk kembali ke *base*.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

A. Implementasi Algoritma Greedy

```
class GreedyDiamondLogic(BaseLogic: BaseLogic)
{ class utama implementasi algoritma Greedy untuk
mengambil/mencari diamond }

KAMUS LOKAL

static static_goals: array of Position
static static_goal_teleport: GameObject
static static_temp_goals: Position
static static_direct_to_base_via_teleporter: boolean
procedure __init__()
function next_move(board_bot: GameObject, board: Board) ->
(integer, integer)
function calculate_near_base() -> boolean
procedure find_nearest_diamond()
function find_nearest_red_button() -> (integer, Position)
function find_nearest_teleport() -> (Position, Position,
GameObject)
function find_other_teleport(teleport: GameObject) -> Position
function find_nearest_diamond_teleport() -> (real, (Position,
Position), GameObject)
function find_nearest_diamond_direct() -> (real, Position)
```

```
procedure __init__()
{ prosedur initiation untuk class GreedyDiamondLogic }

KAMUS LOKAL
directions: array of (integer, integer)
goal_position: Position
current_direction: integer
distance: real

ALGORITMA
directions <- [(1,0), (0,1), (-1,0), (0,-1)]
goal_position <- Nil
current_direction <- 0
distance <- 0
```

```
function next_move(board_bot: GameObject, board: Board) ->
(integer, integer)
{ fungsi untuk menentukan langkah berikutnya dari bot }

KAMUS LOKAL
delta_x: integer
delta_y: integer
i: integer
j: integer
props: Properties
board: Board
board_bot: GameObject
diamonds: array of GameObject
bots: array of GameObject
teleporter: array of GameObject
redButton: array of GameObject
enemy: array of GameObject
enemyDiamond: array of integer

ALGORITMA
props <- board_bot.properties
board <- board
board_bot <- board_bot
diamonds <- board.diamonds
bots <- board.bots

i <- 0
j <- 0
repeat
  if (board.game_objects[i].type = "TeleportGameObject") then
    teleporter[j] <- board.game_objects[i]
    j <- j + 1
  i <- i + 1
until board.game_objects[i] = Nil

i <- 0
j <- 0
repeat
  if (board.game_objects[i].type = "DiamondButtonGameObject")
then
   redButton[j] <- board.game_objects[i]
    j <- j + 1
  i <- i + 1
until board.game_objects[i] = Nil

i <- 0
j <- 0
repeat
  if (bots[i].id = board_bot.id) then
```

```
   enemy[j] <- bots[i]
    j <- j + 1
  i <- i + 1
until bots[i] = Nil

i <- 0
repeat
  enemyDiamond[i] <- enemy[i].properties.diamonds
  i <- i + 1
until bots[i] = Nil

if (board_bot.position = board_bot.properties.base) then
  static_goals <- []
  static_goal_teleport <- Nil
  static_temp_goals <- Nil
  static_direct_to_base_via_teleporter <- false

if (static_goal_teleport and board_bot.position =
find_other_teleport(static_goal_teleport)) then
  static_goals.remove(static_goal_teleport.position)
  static_goal_teleport <- Nil

if (not static_goal_teleport and board_bot.position) then
  i <- 0
  j <- 0
  repeat
    if (static_goals[i] == board_bot.position) then
      static_goals.remove(board_bot.position)
      j <- 1
    i <- i + 1
  until static_goals[i] = Nil or j = 1

if (board_bot.position = static_temp_goals) then
  static_temp_goals <- Nil

if (props.diamond = 5 or (props.miliseconds_left < 5000 and
props.diamonds > 1)) then
  goal_position <- find_best_way_to_base()
  if (not static_direct_to_base_via_teleporter) then
    static_goals <- []
    static_goal_teleport <- Nil
else
  if (length(static_goals) = 0) then
    find_nearest_diamond()
  goal_position <- static_goals[0]

if (calculate_near_base() and props.diamonds > 2) then
  goal_position <- find_best_way_to_base()
  if (not static_direct_to_base_via_teleporter) then
    static_goals <- []
    static_goal_teleport <- Nil
```

```
if (static_temp_goals) then
  goal_position <- static_temp_goals

current_position <- board_bot.position
if (goal_position) then
  if (not static_temp_goals) then
    obstacle_on_path("teleporter", current_position.x,
current_position.y, goal_position.x, goal_position.y)
  if (props.diamonds = 4) then
    obstacle_on_path("redDiamond", current_position.x,
current_position.y, goal_position.x, goal_position.y)
  (delta_x, delta_y) <- get_direction(current_position.x,
current_position.y, goal_position.x, goal_position.y)
else
  delta <- directions[currect_direction]
  delta_x <- delta[0]
  delta_y <- delta[1]
  current_direction <- (current_direction + 1) mod
length(directions)

if (delta_x = 0 and delta_y = 0) then
  static_goals <- []
  static_direct_to_base_via_teleporter <- false
  static_goal_teleport <- Nil
  static_temp_goals <- Nil
  goal_position <- Nil
  tempMove <- next_move(board_bot, board)
  (delta_x, delta_y) <- (tempMove[0], tempMove[1])

-> (delta_x, delta_y)
```

```
function find_best_way_to_base() -> Position
{ fungsi mencari jalan terbaik ke base }

KAMUS LOKAL
current_position: Position
base: Base
base_position: Position
base_distance_direct: integer
nearest_teleport_position: Position
far_teleport_position: Position
nearest_tp: GameObject
base_distance_teleporter: integer

ALGORITMA
current_position <- board_bot.position
base <- board_bot.properties.base
```

```
base_position <- (base.y, base.x)
base_distance_direct <- abs(base.x - current_position.x) -
abs(base.y - current_position.y)
(nearest_teleport_position, far_teleport_position, nearest_tp) <-
find_nearest_teleport()
base_distance_teleporter <- (abs(base.x -
far.teleport.position.x) + abs(base.y - far.teleport.position.y)
+ abs(nearest_teleport_position.x - current_position.x) +
abs(nearest_teleport_position.y - current_position.y))
if (base_distance_direct < base_distance_teleporter) then
  -> base.position
else
  static_direct_to_base_via_teleporter <- true
  static_goal_teleport <- nearest_tp
  static_goals <- [nearest_teleport_position, base]
  -> nearest_teleport_position
```

```
function calculate_near_base() -> boolean
{ fungsi untuk mengkalkulasi jarak antara base, return true
kalau base lebih dekat dibanding lokasi-lokasi diamond dan sudah
mengambil 2 diamond atau lebih }

KAMUS LOKAL
current_position: Position
base: Base
base_distance: integer
base_distance_teleporter: integer

ALGORITMA
current_position <- board_bot.position
base <- board_bot.properties.base
base_distance <- abs(base.x - current_position.x) + abs(base.y -
current_position.y)
base_distance_teleporter <- find_base_distance_teleporter()
if (base_distance_teleporter < base_distance) then
  distance <- base_distance_teleporter
else
  distance <- base_distance

-> distance < self.distance
```

```
function find_base_distance_teleporter() -> integer
{ fungsi untuk mengkalkulasi jarak antara base dan teleporter }

KAMUS LOKAL
```

```
current_position: Position
nearest_teleport_position: Position
far_teleport_position: Position
nearest_teleport: GameObject
base: Base
base_distance_teleporter: integer

ALGORITMA
current_position <- board_bot.position
(nearest_teleport_position, far_teleport_position,
nearest_teleporter) <- find_nearest_teleport()
base <- board_bot.properties.base
base_distance_teleporter <- (abs(base.x -
far.teleport.position.x) + abs(base.y - far.teleport.position.y)
+ abs(nearest_teleport_position.x - current_position.x) +
abs(nearest_teleport_position.y - current_position.y))

-> base_distance_teleporter
```

```
procedure find_nearest_diamond()
{ prosedur mencari diamond terdekat }

KAMUS LOKAL
direct: integer, Position
teleport: (integer, (Position, Position))
redButton: integer, Position

ALGORITMA
direct <- find_nearest_diamond_direct()
teleport <- find_nearest_diamond_teleport()
redButton <- find_nearest_red_button()
if (direct[0] < teleport[0] and direct[0] < redButton[0]) then
  static_goals <- [direct[1]]
  distance <- direct[0]
else
  if (teleport[0] < direct[0] and teleport[0] < redButton[0])
then
    static_goals <- teleport[1]
    static_goal_teleport <- teleport[2]
    distance <- teleport[0]
  else
    static_goals <- [redButton[1]]
    distance <- redButton[0]
```

```
function find_nearest_red_button() -> (integer, Position)
```

```
{ fungsi mencari red button terdekat }

KAMUS LOKAL
current_position: Position
distance: integer

ALGORITMA
current_position <- board_bot.position
distance <- abs(redButton[0].position.x - current_position.x) +
abs(redButton[0].position.y - current_position.y)

-> (distance, redButton[0].position)
```

```
function find_nearest_teleport() -> (Position, Position,
GameObject)
{ fungsi mencari teleport terdekat }

KAMUS LOKAL
nearest_teleport_position: Position
far_teleport_position: Position
nearest_tp: GameObject
min_distance: real
i: integer

ALGORITMA
(nearest_teleport_position, far_teleport_position, nearest_tp) <-
(Nil, Nil, Nil)
min_distance <- 9999.9

i <- 0
repeat
  distance <- abs(teleporter[i].position.x -
board_bot.position.x) + abs(teleporter[i].position.y -
board_bot.position.y)
  if (distance < min_distance) then
    min_distance <- distance
    (nearest_teleport_position, far_teleport_position) <-
(teleporter[i].position, find_other_teleport(teleporter[i]))
    nearest_tp <- teleporter[i]
  i <- i + 1
until teleporter[i] = Nil
```

```
function find_other_teleport(teleport: GameObject) -> Position
{ fungsi mencari teleport lainnya }
```

```
KAMUS LOKAL
i: integer

ALGORITMA
i <- 0
repeat
  if (teleporter[i].id != teleport.id) then
    -> teleporter[i].position
  i <- i + 1
until teleporter[i] = Nil
```

```
function find_nearest_diamond_teleport() -> (real, (Position,
Position), GameObject)
{ fungsi mencari diamond dengan cara melewati teleporter }

KAMUS LOKAL
current_position: Position
nearest_teleport_position: Position
far_teleport_position: Position
nearest_teleport: GameObject
min_distance: real
nearest_diamond: array of Position
distance: integer
i: integer

ALGORITMA
current_position <- board_bot.position
(nearest_teleport_position, far_teleport_position,
nearest_teleport) <- find_nearest_teleport()
min_distance <- 9999.9
nearest_diamond <- Nil

repeat
  distance <- (abs(diamonds[i].position.x -
far_teleport_position.x) + abs(diamonds[i].position.y -
far_teleport_position.y) + abs(nearest_teleport_position.x -
current_position.x) + abs(nearest_teleport_position.y -
current_position.y))
  distance <- distance div diamonds[i].properties.points
  if (distance < min_distance and (diamonds[i],properties.points
= 2 and board_bot.properties.diamonds != 4) or
(diamonds[i].properties.points = 1)) then
    min_distance <- distance
    nearest_diamond <- [nearest_teleport_position,
diamonds[i].position]
until diamonds[i] = Nil

-> (min_distance, nearest_diamond, nearest_teleport)
```

```
function find_nearest_diamond_direct() -> (real, Position)
{ fungsi mencari diamond dengan cara langsung (tidak melewati
teleporter) }

KAMUS LOKAL
current_position: Position
min_distance: real
nearest_diamond: Position
distance: integer
i: integer

ALGORITMA
current_position <- board_bot.position
min_distance <- 9999.9
nearest_diamond <- Nil
i <- 0

repeat
  distance <- abs(diamonds[i].position.x - current_position.x) +
abs(diamonds[i].position.y - current_position.y)
  distance <- distance div diamonds[i].properties.points
  if (distance < min_distance and (diamonds[i],properties.points
= 2 and board_bot.properties.diamonds != 4) or
(diamonds[i].properties.points = 1)) then
    min_distance <- distance
    nearest_diamond <- diamonds[i].position
  i <- i + 1
until diamonds[i] = Nil

-> (min_distance, nearest_diamond)
```

```
procedure obstacle_on_path(type: string, current_x: integer,
current_y: integer, dest_x: integer, dest_y: integer)
{ prosedur akan digunakan ketika mau menuju sesuatu tetapi ada
suatu halangan di jalan }

KAMUS LOKAL
object_array: array of GameObject
Object: GameObject
i: integer
j: integer

ALGORITMA
depend on (type)
  type = "teleporter" : object_array <- teleporter
  type = "redDiamond" : (
```

```
    i <- 0
    j <- 0
    repeat
      if (diamonds[i].properties.points = 2) then
        object_array[j] <- diamonds[i]
        j <- j + 1
      i <- i + 1
    until diamonds[i] = Nil
    )
  type = "RedButton" : object <- redButton

i <- 0
repeat
    object <- object_array[i]

    if current_x == object.position.x and current_y ==
object.position.y then
        i <- + 1
        continue
    if object.position.x == dest_x and (dest_y <
object.position.y <= current_y or current_y <= object.position.y
< dest_y):
        if dest_x != current_x then
            if dest_x > current_x then
                goal_position <- Position(dest_y, dest_x - 1)
            else
                goal_position <- Position(dest_y, dest_x + 1)
        else
            if dest_x <= 1 then
                goal_position <- Position(dest_y, dest_x + 1)
            else
                goal_position <- Position(dest_y, dest_x - 1)
        static_temp_goals <- goal_position
    else if object.position.y == dest_y and (dest_x <
object.position.x <= current_x or current_x <= object.position.x
< dest_x):
        if dest_y != current_y then
            if dest_y > current_y:
                goal_position <- Position(dest_y - 1, dest_x)
            else
                goal_position <- Position(dest_y + 1, dest_x)
        else
            if dest_y <= 1 then
                goal_position <- Position(dest_y + 1, dest_x)
            else
                goal_position <- Position(dest_y - 1, dest_x)
        static_temp_goals <- goal_position
    else if object.position.y == current_y and (dest_x <
object.position.x <= current_x or current_x <= object.position.x
< dest_x) then
        if dest_y != current_y then
```

```
            goal_position = Position(dest_y, current_x)
        else
            if current_y <= 1:
                goal_position <- Position(current_y + 1,
current_x)
            else
                goal_position <- Position(current_y - 1,
current_x)
        static_temp_goals <- goal_position
    i <- i + 1

until index >= length(object_array)
```

B. Penjelasan Struktur Data

Program ini menggunakan beberapa struktur data yang telah tersedia di dalam game dan beberapa variabel yang memanfaatkan struktur data tersebut. Berikut adalah rincian struktur data dan variabel yang dipakai dalam program.

1. Position

```
class Position:
    y: integer
    x: integer
```

Kelas ini menyimpan informasi tentang posisi dalam permainan, yang terdiri dari koordinat x dan y.

2. Game Object

```
class Properties:
    points: integer = None
    pair_id: string = None
    diamonds: integer = None
    score: integer = None
    name: string = None
    inventory_size: integer = None
    can_tackle: boolean = None
    milliseconds_left: integer = None
    time_joined: string = None
    base: Base = None
```

```
class GameObject:
    id: integer
    position: Position
    type: string
    properties: Properties = None
```

Kelas ini merepresentasikan objek dalam permainan, seperti bot, *teleporter*, *diamonds*, *base,* dan *redButton*. Atributnya mencakup: id, *position* yang merupakan instance dari kelas Position , tipe objek dalam *string*, dan *properties* yang berisi informasi tambahan dari objek tersebut, seperti poin, jumlah berlian, dan lain-lain yang merupakan *instance* dari Kelas Properties.

Selanjutnya ada, kelas Properties yang digunakan untuk menyimpan properti tambahan dari objek.. Properti tersebut bisa berupa poin, id pasangan (pair id), jumlah *diamonds*, skor, nama, ukuran *inventory*, sisa waktu dalam milidetik, waktu bergabung, dan posisi *base*.

3. Variabel

```
static_goals : array of Position = []
static_goal_teleport : GameObject = None
static_temp_goals : Position = None
static_direct_to_base_via_teleporter : boolean = False
goal_position: Position = None
distance: int = 0
props: Properties
board: array of GameObject
board_bot: GameObject
Diamonds: array of GameObject
bots: array of GameObject
teleporter: array of GameObject
redButton: array of GameObject
enemy: array of GameObject
enemyDiamond: array of GameObject
```

Berikut adalah penjelasan dari masing-masing variabel dan fungsinya dalam program.

a. *static_goals*: List yang menyimpan kemungkinan posisi tujuan-tujuan yang ingin dicapai oleh bot.
b. static_goal_teleport: Objek yang menyimpan informasi tentang *teleporter* yang menjadi tujuan bot.
c. static_temp_goals: Objek yang menyimpan informasi tentang posisi tujuan sementara. Ini digunakan untuk menyimpan posisi tujuan sementara ketika bot harus menghindari *obstacle* dalam lintasan.
d. *static_direct_to_base_via_teleporter*: Variabel boolean yang menandakan apakah bot harus langsung menuju ke basis melalui teleporter. Jika bernilai True, maka bot akan mencoba untuk langsung menuju ke basis melalui *teleporter*.
e. *goal_position*: Variabel yang menyimpan posisi yang ingin dicapai oleh bot dalam langkah selanjutnya.
f. *distance*: Variabel yang menyimpan jarak antara bot dan *goal.*
g. *props*: Objek yang menyimpan properti-properti yang terkait dengan bot, seperti jumlah *diamonds*, skor, dan lain lain. Properti-properti ini digunakan dalam penentuan langkah selanjutnya dalam *game*.
h. *board*: List yang berisi semua objek dalam papan permainan.
i. *board_bot*: Objek yang mewakili bot dalam *game*.
j. *diamonds*: List yang berisi semua objek *diamonds* dalam permainan.
k. *bots*: List yang berisi semua objek bot dalam permainan, termasuk bot-bot musuh.
l. *teleporter*: List yang berisi semua objek *teleporter* dalam permainan.
m. *redButton*: List yang berisi semua objek *red button* dalam permainan.
n. *enemy*: List yang berisi semua objek bot musuh dalam permainan.
o. *enemyDiamond*: List yang berisi informasi tentang jumlah *diamonds* yang dimiliki oleh setiap bot musuh.

C. Analisis Desain Solusi

Pengujian terhadap algoritma pertama-tama dilakukan dengan melakukan tes keoptimalan bot dalam mengambil diamond dalam waktu 60 detik tanpa musuh. Berikut adalah hasil pengetesan yang telah dilakukan.

Tabel 4.3.1 Hasil Tes Algoritma Terpilih

Percobaan	Diamonds yang diperoleh
1	112
2	93
3	97
4	98
5	106
Rata-rata	101,2

Berdasarkan hasil tersebut, bisa disimpulkan bahwa bot bisa bekerja cukup optimal dalam mengumpulkan *diamonds*. Bot juga berhasil melewati tes ini tanpa ada *bug* sama sekali. Bot berhasil menghindari *obstacle* dan memanfaatkan *teleport* dengan baik untuk mengambil *diamonds* dan kembali ke *base*. Proses perhitungan yang dilakukan juga cukup baik karena hanya memerlukan waktu sekitar $1x10^{-5}$ - $1x10^{-4}$ detik untuk menentukan langkah selanjutnya.

Tes selanjutnya akan dilakukan dengan melawan bot lain. Walaupun bot memiliki hasil yang optimal ketika dijalankan tanpa musuh, bukan berarti bot akan optimal ketika dijalankan pada *board* dengan bot - bot lain. Tes ini akan melakukan pengetesan bagaimana algoritma bekerja ketika kondisi *board* berubah termasuk jumlah *diamond* dan posisi *diamond*. Dalam tes ini, perilaku bot ketika terkena *tackle* juga bisa dianalisis. Tes ini akan dilakukan dengan cara memainkan bot bersamaan dengan 3 bot lain. Bot musuh yang pertama adalah bot dengan algoritma *block* yang telah dibuat sebelumnya, dan 2 bot lainnya adalah bot yang didesain oleh kelompok lain. Berikut adalah hasil tes yang dilakukan.

Tabel 4.3.2 Hasil Tes Algoritma Terpilih dengan Bot lain

Percobaan	Diamonds yang diperoleh			
	Direct	Block	Musuh 1	Musuh 2
1	110	41	102	105

2	80	73	77	13
3	102	67	73	90
4	111	61	90	46
5	89	71	73	116
Rata - rata	**98.4**	**62.6**	**83**	**74**

Berdasarkan hasil yang didapatkan, bot sudah berjalan dengan cukup optimal dan berhasil mengambil paling banyak diamonds pada percobaan 1, 3, dan 4. Hasil percobaan ini cukup bervariasi karena *diamonds* dan *base* di-*generate* secara random sehingga ada kemungkinan bot tertentu memiliki keuntungan dimana *base*-nya berposisi di tengah *map* dan/atau *diamond*s banyak di-*generate* di sekitar *base*.

Gambar 4.3.1 Bot Mengalami Error

Walaupun memberikan hasil yang cukup baik, bot sempat mengalami error pada percobaan ke-2. Bot terkena *tackle* dan kembali ke *base*. Bot seharusnya langsung menentukan gerakan selanjutnya, namun bot hanya diam dan program Python memunculkan error yaitu: "*RecursionError: maximum recursion depth exceeded while calling a Python object*". Error ini kemungkinan terjadi karena bot berusaha menentukan ulang gerakan selanjutnya karena posisinya tiba-tiba berubah, namun bot proses penentuan tidak berhenti.

Secara keseluruhan, algoritma bot ini sudah menunjukkan kemampuan yang baik dalam mengumpulkan *diamonds* dan menghindari *obstacle* saat bermain tanpa musuh. Namun, dalam *game* yang sesungguhnya ketika ada kehadiran bot musuh, performa bot menjadi lebih bervariasi dan berpotensi terjadi *error*. Untuk meningkatkan performa dan kestabilan bot, perlu dilakukan pengembangan lebih lanjut terutama dalam menangani skenario-skenario yang tidak terduga seperti terkena *tackle* oleh bot musuh.

BAB V
KESIMPULAN

A. Kesimpulan

Kesimpulan dari pengembangan algoritma greedy untuk menyelesaikan permainan diamonds adalah sebagai berikut:

1. Algoritma greedy cukup efektif dalam menyelesaikan game Diamonds, terbukti dengan rata-rata diamonds yang berhasil dikumpulkan sebanyak 101,2 dalam mode *single player*. Pendekatan greedy ini memungkinkan bot untuk membuat keputusan berdasarkan langkah yang paling menguntungkan pada setiap tahap permainan, hal tersebut termasuk memanfaatkan *teleporter* secara efisien dan menghindari *obstacle*.
2. Kehadiran bot musuh dalam permainan mempengaruhi performa algoritma *greedy*, dengan rata-rata *diamonds* yang berhasil dikumpulkan turun menjadi 98,6. Hal ini menunjukkan bahwa algoritma yang bekerja baik dalam mode *single player* belum tentu bekerja dengan baik ketika dihadapkan dengan bot musuh.
3. Algoritma greedy masih memerlukan pengembangan lebih lanjut. Pengembangan ini perlu difokuskan pada penanganan *edge case*, seperti kesalahan yang terjadi ketika bot terkena *tackle* dan terjadi *error*. Dengan melakukan pengembangan lebih lanjut, diharapkan performa dan kestabilan algoritma dapat ditingkatkan.

B. Saran

Ada beberapa hal yang bisa dilakukan untuk meningkatkan performa dan kestabilan algoritma greedy dalam menyelesaikan permainan Diamonds. Yang pertama adalah pengembangan harus lebih memperhatikan *edge case* yang dapat menyebabkan bot mengalami *error*. Yang kedua, algoritma bot harus lebih mempertimbangkan kehadiran musuh. Bot perlu memikirkan kejadian - kejadian yang mungkin ketika ada musuh. Yang ketiga adalah perlu dilakukan *testing* lebih banyak terhadap algoritma bot dalam berbagai skenario permainan untuk mengidentifikasi potensi kekurangan dan *error* yang memerlukan perbaikan sehingga bisa ditemukan solusi-solusi yang lebih optimal dan efektif. Dengan

menerapkan hal - hal di atas, diharapkan algoritma *greedy* dapat mengalami peningkatan performa dalam menyelesaikan permainan Diamonds.

LAMPIRAN

Tautan repository GitHub: https://github.com/ValentinoTriadi/Tubes1_MVP

Tautan Video: https://youtu.be/n5fvkAZ5i9M

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. Algoritma Greedy (Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf. Diakses pada 26 Februari 2024.

Munir, Rinaldi. 2024. Algoritma Greedy (Bagian 2). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf. Diakses pada 1 Maret 2024.

Munir, Rinaldi. 2024. Algoritma Greedy (Bagian 3). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag3.pdf. Diakses pada 4 Maret Februari 2024.