IF2211 Strategi Algoritma

# PEMANFAATAN ALGORITMA GREEDY DALAM PEMBUATAN BOT PERMAINAN ETIMO DIAMONDS

**Laporan Tugas Besar 1**

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma
pada Semester 2 (dua) Tahun Akademik 2023/2024


**Oleh**

| | |
|---|---|
| **Rizqika Mulia Pratama** | **13522126** |
| **Shabrina Maharani** | **13522134** |
| **Auralea Alvinia Syaikha** | **13522148** |

**Kelompok ALL IN RM**

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

**2024**

# DAFTAR ISI

# BAB 1 DESKRIPSI TUGAS

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


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*.

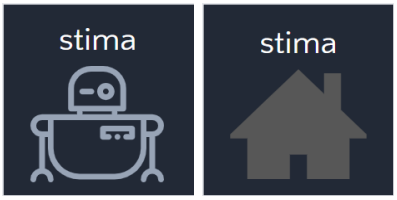
2. **Red Button/Diamond Button**

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**

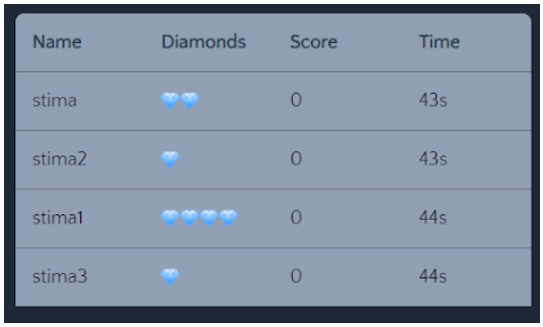
Terdapat 2 *teleporter* yang saling terhubung satu sama lain. Jika bot melewati sebuah *teleporter* maka bot akan berpindah menuju posisi *teleporter* yang lain.

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**




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.

**Spesifikasi Tugas Besar 1**

- Buatlah program sederhana dalam bahasa **Python** yang mengimplementasikan ***algoritma Greedy*** pada *bot* permainan Diamonds dengan tujuan memenangkan permainan.
- Tugas dikerjakan berkelompok dengan anggota minimal 2 orang dan maksimal 3 orang, boleh lintas kelas dan lintas kampus.
- Strategi *greedy* yang diimplementasikan setiap kelompok harus dikaitkan dengan fungsi objektif dari permainan ini, yaitu memenangkan permainan dengan memperoleh *diamond* sebanyak banyak nya dan jangan sampai *diamond* tersebut diambil oleh bot lain. Buatlah strategi *greedy* terbaik, karena setiap bot dari masing-masing kelompok akan diadu dalam kompetisi Tubes 1.
- Strategi *greedy* yang kelompok anda buat harus dijelaskan dan ditulis secara eksplisit pada laporan, karena akan diperiksa saat demo apakah strategi yang dituliskan sesuai dengan yang diimplementasikan. Tiap kelompok dapat menggunakan kreativitas yang bermacam macam dalam menyusun strategi *greedy* untuk memenangkan permainan. Implementasi pemain harus dapat dijalankan pada *game engine* yang telah disebutkan diatas serta dapat dikompetisikan dengan bot dari kelompok lain.
- Program harus mengandung komentar yang jelas, dan untuk setiap strategi *greedy* yang disebutkan, harus dilengkapi dengan kode sumber yang dibuat.
- Mahasiswa dilarang menggunakan kode program yang diunduh dari Internet. Mahasiswa harus membuat program sendiri, diperbolehkan untuk belajar dari program yang sudah ada.
- Mahasiswa dianggap sudah melihat dokumentasi dari *game engine,* sehingga tidak terjadi kesalahpahaman spesifikasi antara mahasiswa dan asisten.
- BONUS (maks 10): Membuat video tentang aplikasi *greedy* pada bot serta simulasinya pada game kemudian mengunggahnya di Youtube. Video dibuat harus memiliki audio dan menampilkan wajah dari setiap anggota kelompok. Untuk contoh video tubes stima

tahun-tahun sebelumnya dapat dilihat di Youtube dengan kata kunci “Tubes Stima”, “strategi algoritma”, “Tugas besar stima”, dll.

- Jika terdapat kesulitan selama mengerjakan tugas besar sehingga memerlukan bimbingan, maka dapat melakukan asistensi tugas besar kepada asisten (opsional). Dengan catatan asistensi hanya bersifat membimbing, bukan memberikan “jawaban”.
- Terdapat juga demo dari program yang telah dibuat. Pengumuman tentang demo menunggu pemberitahuan lebih lanjut dari asisten.
- Bot yang telah dibuat akan dikompetisikan dengan kelompok lain dan disaksikan oleh seluruh peserta kuliah. Terdapat hadiah menarik bagi kelompok yang memenangkan kompetisi.
- Program disimpan dalam repository yang bernama **Tubes1_NamaKelompok** dengan nama kelompok sesuai dengan yang di sheets diatas. Berikut merupakan struktur dari isi repository tersebut:
    a. Folder src berisi **semua bagian yang ada di bot-starter pack**
    b. Folder doc berisi laporan tugas besar dengan format **NamaKelompok.pdf**
    c. README untuk tata cara penggunaan yang minimal berisi:
        i. Penjelasan singkat algoritma *greedy* yang diimplementasikan
        ii. Requirement program dan instalasi tertentu bila ada
        iii. Command atau langkah-langkah dalam meng-compile atau build program
        iv. Author (identitas pembuat)

# BAB 2 LANDASAN TEORI

## 2.1 Algoritma Greedy

Algoritma *greedy* menjadi salah satu pendekatan paling umum dan sederhana dalam menyelesaikan masalah optimasi. Persoalan optimasi (*optimization problems*) yang dimaksud adalah persoalan untuk mencari solusi yang optimal. Persoalan optimasi hanya terdapat dua macam, yaitu maksimasi (*maximization*) dan minimasi (*minimization*).

Algoritma *greedy* merupakan pendekatan yang cenderung rakus atau tamak, dengan prinsip "*take what you can get now!*" atau dapat dikatakan algoritma *greedy* akan mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa perlu memperhatikan konsekuensi dari sebuah pilihan ke depannya. Dalam algoritma ini, solusi dibangun langkah demi langkah (*step by step*), di mana pada setiap langkah harus diambil keputusan terbaik tanpa bisa kembali ke langkah sebelumnya. Pada setiap langkah, pilihan terbaik dipilih dengan harapan mencapai solusi optimum global dengan memilih optimum lokal. Meskipun demikian, keputusan yang diambil mungkin hanya menghasilkan solusi sub-optimum atau pseudo-optimum. Alasannya adalah algoritma *greedy* tidak mengevaluasi secara menyeluruh semua kemungkinan solusi yang ada dan terdapat berbagai fungsi seleksi yang berbeda, sehingga pemilihan fungsi yang tepat menjadi kunci dalam mencapai solusi optimal. Dengan demikian, pada beberapa kasus, algoritma *greedy* tidak selalu dapat memberikan solusi yang optimal, namun hanya solusi sub-optimum.

Jika kebutuhan akan solusi paling optimal tidak terlalu mendesak, algoritma *greedy* dapat menjadi pilihan untuk menghasilkan solusi hampiran. Hal ini lebih disukai daripada menggunakan algoritma yang membutuhkan waktu komputasi eksponensial untuk mencapai solusi yang eksak. Sebagai contoh, dalam mencari tur dengan bobot minimal pada persoalan Travelling Salesman Problem (TSP) dengan jumlah simpul yang besar, algoritma brute force memerlukan waktu komputasi yang sangat lama untuk menemukan solusi eksaknya. Dengan algoritma *greedy*, meskipun tidak selalu menjamin penemuan tur dengan bobot minimal, namun solusi yang dihasilkan dianggap sebagai pendekatan solusi optimal. Namun, jika algoritma *greedy* mampu menghasilkan solusi optimal, keoptimalannya harus dapat dibuktikan secara matematis. Membuktikan optimalitas algoritma *greedy* secara matematis merupakan tantangan tersendiri. Lebih mudah untuk menunjukkan bahwa algoritma *greedy* tidak selalu optimal dengan mengemukakan *counterexample*, yaitu contoh kasus yang menunjukkan solusi yang dihasilkan tidak optimal daripada menunjukkan bahwa algoritma *greedy* menghasilkan hasil yang optimal.

Dengan demikian, algoritma *greedy* mengevaluasi banyak pilihan pada setiap langkahnya, mengambil keputusan yang diharapkan akan membawa ke solusi terbaik secara keseluruhan.

### 2.1.1 Elemen-elemen Algoritma Greedy

Algoritma *greedy* memiliki beberapa komponen elemen diantaranya, yaitu

1. Himpunan kandidat

   Himpunan kandidat yang umumnya dinotasikan dengan simbol C ini berisi kandidat kemungkinan yang akan dipilih pada setiap langkah dalam algoritma *greedy* (Misalnya, simpul/sisi di dalam job, graf, task, koin, benda, karakter, dan contoh lainnya).
2. Himpunan solusi

   Pada umumnya, himpunan solusi dinotasikan dengan simbol S. Himpunan solusi ini berisi kandidat kemungkinan yang sudah dipilih.
3. Fungsi solusi

   Fungsi solusi merupakan fungsi yang akan menentukan apakah suatu himpunan kandidat yang dipilih sudah menghasilkan atau memberikan solusi.
4. Fungsi seleksi (*selection function)*

   Fungsi seleksi merupakan fungsi yang digunakan untuk memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* dalam fungsi ini bersifat heuristik.
5. Fungsi kelayakan (*feasible*)

   Fungsi kelayakan digunakan untuk memeriksa apakah suatu kandidat yang dipilih dapat dengan layak atau tidak untuk dimasukkan ke dalam himpunan solusi.
6. Fungsi obyektif

   Fungsi obyektif digunakan untuk menjelaskan apakah untuk memaksimumkan atau meminimumkan.

## 2.2 *Manhattan Distance*

***Manhattan distance***, juga dikenal sebagai *taxicab* atau *cityblock*, adalah suatu metode untuk menghitung jarak antara dua vektor yang bernilai riil. Jarak ini diperkirakan lebih berguna untuk vektor yang menggambarkan objek-objek pada *uniform grid* (kisi yang seragam), seperti papan catur atau blok kota dalam kota.
Nama *taxicab* untuk pengukuran ini merujuk pada intuisi tentang apa yang dihitung oleh pengukuran tersebut yaitu mencari jalur terpendek yang akan ditempuh oleh *taxicab* antara blok kota (koordinat pada *grid*).

Dalam beberapa kasus, terutama ketika berurusan dengan vektor dalam ruang fitur bilangan bulat, mungkin lebih masuk akal untuk menghitung jarak Manhattan daripada jarak Euclidean. Jarak Manhattan dihitung sebagai jumlah dari selisih absolut antara dua vektor.

ManhattanDistance = jumlah dari i hingga N jumlah |v1[i] – v2[i]|

Jarak Manhattan berkaitan dengan norma vektor L1 dan metrik kesalahan absolut total dan kesalahan absolut rata-rata. Ini karena jarak Manhattan mengukur jumlah perpindahan yang diperlukan untuk mencapai satu titik ke titik lainnya secara horizontal dan vertikal pada *grid*.

Dalam contoh penghitungan jarak Manhattan antara dua vektor bilangan bulat, perhitungannya dapat diperlihatkan dengan menghitung jumlah perbedaan absolut antara masing-masing elemen vektor.

Penggunaan jarak Manhattan sering kali berguna dalam masalah optimasi dan pengelompokan data di mana ruang fitur memiliki struktur yang beraturan, seperti peta kota atau grid. Dalam konteks ini, jarak Manhattan memberikan perkiraan yang baik tentang jarak nyata antara dua titik di *grid*, karena hanya menghitung perpindahan horizontal dan vertikal.

Dibandingkan dengan jarak Euclidean, jarak Manhattan lebih tepat untuk mengukur jarak antara dua titik dalam *grid* yang beraturan karena jarak Euclidean akan mengukur jarak garis lurus antara dua titik, sementara jarak Manhattan akan mengukur jumlah langkah horizontal dan vertikal yang diperlukan untuk mencapai satu titik dari yang lain.

Kesimpulannya, jarak Manhattan adalah metode yang berguna untuk mengukur jarak antara dua titik dalam ruang fitur yang berstruktur dengan baik, seperti papan catur atau *grid,* karena memberikan perkiraan yang baik tentang jumlah langkah yang diperlukan untuk mencapai satu titik dari yang lain dalam grid tersebut.

## 2.3 Cara Kerja Program

Program permainan Diamonds terdiri dari dua komponen utama. Pertama adalah game engine, yang meliputi backend permainan dan frontend permainan. Backend permainan ini berisi logika permainan secara keseluruhan dan menyediakan API untuk berkomunikasi dengan frontend dan program bot. Sementara itu, frontend permainan bertugas memvisualisasikan permainan. Kedua adalah bot starter pack, yang mencakup program untuk memanggil API dari backend, program logika bot yang akan diimplementasikan dengan algoritma greedy, serta program utama dan utilitas lainnya.

Dalam permainan Diamonds, bot berperan penting dalam mengumpulkan diamond. Oleh karena permainan ini merupakan permainan berbasis *web*, setiap aksi yang dilakukan, mulai dari mendaftarkan bot sampai menjalankannya akan memerlukan HTTP *request* terhadap API *endpoint* tertentu yang disediakan oleh *backend.* Program

bot dalam permainan Diamonds melakukan serangkaian langkah untuk berinteraksi dengan backend dan menjalankan aksinya dalam permainan. Pertama, bot akan memeriksa apakah sudah terdaftar atau belum dengan mengirimkan POST request ke endpoint /api/bots/recover, yang berisi email dan password bot. Jika bot sudah terdaftar, backend akan memberikan response code 200 dengan menyertakan id bot tersebut; jika tidak, akan diberikan response code 404. Jika bot belum terdaftar, bot akan mengirimkan POST request ke endpoint /api/bots dengan informasi email, nama, password, dan tim. Jika registrasi berhasil, backend akan memberikan response code 200 dengan id bot yang didaftarkan. Setelah mendapatkan id bot, bot dapat bergabung ke board dengan mengirimkan POST request ke endpoint /api/bots/{id}/join, yang berisi id board yang diinginkan. Jika berhasil bergabung, backend akan memberikan response code 200 dengan informasi dari board. Selanjutnya, bot akan secara berkala mengkalkulasikan langkah selanjutnya berdasarkan kondisi board dan mengirimkan POST request ke endpoint /api/bots/{id}/move dengan arah yang ditentukan. Jika langkah berhasil dieksekusi, backend akan memberikan response code 200 dengan kondisi board setelah langkah tersebut. Proses ini akan berulang terus menerus hingga waktu bot habis. Jika waktu bot habis, bot akan secara otomatis dikeluarkan dari board. Selain itu, program frontend juga secara periodik akan mengirimkan GET request ke endpoint /api/boards/{id} untuk mendapatkan kondisi board terbaru, sehingga tampilan board pada frontend akan selalu ter-update.

Implementasi algoritma greedy ke dalam bot dilakukan dengan langkah-langkah berikut. Pertama, kami membuat file baru pada direktori /game/logic dengan nama manhattan.py. Kedua, kami membuat sebuah kelas baru bernama Manhattan yang meng-inherit kelas BaseLogic, kemudian kami mengimplementasikan *constructor* dan *method* yang kami beri nama next_move pada kelas tersebut. Setelah itu, kami mengimport kelas yang telah dibuat ke dalam file main.py dan mendaftarkannya pada dictionary **CONTROLLERS**.

Langkah terakhir adalah menjalankan program dengan menggunakan script atau command yang disesuaikan dengan bot yang telah dibuat. Untuk menjalankan bot, kami menggunakan command-line interface. Untuk menjalankan satu bot, kami menjalankan perintah yang sesuai dengan logic bot yang ingin dijalankan, sementara untuk menjalankan beberapa bot sekaligus, kami menggunakan script yang disediakan. Proses menjalankan bot ini penting untuk memastikan bahwa bot dapat beroperasi sesuai yang diinginkan dalam permainan Diamonds.

Untuk menjalankan satu bot (pada contoh ini (bukan implementasi sesungguhnya), kami menjalankan satu bot dengan logic yang terdapat pada file game/logic/random.py)

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

Untuk menjalankan beberapa bot sekaligus (sebagai contoh (bukan implementasi sesungguhnya), kami menjalankan 2 bot dengan logic berbeda, yaitu game/logic/bot.py dan qika.py)

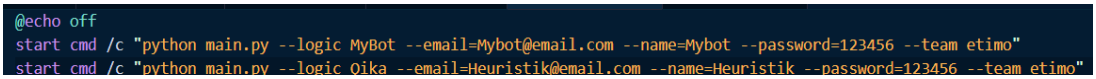
- Untuk windows

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

- Untuk Linux / (possibly) macOS

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

Kami juga dapat menyesuaikan *script* yang ada pada run-bots.bat atau run-bots.sh dari segi logic yang digunakan, email, nama, dan password jika ingin menjalankan beberapa bot seperti gambar berikut.



# BAB 3 APLIKASI STRATEGI GREEDY

## 3.1 Mapping Persoalan Diamond

Dengan menggunakan elemen-elemen yang terdapat dalam algoritma *greedy*, maka dapat dikatakan bahwa Algoritma *greedy* melibatkan pencarian terhadap sebuah himpunan bagian, S, dari himpunan kandidat, C; yang dalam hal ini, S harus memenuhi salah satu atau beberapa kriteria yang telah ditentukan, yaitu S menyatakan sebuah solusi dan S dioptimisasi dengan fungsi obyektif.

Setelah meninjau persoalan diamond pada permainan ini, kami mencoba untuk melakukan mapping terhadap persoalan tersebut menjadi elemen-elemen algoritma greedy agar mempermudah implementasi algoritma *greedy* pada program bot permainan ini. Mapping yang dihasilkan dari persoalan tersebut dapat dilihat pada penjelasan di bawah ini.

1. Himpunan Kandidat

   Himpunan kandidat dalam kasus ini adalah semua jalur yang memungkinkan untuk mencapai diamond merah (2 poin) dan diamond biru (1 poin).
2. Himpunan Solusi

   Himpunan solusi adalah jalur-jalur dari himpunan kandidat yang memiliki total poin diamond terbesar dengan jarak yang paling pendek.
3. Fungsi Solusi

   Fungsi solusi mengecek apakah jalur yang dipilih sudah menghasilkan total poin diamond yang maksimal (5 poin dalam satu perjalanan).
4. Fungsi Seleksi

   Fungsi seleksi memilih jalur yang total poin diamondnya paling besar dengan jarak yang paling pendek, terhindar dari *tackle*-an bot lain, dan atau berpeluang men-*tackle* bot lain.
5. Fungsi Kelayakan

   Semua langkah dalam permainan dianggap layak
6. Fungsi Obyektif

   Fungsi obyektif memaksimalkan total poin diamond yang diperoleh dan meminimumkan jarak yang ditempuh.

Dengan memetakan elemen-elemen algoritma greedy ke dalam permasalahan diamond, kami dapat menerapkan algoritma greedy untuk menentukan jalur terbaik dalam permainan diamond. Algoritma greedy yang kami implementasikan akan memilih jalur yang memaksimalkan total poin diamond dan meminimalkan jarak yang ditempuh.

## 3.2 Eksplorasi Alternatif Solusi Greedy

Dalam menganalisis penyelesaian persoalan diamond ini. Kami telah mengidentifikasi beberapa alternatif solusi greedy yang dapat dipertimbangkan untuk meningkatkan performa bot dalam permainan diamond ini. Berikut beberapa alternatif solusi yang kami dapatkan :

1. Strategi *greedy* : *greedy by value*

   Strategi *greedy* ini memilih jalur dengan total poin diamond terbesar, tanpa memperhitungkan jarak yang ditempuh. Kelebihannya, bot dapat mencapai total poin diamond yang tinggi dalam waktu singkat dan mudah diimplementasikan. Kekurangannya, bot mungkin menempuh jarak yang jauh, menghabiskan waktu dan energi, serta berisiko ditabrak (*di-tackle*) oleh bot lain.
2. Strategi *greedy* : *greedy by distance*

   Strategi *greedy* ini memilih jalur dengan jarak terpendek, tanpa memperhitungkan total poin diamond yang diperoleh. Kelebihannya, bot menghemat waktu dan enerStrukgi dengan menempuh jarak pendek dan memiliki risiko tertabrak bot lain yang lebih rendah. Kekurangannya, bot mungkin mendapatkan total poin diamond yang rendah dan melewatkan diamond bernilai tinggi.
3. Strategi *greedy* : *greedy by value combination distance*

   Strategi *greedy* ini menggabungkan kedua pendekatan sebelumnya dengan memilih jalur yang memiliki kombinasi terbaik antara total poin diamond dan jarak yang ditempuh, tetapi tidak mempertimbangkan pergerakan bot lain. Kelebihannya, bot mencapai keseimbangan antara total poin diamond dan jarak yang ditempuh, serta lebih adaptif terhadap situasi berbeda. Kekurangannya, implementasi algoritma ini lebih kompleks dan penentuan bobot untuk total poin diamond, jarak yang ditempuh dapat menjadi rumit, dan risiko ditabrak oleh bot lain lebih tinggi.
4. Strategi *greedy* : *greedy by value combination distance and time*

   Strategi ini merupakan pengembangan dari strategi sebelumnya dengan menambahkan faktor waktu. Strategi ini memilih jalur yang memiliki kombinasi terbaik antara total poin diamond, jarak yang ditempuh, dan waktu yang tersedia. Strategi ini memungkinkan bot mencapai efisiensi waktu dan meminimalisir hilangnya poin di detik-detik akhir permainan. Namun, strategi ini juga memiliki kekurangan. Pertama, implementasi algoritmanya lebih kompleks. Kedua, penentuan bobot untuk total poin diamond, jarak yang ditempuh, dan waktu yang tersedia dapat menjadi rumit. Ketiga, risiko bot ditabrak oleh bot lain tetap ada.
5. Strategi *greedy* : *greedy by tackle*

   Strategi greedy ini hanya berfokus untuk mengejar bot lain untuk mendapatkan poin (tackle) memiliki keuntungan dan kekurangan. Keuntungannya, bot mendapatkan poin tambahan, menghambat bot lain, dan memberikan keuntungan strategis. Kekurangannya, bot mengabaikan diamond, berisiko tertabrak, dan kurang efisien apabila terdapat banyak bot dalam satu *board*.

6. Strategi *greedy* : *greedy by distance, time, and tackle*

   Strategi *greedy* ini merupakan kombinasi dari dua strategi greedy sebelumnya, yaitu *greedy by value combination distance and time* dan *greedy by tackle*, dengan beberapa pembaharuan. Strategi ini bertujuan untuk mencapai solusi optimal dengan memaksimalkan poin, efisiensi waktu, dan meminimalisir risiko.

   Strategi ini memilih jalur yang memiliki kombinasi terbaik antara total poin diamond, jarak yang ditempuh, waktu yang tersedia, dan faktor risiko. Bot akan menghindari situasi berisiko tinggi, seperti tertabrak bot lain atau terjebak di sudut. Strategi ini juga mempertimbangkan jarak bot kami dengan bot lain untuk memaksimalkan peluang tackle dan meminimalisir risiko tertabrak.

   Bobot untuk total poin, jarak, waktu, dan risiko dapat disesuaikan berdasarkan situasi permainan dan strategi tim. Algoritma pembelajaran mesin dapat digunakan untuk mempelajari pola permainan dan mengoptimalkan strategi secara real-time.

   Strategi ini menawarkan efisiensi tinggi, keamanan tinggi, dan fleksibilitas untuk beradaptasi dengan berbagai situasi. Kekurangannya adalah implementasi yang cukup kompleks dan membutuhkan pengetahuan lebih tentang permainan ini.

7. Strategi *greedy :greedy by object, distance, time, and tackle*

   Strategi *greedy* ini merupakan pengembangan dari strategi *greedy* sebelumnya dengan mempertimbangkan beberapa faktor tambahan untuk pergerakan bot. Faktor-faktor tersebut meliputi keberadaan *object* seperti *red button* atau *teleporter* di seluruh board, jarak tempuh, waktu yang dibutuhkan untuk menyelesaikan rute, dan strategi tackle yang dimiliki bot.

   Strategi ini membantu bot untuk memilih rute yang tidak hanya efisien dalam hal jarak dan waktu, tetapi juga memungkinkan bot untuk mengumpulkan object yang bermanfaat. Bot akan memilih rute yang memiliki jarak total paling pendek dan dapat diselesaikan dalam waktu tercepat. Selain itu, bot juga akan mempertimbangkan tackle yang dimilikinya saat memilih rute, sehingga bot dapat menyelesaikan tugas dengan lebih aman.

## 3.3 Analisis Efisiensi dan Efektivitas Alternatif Solusi Greedy

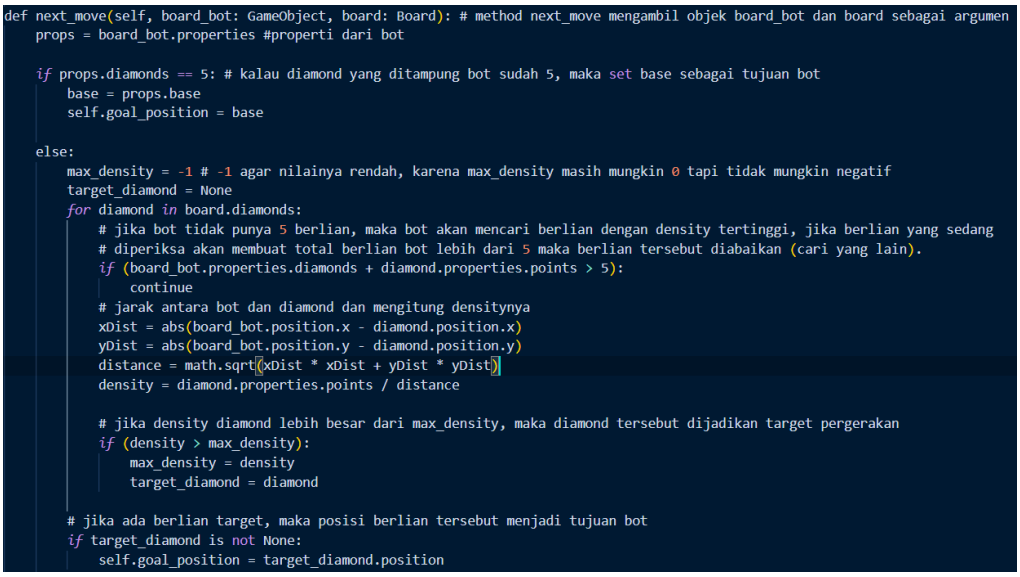
Dari beberapa alternatif solusi *greedy* yang telah disampaikan, kami menganalisis terlebih dahulu efisiensi dan efektivitas dari masing-masing alternatif solusi *greedy*. Berikut adalah poin-poin utama dari efektivitas dan efisiensi masing-masing alternatif solusi *greedy.*

1. Greedy by Value:
    a. Efisiensi: Tinggi dalam mencapai total poin diamond dengan cepat.
    b. Efektivitas: Mungkin kurang efektif dalam menghindari perjalanan yang jauh dan risiko tabrakan dengan bot lain.
2. Greedy by Distance:
    a. Efisiensi: Tinggi dalam menemukan jalur terpendek untuk mengejar diamond.
    b. Efektivitas: Mungkin kurang efektif dalam memaksimalkan total poin diamond yang diperoleh.
3. Greedy by Value Combination Distance:
    a. Efisiensi: Moderat, karena mencoba mencapai keseimbangan antara nilai diamond dan jarak.
    b. Efektivitas: Lebih adaptif terhadap situasi daripada kedua pendekatan sebelumnya, tetapi kompleksitas implementasi lebih tinggi.
4. Greedy by Value Combination Distance and Time:
    a. Efisiensi: Tinggi dalam memperhitungkan waktu akhir permainan.
    b. Efektivitas: Memiliki kemampuan untuk memaksimalkan poin sambil memperhitungkan jarak dan waktu, namun, kompleksitas implementasi lebih tinggi.
5. Greedy by Tackle:
    a. Efisiensi: Tinggi dalam mengejar bot lain untuk mendapatkan poin tambahan.
    b. Efektivitas: Kurang efektif dalam memperoleh poin dari diamond dan rentan terhadap tabrakan.
6. Greedy by Distance, Time, and Tackle:
    a. Efisiensi: Tinggi dalam memperhitungkan berbagai faktor termasuk jarak, waktu, dan risiko tabrakan.
    b. Efektivitas: Memiliki kemungkinan efektivitas yang tinggi dengan memperhitungkan banyak faktor, namun, kompleksitas implementasi lebih tinggi.
7. Greedy by Object, Time, and Tackle:
    a. Efisiensi: Tinggi dalam memperhitungkan berbagai faktor termasuk objek seperti *red button,* jarak, waktu, dan risiko tabrakan.

b. Efektivitas: Memiliki kemungkinan efektivitas yang tinggi dengan memperhitungkan banyak faktor, namun, kompleksitas implementasi lebih tinggi.

Pada pengerjaan tugas besar yang kami lakukan, kami tidak mengimplementasikan keseluruhan alternatif solusi tersebut, tetapi kami menerapkan sistem iterasi untuk perbaikan yang kami lakukan pada setiap algoritma yang telah kami buat.

Pada iterasi pertama, kami telah menentukan bahwa strategi greedy yang hanya menjadikan jarak atau hanya menjadikan *value* sebagai faktor perhitungan untuk penentuan rute bot sangatlah tidak efektif untuk mendapatkan poin terbesar sehingga kami mulai membuat algoritma *greedy* dengan strategi mengkombinasikan faktor *value* dengan jarak menggunakan **teorema euclidean**. Algoritma pada tahap ini menghasilkan pergerakan bot yang sangat sederhana karena hanya berfokus pada pencarian rute terpendek untuk mendapatkan diamond dengan *value* terbesar. Implementasi algoritma nya pun paling sederhana dibandingkan iterasi lainnya. Berikut adalah implementasi *method* next_move dari iterasi pertama kami.



```
def next_move(self, board_bot: GameObject, board: Board): # method next_move mengambil objek board_bot dan board sebagai argumen
    props = board_bot.properties #properti dari bot

    if props.diamonds == 5: # kalau diamond yang ditampung bot sudah 5, maka set base sebagai tujuan bot
        base = props.base
        self.goal_position = base

    else:
        max_density = -1 # -1 agar nilainya rendah, karena max_density masih mungkin 0 tapi tidak mungkin negatif
        target_diamond = None
        for diamond in board.diamonds:
            # jika bot tidak punya 5 berlian, maka bot akan mencari berlian dengan density tertinggi, jika berlian yang sedang
            # diperiksa akan membuat total berlian bot lebih dari 5 maka berlian tersebut diabaikan (cari yang lain).
            if (board_bot.properties.diamonds + diamond.properties.points > 5):
                continue
            # jarak antara bot dan diamond dan mengitung densitynya
            xDist = abs(board_bot.position.x - diamond.position.x)
            yDist = abs(board_bot.position.y - diamond.position.y)
            distance = math.sqrt(xDist * xDist + yDist * yDist)
            density = diamond.properties.points / distance

            # jika density diamond lebih besar dari max_density, maka diamond tersebut dijadikan target pergerakan
            if (density > max_density):
                max_density = density
                target_diamond = diamond

        # jika ada berlian target, maka posisi berlian tersebut menjadi tujuan bot
        if target_diamond is not None:
            self.goal_position = target_diamond.position
```

```
# jika ada tujuan, maka bot akan menghitung arah bergerak menuju tujuan tersebut
if self.goal_position:
    current_position = board_bot.position
    cur_x = current_position.x
    cur_y = current_position.y
    delta_x, delta_y = get_direction(
        cur_x,
        cur_y,
        self.goal_position.x,
        self.goal_position.y,
    )

    # jika bot tidak bergerak pada langkah sebelumnya, maka bot akan berbelok untuk keluar dari posisi yang menjebak (contohnya batas kotak)
    if (cur_x, cur_y) == self.previous_position:
        if delta_x != 0:
            delta_y = delta_x * self.turn_direction
            delta_x = 0
        elif delta_y != 0:
            delta_x = delta_y * self.turn_direction
            delta_y = 0
        self.turn_direction = -self.turn_direction

    # menyimpan posisi saat ini sebagai posisi sebelumnya dan mengembalikan arah bergerak
    self.previous_position = (cur_x, cur_y)
    return delta_x, delta_y

# jika tidak ada tujuan, maka bot tidak bergerak
return 0, 0
```

Dari implementasi tersebut, dapat dianalisis bahwa bot akan menetapkan tujuan bergeraknya berdasarkan kondisi saat ini, mencari berlian dengan *density* tertinggi jika belum mengumpulkan 5 berlian, dan menghitung arah geraknya menuju tujuan tersebut. Bot juga akan melakukan penyesuaian gerakan untuk menghindari terjebak dan menyimpan posisi sebelumnya untuk memeriksa apakah bot tidak bergerak pada langkah selanjutnya, memungkinkan bot untuk melakukan gerakan yang efisien dan cerdas dalam mencapai tujuan yang ditetapkan. Selain itu, penting untuk mempertimbangkan kemungkinan adanya pertandingan dengan bot lai. Kelompok kami berpikir bahwa dengan memperhitungkan kemungkinan tersebut, kami dapat mengembangkan strategi tambahan untuk mengantisipasi interaksi dengan bot lainnya.

Pada iterasi kedua, kami memperluas algoritma dengan menambahkan logika untuk mendeteksi keberadaan bot lain di sekitar dan mengadaptasi pergerakan bot sesuai dengan informasi tersebut. Implementasi dari algoritma pada tahap iterasi ini sedikit lebih kompleks dan adaptif untuk menghadapi skenario permainan dengan bot lain yang mungkin terjadi, meningkatkan kemampuan bot untuk bersaing dengan bot lainnya dengan lebih efektif. Berikut adalah implementasi *method* next_move dari iterasi tahap kedua.

```python
def next_move(self, board_bot: GameObject, board: Board):
    props = board_bot.properties

    teleporter = [obj for obj in board.game_objects if obj.type == "TeleportGameObject"]
    teleporter
    if(board.game_objects == 'portal') :
        var = board.game_objects

    # if board.game_objects[i].type == "teleporter game object"

    if props.diamonds == 5:
        base = props.base
        self.goal_position = base

    else:
        max_score = -1
        target = None
        for diamond in board.diamonds:
            if (board_bot.properties.diamonds + diamond.properties.points > 5):
                continue
            xDist = abs(board_bot.position.x - diamond.position.x)
            yDist = abs(board_bot.position.y - diamond.position.y)
            distance = math.sqrt(xDist * xDist + yDist * yDist)
            score = diamond.properties.points / distance

            if (score > max_score):
                max_score = score
                target = diamond
```

```python
            for other_bot in board.bots:
                if other_bot.id == board_bot.id:
                    continue

                xDist = abs(board_bot.position.x - other_bot.position.x)
                yDist = abs(board_bot.position.y - other_bot.position.y)
                distance = math.sqrt(xDist * xDist + yDist * yDist)
                score = other_bot.properties.diamonds / distance

                if other_bot.properties.diamonds > board_bot.properties.diamonds:
                    score = -score

                if (score > max_score):
                    max_score = score
                    target = other_bot

            if target is not None:
                self.goal_position = target.position

        if self.goal_position:
            current_position = board_bot.position
            cur_x = current_position.x
            cur_y = current_position.y
            delta_x, delta_y = get_direction(
                cur_x,
                cur_y,
                self.goal_position.x,
                self.goal_position.y,
            )

            if (cur_x, cur_y) == self.previous_position:
                if delta_x != 0:
                    delta_y = delta_x * self.turn_direction
                    delta_x = 0
                elif delta_y != 0:
                    delta_x = delta_y * self.turn_direction
                    delta_y = 0
                self.turn_direction = -self.turn_direction

            self.previous_position = (cur_x, cur_y)
            return delta_x, delta_y

        return 0, 0
```

Dari implementasi tersebut bot akan menambahkan logika tambahan untuk mendeteksi keberadaan bot lain di sekitar dan mengadaptasi pergerakan bot sesuai dengan informasi tersebut. Kami mencoba meneliti pergerakan dari bot yang telah kami

buat. Ternyata, saat waktu permainan mendekati akhir dan bot kami sudah menyimpan sebanyak 5 diamond, skor tersebut tidak tersimpan karena tidak cukup waktu untuk kembali ke base. Dari kasus tersebut, kami memutuskan untuk menambahkan faktor perhitungan waktu tersisa agar skor yang kami dapatkan tidak terbuang sia-sia.

Pada tahap iterasi ketiga, kami menambahkan faktor waktu yang telah dijelaskan sebelumnya. Berikut adalah implementasi tambahan yang kami tambahkan pada algoritma dari tahap iterasi kedua.

```
teleporter = [obj for obj in board.game_objects if obj.type == "TeleportGameObject"]

if props.diamonds == 5 or props.milliseconds_left == 5000 or props.milliseconds_left == 3000:
    base = props.base
    self.goal_position = base
```

Dari implementasi tersebut, dapat dilihat bahwa saat waktu permainan tersisa 5 detik atau 3 detik bot akan kembali ke base untuk menyimpan diamond agar skornya tidak terbuang. Pada tahap ini, kami memulai pertandingan sederhana dengan teman-teman dari kelompok lain dan berhasil menemukan kekurangan dari algoritma kami, yaitu bot akan lebih fokus mengejar bot lain saat bertemu dengan bot lain.

Dari kekurangan tersebut muncullah iterasi keempat pada algoritma kami. Kami menambahkan syarat kondisi untuk strategi tackle. Bot hanya dapat melakukan strategi tackle saat jarak antara bot kami dengan bot lain berkisar 1 dalam perhitungan jarak euclidean. Berikut adalah implementasi tambahan yang kami tambahkan dari algoritma iterasi ketiga yang kami buat (perbaikan dilakukan pada strategi tackle).

```
for other_bot in board.bots:
    if other_bot.id == board_bot.id:
        continue

    # Menghitung jarak antara bot yang sedang diproses dengan bot lainnya
    xDist = abs(board_bot.position.x - other_bot.position.x)
    yDist = abs(board_bot.position.y - other_bot.position.y)
    distance = math.sqrt(xDist * xDist + yDist * yDist)
    # Mengecek apakah bot lain berada dalam jangkauan tertentu (1 <= distance <= 2)
    if distance == 1:
        target = other_bot

# Set goal_position ke posisi bot lain yang menjadi target
if target is not None:
    self.goal_position = target.position
```

Setelah kami mencoba menggunakan implementasi yang sudah diiterasi sebanyak empat kali ini. Kami mulai menganalisis kembali pergerakan bot dengan berkompetisi dengan beberapa kelompok lain. Bot kami bergerak aman dan efektif saat kondisi *board* banyak diamond dan pintu teleporter dengan ujung teleporter terletak tidak pada rute yang bot kami ingin lewati. Namun, kami menemukan kejanggalan saat teleporter baik pintu maupun ujung berada di rute yang bot kami ingin lewati. Bot kami akan terkena

*looping* yang tidak terbatas selama teleporter letaknya masih sama. Hal ini sangat membuat bot kami tidak efisien dan membuat bot kami tidak bisa mendapatkan skor yang optimal. Ini membuat bot kami tidak konsisten perjalanannya terhadap board yang dimainkan.

Dari permasalahan tersebut, kami iterasi kembali program kami dan memutuskan untuk menambahkan faktor *object* yaitu teleporter untuk masuk ke dalam perhitungan dan untuk dihindari agar tidak membuat bot kami menjadi *infinite looping* serta *red button*. Tetapi, saat dicoba memasukkan faktor teleporter ke dalam perhitungan, bot kami lebih banyak mengalami *stuck* jika jarak antara base dengan teleporter cukup dekat sehingga pada tahap iterasi kelima ini, kami hanya menambahkan faktor *red button* pada perhitungan. *Red button* menambahkan keuntungan pada program kami karena memiliki peluang untuk membuat bot lain gagal menuju tujuan diamond sebelum *red button* diduduki oleh bot kami. Berikut ini adalah implementasi tambahan yaitu strategi menghadapi *red button.*

```
for btn in button:
    xDist = abs(board_bot.position.x - btn.position.x)
    yDist = abs(board_bot.position.y - btn.position.y)
    distance = math.sqrt(xDist * xDist + yDist * yDist)

    if distance <= 1 and not self.button_pressed:
        target = btn
        self.button_pressed = True

if target is not None:
    self.goal_position = target.position
```

Algoritma pada tahap iterasi kelima ini sebetulnya lumayan optimal baik saat dikompetisikan dengan beberapa bot dari kelompok lain maupun dari pengambilan diamond. Namun, setelah kami lakukan pertandingan berulang kali dengan beberapa kelompok lain, kami merasa bahwa algoritma ini kurang konsisten untuk dapat memenangkan setiap permainannya. Akhirnya, kami mencoba untuk melakukan iterasi yang terakhir.

Pada iteraksi terakhir atau iteraksi keenam ini, kami menemukan metode pencarian rute yang ternyata lebih efektif dan efisien daripada menggunakan metode jarak euclidean. Metode yang kami temukan adalah metode yang menerapan jarak Manhattan. Dengan jarak manhattan algoritma kami menemukan rute terpendek dengan lebih akurat dan sesuai dengan penggunaanya dalam permainan kisi (*grid*) ini. Selain itu, kami juga menemukan cara untuk menghadapi objek teleporter dengan cara menambahkan syarat atau kondisi agar bot hanya dapat melewati teleporter dengan ujungnya semaksimalnya satu kali. Berikut adalah implementasi dari tahap terakhir ini.

```python
class Manhattan(object):
    def __init__(self):
        self.goal_position = None
        self.previous_position = (None, None)
        self.turn_direction = 1
        self.teleporter_visits = {}
        self.button_pressed = False

    def next_move(self, board_bot: GameObject, board: Board):
        # Dapatkan properti dari board
        props = board_bot.properties

        # Kumpulkan teleporter dan button yang berada di board
        teleporter = [obj for obj in board.game_objects if obj.type == "TeleportGameObject"]
        button = [obj for obj in board.game_objects if obj.type == "DiamondButtonGameObject"]

        # Hitung jarak antara bot dengan base
        base = props.base
        xDist_base = abs(board_bot.position.x - base.x)
        yDist_base = abs(board_bot.position.y - base.y)
        distance_base = xDist_base + yDist_base

        # Apabila bot memiliki 5 diamond, atau bot memiliki 3 sampai 4 diamond tetapi dekat ke base, atau waktu tersisa kurang dari 5 detik, maka jadikan base sebagai tujuan
        if (props.diamonds == 5 or props.milliseconds_left == 5000 or props.milliseconds_left == 3000) or (props.diamonds in [3, 4] and distance_base <= 3):
            self.goal_position = base

        else:
            max_score = -1
            target = None

            # Cari letak diamond
            for diamond in board.diamonds:
                if (board_bot.properties.diamonds + diamond.properties.points > 5):
                    continue
                # Hitung density diamond terhadap bot dengan pembagian jarak berdasarkan jarak manhattan dengan bobot diamond
                xDist = abs(board_bot.position.x - diamond.position.x)
                yDist = abs(board_bot.position.y - diamond.position.y)
                distance = xDist + yDist
                score = diamond.properties.points / distance

                # Diamond dengan density terbesar dijadikan target oleh bot untuk didatangi
                if (score > max_score):
                    max_score = score
                    target = diamond
```

```python
# Interaksi dengan bot lain
for other_bot in board.bots:
    if other_bot.id == board_bot.id:
        continue

    # Menghitung jarak bot dengan bot lain menggunakan jarak manhattan
    xDist = abs(board_bot.position.x - other_bot.position.x)
    yDist = abs(board_bot.position.y - other_bot.position.y)
    distance = xDist + yDist

    # Apabila jarak bot lain secara manhattan dibawah satu dan can_tackle bernilai true, maka jadikan bot lain sebagai target
    if distance <= 1 and props.can_tackle:
        target = other_bot

# Interaksi dengan button
for btn in button:
    # Hitung jarak button dengan bot menggunakan jarak manhattan
    xDist = abs(board_bot.position.x - btn.position.x)
    yDist = abs(board_bot.position.y - btn.position.y)
    distance = xDist + yDist

    # Jika jarak button kurang dari 1 dan button belum terpencet, maka jadikan button sebagai target bot
    if distance <= 1 and not self.button_pressed:
        target = btn
        self.button_pressed = True

if target is not None:
    self.goal_position = target.position
```

```
# Sesuaikan arah pergerakan bot, khususnya untuk gerak berbelok
if self.goal_position:
    current_position = board_bot.position
    cur_x = current_position.x
    cur_y = current_position.y
    delta_x, delta_y = get_direction(
        cur_x,
        cur_y,
        self.goal_position.x,
        self.goal_position.y,
    )

    # Interaksi dengan teleport
    for tp in teleporter:
        if tp.position.x == cur_x + delta_x and tp.position.y == cur_y + delta_y:
            # Bila sudah sekali masuk ke teleporter koordinat tertentu, maka tidak bisa masuk lagi
            if self.teleporter_visits.get(tp.id, 0) >= 2:
                delta_x, delta_y = -delta_x, -delta_y
                break
            else:
                self.teleporter_visits[tp.id] = self.teleporter_visits.get(tp.id, 0) + 1

    if (cur_x, cur_y) == self.previous_position:
        if delta_x != 0:
            delta_y = delta_x * self.turn_direction
            delta_x = 0
        elif delta_y != 0:
            delta_x = delta_y * self.turn_direction
            delta_y = 0
        self.turn_direction = -self.turn_direction

    self.previous_position = (cur_x, cur_y)
    # Kembalikan nilai delta x dan delta y untuk pergerakan bot
    return delta_x, delta_y

return 0, 0
```

Dari implementasi tersebut, kami melakukan kembali uji coba dengan iterasi-iterasi sebelumnya dan dengan beberapa kelompok lain. Hasilnya, algoritma pada tahap iterasi ini memberikan hasil yang optimal, konsisten pada setiap pertandingannya, dan kecil menungkinan untuk berada di skor yang paling bawah.

## 3.4 Strategi Greedy yang Dipilih

Dari penjelasan solusi sebelumnya, terlihat bahwa setiap solusi memiliki keunggulan dan kelemahan masing-masing. Pada implementasi program bot kami, kami memutuskan untuk menggunakan pendekatan algoritma greedy berdasarkan *object, distance*, *time*, dan *tackle* karena algoritma ini memiliki beberapa kelebihan yang sesuai dengan kebutuhan dan tujuan dalam permainan:

1. Efisiensi Waktu dan Energi:
   Algoritma ini memperhitungkan jarak yang harus ditempuh oleh bot untuk mencapai tujuan tertentu (berlian atau posisi strategis lainnya). Dengan memilih jalur terpendek, bot dapat menghemat waktu dan energi yang diperlukan untuk mencapai tujuan tersebut.

2. Optimisasi terhadap Waktu yang Tersisa:
   Algoritma ini juga mempertimbangkan waktu yang tersisa dalam permainan. Dengan memperhitungkan waktu yang tersedia, bot dapat mengoptimalkan langkah-langkahnya untuk mendapatkan sebanyak mungkin poin dalam batas waktu yang terbatas.

3. Minimalkan Risiko Tertabrak (*Tackle*):
   Selain itu, algoritma ini juga mempertimbangkan risiko tertabrak oleh bot lain (tackle). Dengan memperhitungkan jarak dan posisi bot lain di sekitarnya, bot dapat mencoba menghindari jalur yang berisiko tinggi untuk ditabrak oleh bot lain.

4. Minimalkan Risiko Kejadian Tak Terduga akibat Objek Teleporter dan Red Button
   Algoritma greedy yang kami terapkan juga meminimalkan risiko kejadian tak terduga yang mungkin timbul akibat objek teleporter dan *red button* dalam permainan. Bot kami mempertimbangkan kemungkinan efek dari objek-objek tersebut dan mencoba untuk menghindari situasi yang dapat mengakibatkan dampak negatif, seperti teleporter yang dapat membawa bot ke lokasi yang tidak diinginkan atau red button yang dapat mengubah konfigurasi *board* secara drastis. Dengan memperhitungkan risiko dari kedua objek tersebut, bot kami dapat mengambil keputusan yang lebih bijaksana dalam menjalankan strateginya dan mengoptimalkan kemungkinan kemenangan dalam permainan.

Dengan memperhitungkan semua faktor tersebut, pendekatan algoritma "greedy by object, distance, time, dan tackle" mampu memberikan strategi yang efisien, adaptif, dan optimal untuk bot dalam mencapai tujuan permainan.

# BAB 4 IMPLEMENTASI DAN PENGUJIAN

## 4.1 Implementasi Algoritma Greedy pada Program Bot

1. Fungsi init

```
function __init__(self)

KAMUS LOKAL

    goal_position: tuple
    {Menyimpan posisi tujuan yang ingin dicapai oleh bot}
    previous_position: tuple
    {Menyimpan posisi sebelumnya dari bot}
    turn_direction: integer
    {Menyimpan arah putaran bot}

ALGORITMA

    goal_position ← None
    previous_position ←(None, None)
    turn_direction ← 1
```

2. Fungsi next_move

```
function next_move(self, board_bot: GameObject, board: Board)

KAMUS LOKAL

    props: Properties
    teleporter: array of GameObject
    max_score: float
    target: GameObject
    diamond: GameObject
    other_bot: GameObject
    xDist, yDist, distance: float
    cur_x, cur_y, delta_x, delta_y: integer

ALGORITMA

    props ← board_bot.properties
    teleporter ← [obj for obj in board.game_objects if obj.type
    == "TeleportGameObject"]
```

```
    { Kondisi jika sudah waktunya untuk balik ke base }
    if props.diamonds = 5 or props.milliseconds_left = 5000 or
    props.milliseconds_left = 3000 then
       base ← props.base
       self.goal_position ← base

    else { Kondisi jika belum waktunya untuk balik ke base }
       max_score ← -1
       target ← None

    { Mencari diamond mana yang paling optimal untuk diambil }
       for each diamond in board.diamonds do
           if board_bot.properties.diamonds +
    diamond.properties.points > 5 then
              continue

           xDist ← absolute(board_bot.position.x -
    diamond.position.x)
           yDist ← absolute(board_bot.position.y -
    diamond.position.y)
           distance ← square_root(xDist * xDist + yDist *
    yDist)
           score ← diamond.properties.points / distance

           if score > max_score then
              max_score ← score
              target ← diamond

    { Proses untuk melakukan tackle terhadap bot lain}
       for each other_bot in board.bots do
           if other_bot.id = board_bot.id then
              continue
           xDist ← absolute(board_bot.position.x -
    other_bot.position.x)
           yDist ← absolute(board_bot.position.y -
    other_bot.position.y)
           distance ← square_root(xDist * xDist + yDist *
    yDist)

           if 1 <= distance <= 2 then
              score ← other_bot.properties.diamonds

              if score > max_score then
```

```
                max_score ← score
                target ← other_bot

      { Menetapkan tujuan bot jika target telah dipilih }
       if target is not None then
              self.goal_position ← target.position

    { Menentukan langkah selanjutnya berdasarkan posisi tujuan
    bot }
    if self.goal_position is not None then
       current_position ← board_bot.position
       cur_x ← current_position.x
       cur_y ← current_position.y
       delta_x, delta_y ← get_direction(
            cur_x, cur_y,
            self.goal_position.x, self.goal_position.y
       )

       if (cur_x, cur_y) = self.previous_position then
          if delta_x ≠ 0 then
             delta_y ← delta_x * self.turn_direction
             delta_x ← 0
          elif delta_y ≠ 0 then
               delta_x ← delta_y * self.turn_direction
               delta_y ← 0
          self.turn_direction ← -self.turn_direction
       self.previous_position ← (cur_x, cur_y)
       return delta_x, delta_y

    return 0, 0
```

## 4.2 Penjelasan Struktur Data dalam Program Bot Diamonds

Program ini dibuat dengan menggunakan paradigma pemrograman berorientasi objek. Dalam paradigma ini, program dibagi menjadi unit-unit yang disebut objek, yang masing-masing mewakili entitas atau konsep dalam program. Setiap program memiliki atribut dan metode yang mendefinisikan sifat dan perilaku objek tersebut.

Dalam konteks program Bot Etimo Diamonds ini, terdapat beberapa kelas yang mendefinisikan objek-objek dalam program. Misalnya, ada kelas *Manhattan* yang mendefinisikan bot itu sendiri. Bot ini memiliki atribut seperti *goal_position*, *previous_position*, dan *turn_direction* yang digunakan untuk menyimpan informasi tentang posisi tujuan, posisi sebelumnya, dan arah putaran bot.

Selain itu, ada kelas *Bot*, *Position*, *Base*, *Properties*, *GameObject*, *Config*, *Feature*, dan *Board* yang didefinisikan dalam models.py. Kelas-kelas ini digunakan untuk menyimpan informasi tentang bot, posisi objek di papan, properti objek, konfigurasi fitur, dan papan permainan itu sendiri. Ada beberapa fungsi utilitas yang didefinisikan dalam util.py, seperti *clamp*, *get_direction*, dan *position_equals*. Fungsi-fungsi ini digunakan untuk melakukan operasi yang berkaitan dengan posisi dalam *game*.

Secara keseluruhan, struktur data ini memungkinkan bot untuk berinteraksi dengan lingkungan permainannya dan membuat keputusan tentang gerakan berikutnya berdasarkan kondisi saat ini di papan permainan. Struktur data ini merupakan bagian penting dari program Bot Diamonds dan memainkan peran kunci dalam fungsionalitas dan kinerja bot.

## 4.3 Analisis Desain Solusi Algoritma Greedy

Algoritma *greedy* merupakan algoritma yang mencari keputusan atau langkah yang paling optimal tanpa bisa kembali ke keadaan sebelumnya. Ada beberapa pendekatan algoritma *greedy* untuk kasus bot pada game ini, antara lain berdasarkan jarak, berdasarkan bobot diamond, serta gabungan antara jarak dan bobot, atau biasa disebut *density*. Pada bot ini, diterapkan pendekatan algoritma greedy berbasis density, untuk menghitung density, digunakan bantuan perhitungan jarak menggunakan perhitungan jarak *manhattan*, jarak *manhattan* dipilih karena merupakan metode perhitungan jarak paling optimal untuk medium dua dimensi berbentuk grid.

Pertama-tama program bot akan melakukan *looping* terhadap matriks papan permainan dan mendapatkan seluruh objek yang ada di arena khususnya *diamonds* dan *base*, setelah koordinat setiap objek didapat, bot akan menghitung jarak seluruh objek terhadap bot, apabila jarak *base* cukup dekat, atau jumlah diamond sama dengan lima, atau waktu tersisa tujuh detik, maka bot akan menjadikan *base* sebagai target untuk didatangi, apabila kondisi di atas tidak terpenuhi, maka bot akan mulai menjadikan *diamond* sebagai targetnya.

Mekanisme pengambilan keputusan dipengaruhi oleh atribut *score* yang merepresentasikan *density* dari bot terhadap objek, *density* dihitung dengan cara membagi jarak hasil perhitungan yang dihasilkan algoritma perhitungan *manhattan* dengan bobot masing-masing *diamond*, *diamond* dengan *density* terbesar terhadap bot akan dijadikan target oleh bot untuk didatangi.

Selain diamond, ada beberapa parameter yang dijadikan acuan untuk mengambil keputusan yang optimal, salah satunya adalah *teleporter*, untuk mencegah bot mengalami *stuck* dan melakukan *loop* di dua buah *teleporter*, maka telah ditambahkan algoritma agar bot hanya bisa menggunakan *teleporter* di suatu koordinat sekali, ketika bot akan menabrak teleporter untuk kedua kalinya di koordinat yang sama, bot akan menghindarinya dan mencari jalan lain untuk mencapai targetnya, bot baru akan bisa menggunakan kembali *teleporter* setelah *teleporter* berubah koordinat.

Untuk melakukan *tackle* terhadap bot lain, telah diterapkan algoritma agar bot hanya bisa melakukan *tackle* ketika jarak bot terhadap bot lain menurut jarak *manhattan* kurang dari 1 dan status atribut *can_tackle* bernilai *true*, jarak yang ditetapkan untuk melakukan *tackle* cukup kecil untuk mencegah bot kejar-kejaran dengan bot lain.

*Diamond button* juga dimanfaatkan sebaik mungkin untuk keuntungan bot, selain untuk melakukan *reset* jumlah *diamond*, bot juga memanfaatkan *diamond button* untuk mencegah bot lain mencapai tujuan bot lain saat *red button* belum diduduki, sehingga bot memiliki skor lebih tinggi dibanding bot lain. Bot akan menjadikan *diamond button* sebagai target dan menekannya ketika jarak *diamond button* dengan bot cukup dekat.

# BAB 5 KESIMPULAN DAN SARAN

## 5.1 Kesimpulan

Dengan menggunakan pendekatan algoritma greedy, penulis telah berhasil mengembangkan strategi untuk bot dalam permainan Etimo Diamonds. Program akhir yang penulis hasilkan merupakan hasil gabungan dari beberapa solusi *greedy* yang telah penulis uji, dengan tujuan menghasilkan strategi terbaik menurut penilaian penulis.

## 5.2 Saran

Saran untuk pengembang lain yang ingin mengembangkan permainan Diamonds:

1. Kombinasikan berbagai solusi algoritma *greedy* yang ada untuk menciptakan strategi bot yang efektif dalam berbagai situasi.
2. Terkadang, solusi yang paling optimal dapat dihasilkan dari program yang memiliki kode sederhana dan memperhatikan objek-objek yang ada di dalam permainan.

# LAMPIRAN

1. **Link Repository Github** : https://github.com/rizqikapratamaa/Tubes1_ALL-IN-RM
2. **Link Video** : https://youtu.be/zgiZnn6jxe8

# DAFTAR PUSTAKA


Rinaldi Munir. 2022. "Algoritma Brute Force (Bagiann 1)"
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf .

Rinaldi Munir. 2021. "Algoritma Brute Force (Bagiann 2)"
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf .

Rinaldi Munir. 2021. "Algoritma Brute Force (Bagiann 3)"
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf .

Szabo, Fred E. 2015. "The Linear Algebra Survival Guide : Illustrated with Mathematica" Manhattan Distance. ISBN 978-0-12-409520.
Manhattan Distance - an overview | ScienceDirect Topics .

Brownlee, Jason. 2020. 4 Distance Measures for Machine Learning.
4 Distance Measures for Machine Learning - MachineLearningMastery.com .