

# Laporan Tugas Besar 1

## IF2211 Strategi Algoritma

Pemanfaatan Algoritma Greedy dalam  
Pembuatan Bot Permainan Diamonds



Disusun oleh:

Akbar Al Fattah	13522036
Adril Putra Merin	13522068
Christopher Brian	13522106

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

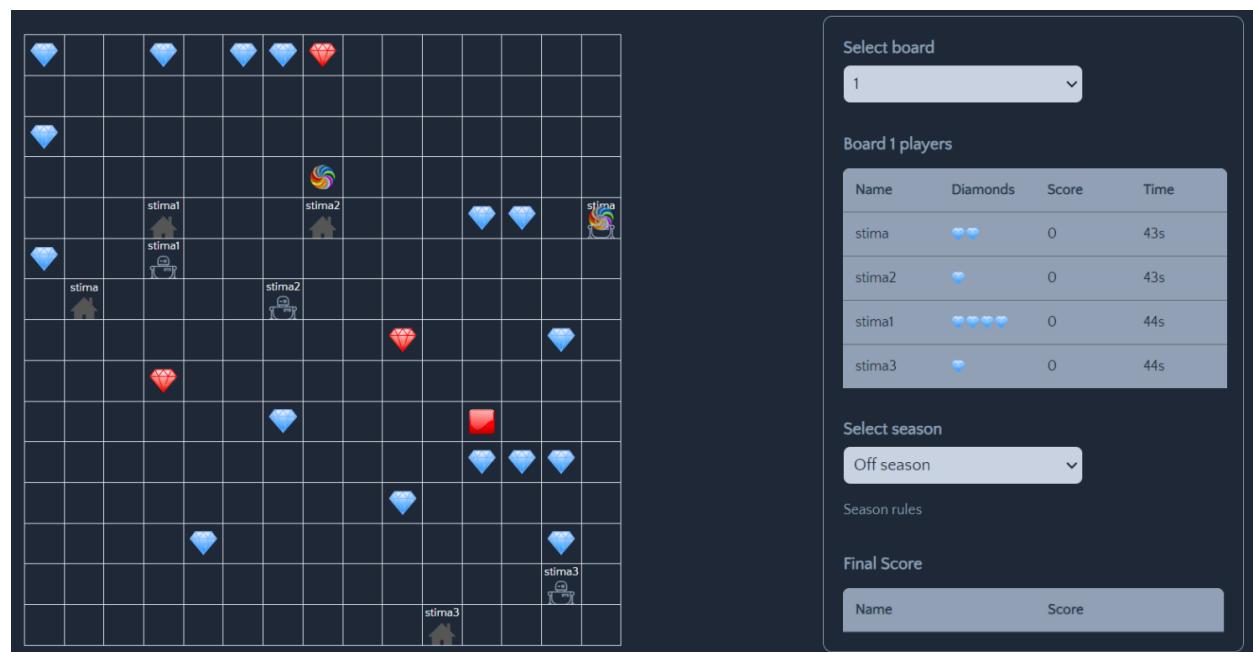
# Daftar Isi

<b>Daftar Isi</b>	<b>2</b>
<b>Bab 1: Deskripsi Tugas</b>	<b>3</b>
1.1. Deskripsi Umum	3
1.2. Komponen	4
1.2.1. <i>Diamonds</i>	4
1.2.2. <i>Red Button/Diamond Button</i>	4
1.2.3. <i>Teleporters</i>	4
1.2.4. <i>Bots and Bases</i>	4
1.2.5. <i>Inventory</i>	4
1.3. Cara Kerja Permainan Diamonds	5
<b>Bab 2: Landasan Teori</b>	<b>6</b>
2.1. Dasar Teori	6
2.2. Cara Kerja Program	6
2.2.1. Cara Kerja <i>Bot</i>	6
2.2.2. Cara Implementasi Algoritma Greedy ke dalam <i>Bot</i>	7
2.2.3. Cara Menjalankan <i>Bot</i>	7
<b>Bab 3: Aplikasi Strategi Greedy</b>	<b>8</b>
3.1. <i>Mapping</i> Persoalan	8
3.2. Eksplorasi Alternatif Solusi Greedy	8
3.2.1. <i>Naive Shortest Distance</i>	8
3.2.2. <i>Find Highest Density</i>	9
3.2.2. <i>Dynamic Shortest Distance</i>	9
3.3. Analisis Efisiensi dan Efektivitas Alternatif Solusi Greedy	11
3.3.1. Analisis <i>Naive Shortest Distance</i>	11
3.3.1. Analisis <i>Find Highest Density</i>	12
3.3.1. Analisis <i>Dynamic Shortest Distance</i>	13
3.4. Strategi Greedy Pilihan	14
<b>Bab 4: Implementasi dan Pengujian</b>	<b>16</b>
4.1. Implementasi Algoritma Greedy	16
4.2. Struktur Data <i>Bot</i>	24
4.3. Analisis Desain Solusi Algoritma Greedy	25
<b>Bab 5: Kesimpulan dan Saran</b>	<b>30</b>
<b>Lampiran</b>	<b>31</b>
Link Repository GitHub	31
Link Video	31
<b>Daftar Pustaka</b>	<b>32</b>

# Bab 1: Deskripsi Tugas

## 1.1. Deskripsi Umum

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, kami diminta untuk membuat sebuah *bot* yang nantinya akan dipertandingkan satu sama lain. Tentunya kami 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 diimplementasikan dengan algoritma *greedy* untuk *bot*)
- c. Program utama (*main*) dan utilitas lainnya

## 1.2. Komponen

Berikut adalah komponen-komponen dari permainan Diamonds.

### 1.2.1. *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*.

### 1.2.2. *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.

### 1.2.3. *Teleporters*

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

### 1.2.4. *Bots and Bases*

Pada *game* ini kita akan menggerakkan *bot* untuk mendapatkan *diamond* sebanyak banyaknya. Semua *bot* memiliki sebuah *Base* di mana *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.

### 1.2.5. *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.

### 1.3. Cara Kerja Permainan Diamonds

Untuk mengetahui *flow* dari *game* ini, berikut ini adalah cara kerja permainan Diamonds.

1. Pertama, setiap pemain (*bot*) akan ditempatkan pada *board* secara *random*. Masing-masing *bot* akan mempunyai *home base*, serta memiliki *score* dan *inventory* awal bernilai nol.
2. Setiap *bot* diberikan waktu untuk bergerak, waktu yang diberikan semua sama untuk setiap pemain.
3. Objektif utama *bot* adalah mengambil *diamond-diamond* yang ada di peta sebanyak-banyaknya. Seperti yang sudah disebutkan di atas, *diamond* yang berwarna merah memiliki 2 poin dan *diamond* yang berwarna biru memiliki 1 poin.
4. Setiap bot juga memiliki sebuah *inventory*, dimana *inventory* berfungsi sebagai tempat penyimpanan sementara *diamond* yang telah diambil. *Inventory* ini sewaktu-waktu bisa penuh, maka dari itu *bot* harus segera kembali ke *home base*.
5. Apabila *bot* menuju ke posisi *home base*, *score* *bot* akan bertambah senilai *diamond* yang tersimpan pada *inventory* dan *inventory* *bot* akan menjadi kosong kembali.
6. Usahakan agar *bot* anda tidak bertemu dengan *bot* lawan. Jika *bot* A menimpa posisi *bot* B, *bot* B akan dikirim ke *home base* dan semua *diamond* pada *inventory* *bot* B akan hilang, diambil masuk ke *inventory* *bot* A (istilahnya *tackle*).
7. Selain itu, terdapat beberapa fitur tambahan seperti *teleporter* dan *red button* yang dapat digunakan apabila anda menuju posisi objek tersebut.
8. Apabila waktu seluruh *bot* telah berakhir, maka permainan berakhir. Score masing-masing pemain akan ditampilkan pada tabel *Final Score* di sisi kanan layar.

## Bab 2: Landasan Teori

### 2.1. Dasar Teori

Algoritma *greedy* meliputi semua algoritma yang mengikuti heuristik pemecahan masalah yang membuat pilihan optimal lokal di setiap langkah. Dalam banyak permasalahan, strategi *greedy* tidak menghasilkan solusi optimal, tetapi heuristik *greedy* bisa membuat solusi-solusi yang optimal secara lokal yang mengaproksimasikan solusi optimal global dalam waktu yang lebih singkat. Sebuah algoritma *greedy* adalah pendekatan yang memecahkan permasalahan dengan memilih opsi terbaik yang tersedia di setiap saat. Algoritma *greedy* tidak mementingkan apakah hasil terbaik saat ini akan menghasilkan hasil terbaik secara keseluruhan. Algoritma *greedy* juga tidak pernah membalikkan keputusan yang telah dibuat sebelumnya bahkan jika ternyata keputusan yang diambil salah. Hal ini dikarenakan algoritma *greedy* selalu menggunakan opsi terbaik lokal untuk memprediksi hasil terbaik global. Keuntungan dari algoritma *greedy* yaitu lebih mudah untuk dideskripsikan dan umumnya dapat bekerja lebih cepat dibanding algoritma lainnya. Di samping itu, kekurangan dari algoritma *greedy*, seperti yang telah disebutkan sebelumnya, yaitu tidak selalu menghasilkan solusi optimal global.

### 2.2. Cara Kerja Program

#### 2.2.1. Cara Kerja *Bot*

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

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

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

### 2.2.2. Cara Implementasi Algoritma *Greedy* ke dalam *Bot*

1. Buatlah *folder* baru pada direktori `/game/logic` (misalnya `mybot.py`)
2. Buatlah kelas yang meng-*inherit* kelas `BaseLogic`, lalu implementasikan *constructor* dan *method* `next_move` pada kelas tersebut
3. *Import* kelas yang telah dibuat pada `main.py` dan daftarkan pada *dictionary* `CONTROLLERS`

### 2.2.3. Cara Menjalankan *Bot*

Jalankan program seperti step c pada bagian 2 (sesuaikan argumen *logic* pada *command/script* tersebut menjadi nama *bot* yang telah terdaftar pada `CONTROLLERS`). Anda bisa menjalankan satu *bot* saja atau beberapa *bot* menggunakan .bat atau .sh *script*.

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

## Bab 3: Aplikasi Strategi Greedy

### 3.1. Mapping Persoalan

Secara umum, persoalan game Diamonds bisa dipetakan menurut heuristik greedy sebagai berikut.

1. Himpunan Kandidat: Himpunan *diamonds* yang terdapat dalam *board*.
2. Himpunan solusi: *Diamonds* yang terpilih
3. Fungsi solusi: Memeriksa apakah kapasitas *inventory* sudah penuh dan sisa waktu yang tersisa
4. Fungsi seleksi: Memilih *diamond* yang paling optimal berdasarkan strategi yang dipilih
5. Fungsi kelayakan: Memeriksa apakah *diamond* yang dipilih, jika ditambahkan ke dalam *inventory*, tidak melebihi kapasitas *inventory*
6. Fungsi obyektif: Mengumpulkan poin sebanyak mungkin sebelum waktu habis.

### 3.2. Eksplorasi Alternatif Solusi Greedy

#### 3.2.1. Naive Shortest Distance

Algoritma greedy ini mungkin merupakan pilihan algoritma yang cara kerjanya paling *straightforward* dan sederhana. Algoritma *shortest distance* murni bekerja dengan memilih *diamond* terdekat dari posisi saat ini di setiap langkah. Dalam menghitung jarak *diamond* terdekat, algoritma ini juga akan memperhitungkan penggunaan *teleporter*. *Bot* hanya akan kembali ke *base* jika waktu yang tersisa sedikit atau *inventory bot* sudah penuh. *Bot* ini tidak memperhitungkan jarak *diamond* dengan *base*, total jarak yang ditempuh, ataupun densitas *diamond* di area tertentu. Adapun *mapping* strategi ini ke dalam persoalan greedy adalah sebagai berikut.

Himpunan Kandidat	Himpunan <i>GameObjects</i> yang terdapat pada papan permainan
Himpunan Solusi	<i>Himpunan GameObjects</i> yang terpilih
Fungsi Solusi	Memeriksa apakah kapasitas <i>inventory</i> sudah penuh dan apakah sisa waktu masih cukup
Fungsi Seleksi	Memilih <i>diamond</i> yang terdekat sebagai target, dapat melalui <i>teleports</i> maupun tidak. Jika <i>inventory</i> sudah penuh, pilih <i>base</i> sebagai target.
Fungsi Kelayakan	Memeriksa apakah <i>diamond</i> yang dipilih, jika ditambahkan ke dalam <i>inventory</i> , tidak melebihi kapasitas <i>inventory</i> .
Fungsi Objektif	Mengumpulkan waktu sebanyak mungkin sebelum waktu habis

Strategi Tambahan	Jika jarak antara bot dan base ditambah 2 lebih besar daripada waktu yang tersisa (dalam sekon), bot akan kembali ke base jika <i>inventory</i> tidak kosong
-------------------	--

### 3.2.2. Find Highest Density

Algoritma *greedy* ini pada dasarnya mencari area pada *board* dengan densitas/jumlah *diamond* terbesar. Algoritma ini bekerja dengan melakukan *scanning* pada setiap blok area dengan ukuran yang telah ditentukan pada *board*, kemudian bergerak pada blok area dengan densitas terbesar. Bot hanya akan kembali ke *base* jika waktu yang tersisa sedikit atau *inventory bot* sudah penuh. *Bot* ini tidak memperhitungkan jarak *diamond* dengan *base* ataupun total jarak yang ditempuh. Adapun *mapping* strategi ini ke dalam persoalan *greedy* adalah sebagai berikut.

Himpunan Kandidat	Himpunan <i>GameObjects</i> yang terdapat pada papan permainan
Himpunan Solusi	<i>Himpunan GameObjects</i> yang terpilih
Fungsi Solusi	Memeriksa apakah kapasitas <i>inventory</i> sudah penuh dan apakah sisa waktu masih cukup
Fungsi Seleksi	Memilih area dengan jumlah <i>diamond</i> terbanyak dan kemudian mengambil <i>diamond</i> yang ada pada area tersebut hingga habis
Fungsi Kelayakan	Memeriksa apakah <i>diamond</i> yang dipilih, jika ditambahkan ke dalam <i>inventory</i> , tidak melebihi kapasitas <i>inventory</i> .
Fungsi Objektif	Mengumpulkan waktu sebanyak mungkin sebelum waktu ha
Strategi Tambahan	Jika jarak antara bot dan base ditambah 2 lebih besar daripada waktu yang tersisa (dalam sekon), bot akan kembali ke base jika <i>inventory</i> tidak kosong

### 3.2.3. Dynamic Shortest Distance

Algoritma *greedy* ini bekerja dengan mencari *diamond* optimal, yaitu *diamond* dengan skor terendah. Skor yang didapatkan oleh tiap *diamond* bergantung pada banyak *diamond* yang dibutuhkan saat ini dan jumlah poin dari suatu *diamond*. Dalam hal ini, skor yang dapat diperoleh dibagi menjadi 3 kasus, yaitu:

1. Jumlah *diamond* yang dibutuhkan adalah satu  
Pada kasus ini, *diamond* yang mungkin diambil hanyalah *diamond* yang memiliki

- 1 poin. Karena itu, skor pada kasus ini adalah jarak *bot* ke *diamond* ditambah jarak *diamond* tersebut ke *base* (*curr\_dist* + *base\_dist*).
2. Jumlah *diamond* yang dibutuhkan adalah dua  
Kasus ini dapat dibagi menjadi dua lagi. Pertama, jika *diamond* yang ingin dipilih memiliki 1 poin dan terdapat *diamond* lain selain *diamond* yang akan dipilih. Skor yang didapat oleh kasus ini adalah jarak *bot* ke *diamond* yang akan dipilih, ditambah jarak *diamond* yang akan dipilih ke *diamond* terdekat lain (relatif terhadap *diamond* yang akan dipilih), ditambah jarak dari *diamond* terdekat tersebut ke *base*. Kedua, jika *diamond* yang dipilih memiliki 1 poin, tetapi tidak terdapat *diamond* lain, selain *diamond* yang akan dipilih, atau jika *diamond* yang dipilih memiliki 2 poin. Skor yang didapat oleh kasus ini hanya jarak *bot* ke *diamond* yang akan dipilih, ditambah dengan jarak *diamond* yang akan dipilih ke *diamond* terdekat lain (relatif terhadap *diamond* yang akan dipilih).
  3. Jumlah *diamond* yang dibutuhkan lebih dari dua  
Kasus ini juga dibagi menjadi dua. Pertama, jika *diamond* yang akan dipilih memiliki 1 poin dan terdapat *diamond* lain selain *diamond* yang akan dipilih. Skor yang didapat oleh kasus ini adalah jarak *bot* ke *diamond* yang akan dipilih, ditambah jarak *diamond* yang akan dipilih ke *diamond* terdekat lain (relatif terhadap *diamond* yang akan dipilih). Kedua, jika *diamond* yang akan dipilih memiliki 1 poin tetapi tidak terdapat *diamond* lain, atau jika *diamond* yang akan dipilih memiliki 2 poin. Skor yang didapat oleh kasus ini hanyalah jarak *bot* ke *diamond* yang akan dipilih. Jika skor yang dimiliki sama, jarak dari *diamond* ke *base* akan diperhitungkan.

Selain pencarian skor, terdapat beberapa strategi tambahan pada algoritma ini. Jika, jarak antara *bot* dan *base* ditambah 2 lebih besar daripada waktu yang tersisa (dalam sekon), *bot* akan kembali ke *base* jika *inventory* tidak kosong. Selain itu, jika *inventory* belum penuh, tetapi *base* terletak di antara *bot* dan *diamond* target, *bot* akan melalui *base* terlebih dahulu untuk menyetor *diamond* yang sudah dikumpulkan sehingga kapasitas *diamond* menjadi banyak lagi. Hal yang sama juga dilakukan jika terdapat *red button* di antara *bot* dan *base* ketika *bot* hendak kembali ke *base*. Adapun *mapping* persoalan ini ke dalam persoalan *greedy* adalah sebagai berikut.

Himpunan Kandidat	Himpunan <i>GameObjects</i> yang terdapat pada papan permainan
Himpunan Solusi	Himpunan <i>GameObjects</i> yang terpilih
Fungsi Solusi	Memeriksa apakah kapasitas <i>inventory</i> sudah penuh dan apakah sisa waktu masih cukup
Fungsi Seleksi	Memilih <i>diamond</i> dengan skor paling kecil sebagai target dan kembali ke <i>base</i> jika <i>inventory</i> sudah penuh.
Fungsi Kelayakan	Memeriksa apakah <i>diamond</i> yang dipilih, jika ditambahkan ke dalam <i>inventory</i> , tidak melebihi kapasitas <i>inventory</i> .

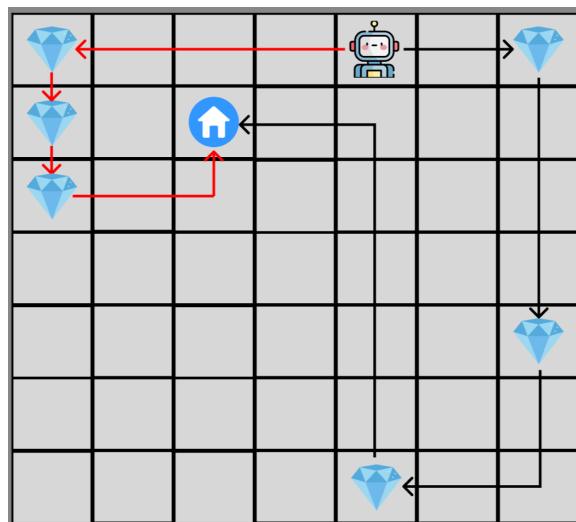
Fungsi Objektif	Mengumpulkan waktu sebanyak mungkin sebelum waktu habis
Strategi Tambahan	Jika jarak antara bot dan base ditambah 2 lebih besar daripada waktu yang tersisa (dalam sekon), bot akan kembali ke base jika <i>inventory</i> tidak kosong. Jika <i>inventory</i> belum penuh, tetapi <i>base</i> terletak di antara <i>bot</i> dan <i>diamond target</i> , <i>bot</i> akan melalui <i>base</i> terlebih dahulu. Jika terdapat <i>red button</i> di antara <i>bot</i> dan <i>base</i> ketika <i>bot</i> hendak kembali ke <i>base</i> , <i>bot</i> akan melalui <i>red button</i> terlebih dahulu.

### 3.3. Analisis Efisiensi dan Efektivitas Alternatif Solusi Greedy

#### 3.3.1. Analisis Naive Shortest Distance

Algoritma ini bekerja dengan melakukan iterasi terhadap setiap diamond yang tersedia dan menghitung jarak antara bot dan diamond terkait. Diamond yang dipilih adalah diamond dengan jarak terpendek relatif terhadap bot. Operasi dengan perhitungan jarak dilakukan dengan mempertimbangkan jarak jika melalui teleporter. Karena teleporter hanya dua, kompleksitas waktu big-O pada proses perhitungan jarak adalah  $O(1)$ . Jadi, kompleksitas waktu total adalah  $T(n) = cn$  dimana  $c$  adalah konstanta yang menggambarkan banyak operasi yang dilakukan terhadap suatu diamond dan  $n$  adalah banyak diamond yang tersedia. Karena itu, kompleksitas waktu total dalam notasi big-O adalah  $O(cn) = O(n)$ .

Algoritma ini tidak selalu memberikan solusi optimal, walaupun dalam banyak kasus, langkah yang dipilih sudah optimal. Salah satu contoh kasus dimana algoritma ini gagal memberikan solusi yang optimal adalah sebagai berikut



Misalkan bot tersebut memiliki inventory dengan ukuran lima diamond dan sudah terisi sebanyak dua diamond sehingga hanya perlu mengambil tiga diamond sebelum kembali ke

base. Jalur yang diambil oleh algoritma ini adalah ditandai dengan warna hitam, sedangkan jalur yang optimal ditandai dengan warna merah. Pada jalur yang diambil oleh algoritma ini, total jarak yang ditempuh adalah 17 blok, sedangkan total jarak yang ditempuh oleh jalur yang optimal adalah 9 blok.

### 3.3.2. Analisis *Find Highest Density*

Algoritma ini bekerja dengan melakukan *scanning* pada papan permainan dengan area tertentu dan kemudian mengambil semua diamond pada area tersebut. Tahap pertama algoritma ini adalah proses *scanning*. Misalkan area scanning yang dipilih adalah  $k \times k$  dengan  $k \leq n$  dan  $k \leq m$  untuk sebuah papan permainan dengan ukuran  $n \times m$ . Karena itu, proses *scanning* dilakukan sebanyak  $(n - k)(m - k)$  kali. Untuk setiap proses *scanning*, akan dilakukan iterasi terhadap setiap *diamond* yang tersedia untuk memeriksa apakah *diamond* tersebut terletak pada area *scanning* dan menyimpan setiap diamond yang terletak pada area pencarian tersebut (dapat menggunakan *list*). Proses tersebut, dilakukan sebanyak  $d$  kali, dengan  $d$  adalah banyak diamond. Jadi, kompleksitas waktu untuk tahap ini adalah

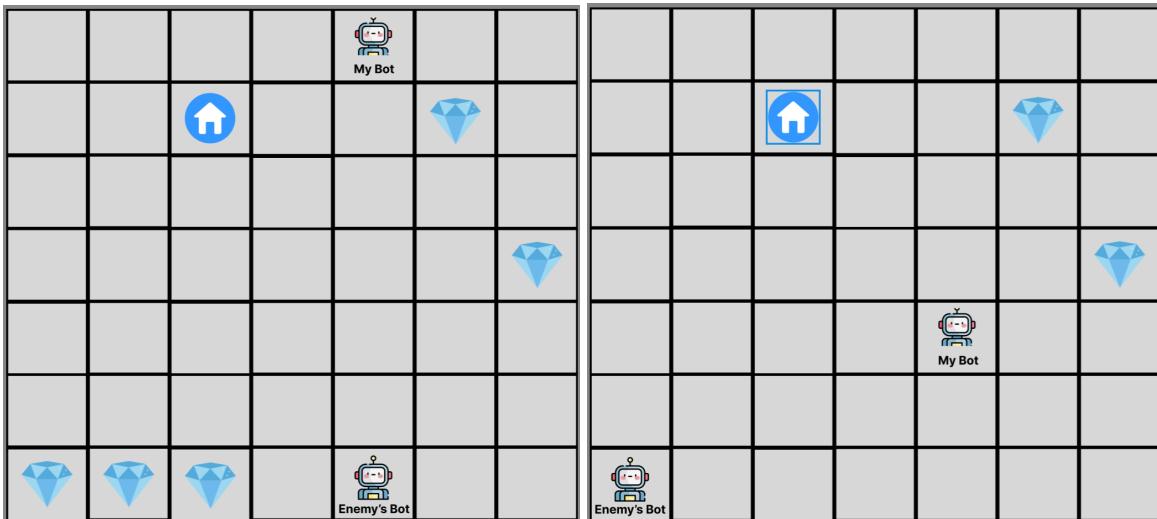
$$T(n, m, k, d) = (n - k)(m - k)d$$

Nilai  $k$  relatif kecil terhadap  $n$  dan  $m$  sehingga  $(n - k)(m - k)d \leq nmd$ . Jadi kompleksitas waktu dalam notasi big-O adalah

$$O((n - k)(m - k)d) = O(nmd)$$

Tahap kedua algoritma ini adalah mengambil semua *diamond* pada area yang terpilih berdasarkan *list* dari posisi *diamond* yang berada pada area ini. Jika suatu *diamond* pada *list* sudah diambil oleh *bot* lain, pergi ke *diamond* selanjutnya yang belum diambil. Jika *diamond* pada area tersebut sudah habis, kembali lakukan proses *scanning*. Kompleksitas waktu pada tahap ini adalah  $O(n)$  karena kita perlu memeriksa apakah *diamond* yang ingin dituju sudah diambil atau belum.

Berdasarkan analisis kompleksitas diatas, dapat dilihat bahwa proses kalkulasi untuk algoritma ini memakan waktu yang cukup besar. Algoritma ini sering kali tidak memberikan solusi karena hanya mempertimbangkan area terpadat saja, padahal jarak terpendek adalah salah satu parameter penting yang harus diperhatikan juga. Selain itu, algoritma ini sangat tidak dinamis, terutama pada tahap kedua, karena terlalu berfokus pada *list* posisi *diamond* yang dikalkulasi pada tahap pertama. Hal ini menjadi sangat tidak efektif ketika *diamond-diamond* pada area yang dituju, diambil oleh *bot* lain saat *bot* kita sedang berjalan menuju area tersebut. Untuk lebih jelasnya, kondisi ini dapat dilihat pada ilustrasi berikut.



Misalkan area pencarian pada ilustrasi tersebut adalah  $3 \times 3$ , maka area terpadat yang akan dipilih oleh *My Bot* adalah area pada kiri bawah. Di sisi lain, terdapat *Enemy's Bot* yang mungkin menggunakan algoritma dengan jarak terpendek sehingga terlebih dahulu mengambil *diamond* pada area tersebut. Saat *My Bot* sudah setengah jalan, *diamond* pada area yang ingin dituju sudah habis sehingga *bot* tersebut perlu melakukan kalkulasi ulang dan tidak mendapatkan apa-apa.

### 3.3.3. Analisis Dynamic Shortest Distance

Algoritma bekerja dengan melakukan iterasi terhadap seluruh *diamond* yang tersedia dan melakukan kalkulasi skor untuk *diamond* berdasarkan poin yang dimiliki oleh *diamond* dan banyak *diamond* yang dibutuhkan saat ini (ukuran inventory - banyak *diamond* terkumpul). *Diamond* dengan skor paling kecil kemudian dipilih sebagai target.

Algoritma ini dimulai dengan melakukan *preprocessing* untuk menghitung jarak antara *diamond* ke-*i* dan *diamond* ke-*j* dengan *diamond* ke-*j* adalah *diamond* terdekat relatif terhadap *diamond* ke-*i*. Hal ini dilakukan dengan melakukan perhitungan jarak antara tiap pasang *diamond* dan melakukan penyimpanan jarak minimum dan indeks *diamond* yang berkaitan secara dinamis. Hal ini dilakukan sebanyak  $\frac{n(n-1)}{2}$  kali, dengan *n* adalah banyak *diamond*. Selain itu, jarak antara tiap *diamond* ke *base* juga disimpan. Jika dilakukan secara efisien, kompleksitas waktu dari *preprocessing* adalah

$$T(n) = \frac{cn(n-1)}{2} = O\left(\frac{cn^2 - cn}{2}\right) = O(n^2), \text{ dengan } c \text{ konstanta}$$

Selanjutnya, algoritma ini akan melakukan kalkulasi skor untuk setiap *diamond* sesuai dengan poin yang dimiliki *diamond* dan banyak *diamond* yang dibutuhkan (sudah dijelaskan pada bagian sebelumnya). *Diamond* dengan skor paling kecil adalah target destinasi yang akan dituju. Kompleksitas waktu dari proses ini adalah

$$T(n) = kn = O(kn) = O(n), \text{ dengan } k \text{ konstanta}$$

Jadi, kompleksitas waktu keseluruhan proses adalah

$$T(n) = \frac{cn(n-1)}{2} + kn = \frac{cn^2}{2} + (k - \frac{c}{2})n = O(\frac{cn^2}{2}) = O(n^2)$$

Pada sebagian kasus, algoritma ini menjamin bahwa *diamond* yang dipilih pasti optimal, terutama pada kasus dimana *diamond* yang dibutuhkan adalah satu atau *diamond* yang dibutuhkan adalah dua.

Pada kasus dimana *diamond* yang dibutuhkan adalah satu, skor akan dikalkulasi berdasarkan jarak minimum antara *bot* dan *diamond*, ditambah jarak antara *diamond* dan *base*. Semakin kecil skor suatu *diamond*, semakin sedikit pula langkah yang dibutuhkan untuk mengambil *diamond* dan kembali ke *base*. Jadi, algoritma ini akan memberikan solusi optimal pada kasus ini.

Pada kasus dimana *diamond* yang dibutuhkan adalah dua. Jika poin dari *diamond* yang dipilih adalah dua, maka skor saat ini adalah jarak *bot* ke *diamond* ditambah jarak *diamond* ke *base*. Jika poin dari *diamond* yang dipilih adalah satu, maka skor saat ini adalah jarak *bot* ke *diamond* yang dipilih, ditambah jarak *diamond* terpilih tersebut ke *diamond* terdekat relatif terhadap *diamond* terpilih, ditambah jarak *diamond* terdekat tersebut ke *base*. Pada dasarnya, skor adalah banyak langkah untuk mengambil *diamond* dan kembali ke *base* sehingga semakin kecil skor, semakin sedikit pula langkah yang dibutuhkan. Karena itu, algoritma ini menjamin bahwa *diamond* yang dipilih adalah yang paling optimal.

Untuk kasus dimana *diamond* yang dibutuhkan lebih dari dua, skor akan dihitung berdasarkan poin yang dimiliki dari suatu *diamond*. Jika *diamond* memiliki satu poin, skor adalah jarak *bot* ke *diamond*, ditambah jarak *diamond* ke *diamond* terdekat relatif terhadap *diamond* sebelumnya. Jika *diamond* memiliki dua poin, skor adalah jarak *bot* ke *diamond* saja. Hal ini dilakukan untuk membandingkan apakah pengambilan *diamond* dengan poin dua membutuhkan langkah yang lebih sedikit dibandingkan mengambil dua *diamond*. Pada kasus ini, algoritma ini seringkali memberikan solusi yang optimal. Namun, harus diakui bahwa algoritma ini tidak dapat menjamin bahwa solusi yang diberikan adalah yang paling optimal.

### 3.4. Strategi Greedy Pilihan

Berdasarkan analisis dan penjelasan diatas, kelompok kami memilih algoritma *dynamic shortest distance* untuk dijadikan strategi utama. Secara umum, algoritma *dynamic shortest distance* dijamin dapat memberikan solusi optimal pada sebagian kasus dan seringkali memberikan solusi optimal pada kasus lainnya. Walaupun begitu, jika variabel eksternal seperti *bot* lain dilibatkan, algoritma ini tidak dapat menjamin bahwa langkah yang diambil selalu optimal karena *bot* lain dapat men-tackle *bot* kita dan mengambil *diamond* yang kita inginkan. Pertimbangan lain dari algoritma ini adalah proses perhitungan yang cukup cepat untuk jumlah *diamond* yang tidak terlalu banyak.

Alasan kami tidak memilih *naive shortest distance* sebagai strategi utama adalah karena algoritma tersebut tidak menjamin bahwa solusi yang dipilih adalah solusi yang optimal pada kasus apapun, walaupun dalam banyak kejadian, solusi yang diberikan memang optimal. Selain itu, variabel seperti banyak poin suatu *diamond* dan jarak *diamond* ke base juga tidak dijadikan pertimbangan dari algoritma ini sehingga memiliki potensi yang lebih besar untuk memberikan solusi yang tidak optimal.

Alasan kami tidak memilih algoritma *find highest density* sebagai strategi utama adalah algoritma tersebut memiliki kompleksitas waktu yang cukup tinggi. Hal ini karena algoritma tersebut bergantung pada ukuran papan permainan. Selain itu, algoritma ini sangat tidak dinamis (seperti dibahas pada bagian sebelumnya) sehingga sangat berpotensi untuk memberikan solusi yang tidak optimal. Algoritma ini juga tidak mempertimbangkan jarak antara *bot* dan area terpadat.

## Bab 4: Implementasi dan Uji Coba

### 4.1. Implementasi Algoritma Greedy

```
# constants
MAX_INT: int <- float("inf")

# UTILITY FUNCTIONS
# TODO: return manhattan distance between two points
function count_pos_dist(p1: Position, p2: Position)
    -> abs(p2.x - p1.x) + abs(p2.y - p1.y)

# TODO: return minimum distance between two points (current point and
destination point) by considering if an object should go through teleports
or not
# this function will return a tuple consists of the minimum distance, the
target position, and a boolean use_portal
# if the minimum distance is achieved by stepping through the teleports,
the target position will be the corresponding teleport and use_portal will
be true
# otherwise, the target position will be the destination position and
use_portal will be false
function get_minimum_dist(curr: Position, dest: Position, teleport_list:
arrayofGameObject, board_width: int, board_height: int) -> int, Position,
bool
    # count relative distance without teleports
    curr_dist <- count_pos_dist(curr, dest)
    init_curr_dist <- curr_dist
    curr_target <- dest
    # These long if-else codes aim to find the exact distance from current
position to a destination without utilizing
    # teleports. This codes consider almost all (if not all) edge cases that
occurs if the current x-axis or y-axis position
    # only have one step difference from the destination x-axis or y-axis
position, and the teleports are in the area of those two points

    # CASE 1: current y-axis position only have 1 step difference from
destination y-axis position
    # check if the teleports are in the area between those two points
    if (curr.x <= teleport_list[0].position.x <= dest.x or curr.x >=
teleport_list[0].position.x >= dest.x) and (curr.x <=
teleport_list[1].position.x <= dest.x or curr.x >=
teleport_list[1].position.x >= dest.x) then
        if curr.y = dest.y then
            if teleport_list[0].position.y = dest.y then
                if teleport_list[1].position.y = teleport_list[0].position.y then
                    curr_dist <- curr_dist + 2
                else if teleport_list[1].position.y = (teleport_list[0].position.y
- 1) then
                    if (curr.y + 1) < board_height then
                        curr_dist <- curr_dist + 2
                    else
                        curr_dist <- curr_dist + 4
                    if teleport_list[1].position.x = dest.x then
                        curr_dist <- curr_dist + 2
                else if teleport_list[1].position.y = (teleport_list[0].position.y
+ 1) then
                    if (curr.y - 1) >= 0 then
                        curr_dist <- curr_dist + 2
                    else
                        curr_dist <- curr_dist + 4
                    if teleport_list[1].position.x = dest.x then
                        curr_dist <- curr_dist + 2

```

```

else if teleport_list[1].position.y = dest.y then
    if teleport_list[1].position.y = teleport_list[0].position.y then
        curr_dist <- curr_dist + 2
    else if (teleport_list[1].position.y - 1) =
teleport_list[0].position.y then
        if (curr.y + 1) < board_height then
            curr_dist <- curr_dist + 2
        else
            curr_dist <- curr_dist + 4
        if teleport_list[0].position.x = dest.x then
            curr_dist <- curr_dist + 2
    else if (teleport_list[1].position.y + 1) =
teleport_list[0].position.y then
        if (curr.y - 1) >= 0 then
            curr_dist <- curr_dist + 2
        else
            curr_dist <- curr_dist 4
        if teleport_list[0].position.x = dest.x then
            curr_dist <- curr_dist + 2
    else if curr.y = (dest.y - 1) then
        if curr.y = teleport_list[0].position.y and dest.y =
teleport_list[1].position.y then
            curr_dist <- curr_dist + 3
        if (dest.y + 1) >= board_height and teleport_list[0].position.x =
dest.x then
            curr_dist <- curr_dist + 2
    else if curr.y = teleport_list[1].position.y and dest.y =
teleport_list[0].position.y then
        curr_dist <- curr_dist + 3
        if (dest.y + 1) >= board_height and teleport_list[1].position.x =
dest.x then
            curr_dist <- curr_dist + 2
    else if curr.y = teleport_list[1].position.y and dest.y =
teleport_list[0].position.y then
        curr_dist <- curr_dist + 3
        if (dest.y + 1) >= board_height and teleport_list[1].position.x =
dest.x then
            curr_dist <- curr_dist + 2
    else if curr.y = dest.y + 1 then
        if curr.y = teleport_list[0].position.y and dest.y =
teleport_list[1].position.y then
            curr_dist <- curr_dist + 3
            if (dest.y - 1) < 0 and teleport_list[0].position.x = dest.x then
                curr_dist <- curr_dist + 2
        else if curr.y = teleport_list[1].position.y and dest.y =
teleport_list[0].position.y then
            curr_dist <- curr_dist + 3
            if (dest.y - 1) < 0 and teleport_list[1].position.x = dest.x then
                curr_dist <- curr_dist + 2

# CASE 2: current x-axis position only have 1 step difference from
destination y-axis position
# check if the teleports are in the area between those two points
if (curr.y <= teleport_list[0].position.y <= dest.y or curr.y >=
teleport_list[0].position.y >= dest.y) and (curr.y <=
teleport_list[1].position.y <= dest.y or curr.y >=
teleport_list[1].position.y >= dest.y) and curr_dist = init_curr_dist
then
    if curr.x = dest.x then
        if teleport_list[0].position.x = dest.x then
            if teleport_list[1].position.x = teleport_list[0].position.x then
                curr_dist <- curr_dist + 2
            else if teleport_list[1].position.x = (teleport_list[0].position.x
- 1) then
                if (curr.x + 1) < board_width then
                    curr_dist <- curr_dist + 2
                else

```

```

        curr_dist <- curr_dist + 4
        if teleport_list[1].position.y = dest.y then
            curr_dist <- curr_dist + 2
        else if teleport_list[1].position.x = (teleport_list[0].position.x
+ 1) then
            if (curr.x - 1) >= 0 then
                curr_dist <- curr_dist + 2
            else
                curr_dist <- curr_dist + 4
                if teleport_list[1].position.y = dest.y then
                    curr_dist <- curr_dist + 2
            else if teleport_list[1].position.x = dest.x then
                if teleport_list[1].position.x = teleport_list[0].position.x:
                    curr_dist <- curr_dist + 2
                else if (teleport_list[1].position.x - 1) =
                    teleport_list[0].position.x then
                    if (curr.x + 1) < board_width then
                        curr_dist <- curr_dist + 2
                    else
                        curr_dist <- curr_dist + 4
                        if teleport_list[0].position.y = dest.y then
                            curr_dist <- curr_dist + 2
                else if (teleport_list[1].position.x + 1) =
                    teleport_list[0].position.x then
                    if (curr.x - 1) >= 0 then
                        curr_dist <- curr_dist + 2
                    else
                        curr_dist <- curr_dist + 4
                        if teleport_list[0].position.y = dest.y then
                            curr_dist <- curr_dist + 2
            else if curr.x = (dest.x - 1) then
                if curr.x = teleport_list[0].position.x and dest.x =
                    teleport_list[1].position.x then
                    curr_dist <- curr_dist + 3
                    if (dest.x + 1) >= board_width and teleport_list[0].position.y =
                        dest.y then
                        curr_dist <- curr_dist + 2
                else if curr.x = teleport_list[1].position.x and dest.x =
                    teleport list[0].position.x then
                    curr_dist <- curr_dist + 3
                    if (dest.x + 1) >= board_width and teleport_list[1].position.y =
                        dest.y then
                        curr_dist <- curr_dist + 2
                else if curr.x = (dest.x + 1) then
                    if curr.x = teleport_list[0].position.x and dest.x =
                        teleport_list[1].position.x then
                        curr_dist <- curr_dist + 3
                        if (dest.x - 1) < 0 and teleport_list[0].position.y = dest.y then
                            curr_dist <- curr_dist + 2
                    else if curr.x = teleport_list[1].position.x and dest.x =
                        teleport_list[0].position.x then
                        curr_dist <- curr_dist + 3
                        if (dest.x - 1) < 0 and teleport_list[1].position.y = dest.y then
                            curr_dist <- curr_dist + 2

# count the distance between two points by using teleports
for i in range(0, len(teleport_list)):
    dist += count_pos_dist(curr, teleport_list[i].position) +
    count_pos_dist(dest, teleport_list[-i].position)
    # if current distance is lesser than the previously computed distance,
    choose current distance as the minimum distance

```

```

# while changing the current target
if dist < curr_dist then
    use_portal <- true
    curr_dist <- dist
    curr_target <- teleport_list[i].position
-> curr_dist, curr_target, use_portal

# TODO: randomize delta_x and delta_y for determining next position
# this function is used when the delta_x != 0 and delta_y != 0 to give
random move effect
function randomize_position(delta_x: int, delta_y: int) -> int, int
    random_number <- random(0, 1)
    if random_number <- 0 then
        -> 0, delta_y
    else
        -> delta_x, 0

# TODO: get next move position if bot want to go from the current position
to destination position
# if bot want to avoid teleports, set avoid to true. otherwise, set to
false
function get_direction_position(curr: Position, dest: Position,
teleport_list: arrayofGameObject, avoid: bool, width: int, height: int) ->
int, int
    # get initial delta_x and delta_y by using clamp function provided in
    util.py
    delta_x <- clamp(dest.x - curr.x, -1, 1)
    delta_y <- clamp(dest.y - curr.y, -1, 1)
    # get teleport position
    tell1_pos <- teleport_list[0].position
    tell2_pos <- teleport_list[1].position
    # if current position is equal to destination position, use random move
    # while this case rarely (or hardly ever) appears, it still important to
    prevent error by not sending (0,0) to the server
    if delta_x = 0 and delta_y = 0 then
        -> get_random_move(curr, width, height)

    # if delta_y = 0, delta_x != 0, and avoid is true,
    # check if the next teleport is located in the next position
    # if it's true, then go to the available y-axis first
    if delta_y = 0 and avoid then
        if (tell1_pos.y = curr.y and tell1_pos.x = (curr.x + delta_x)) or
        (tel2_pos.y = curr.y and tel2_pos.x = (curr.x + delta_x)) then
            if (curr.y + 1) < height then
                delta_x <- 0
                delta_y <- 1
            else
                delta_x <- 0
                delta_y <- -1
        # if delta_x = 0, delta_y != 0, and avoid is true,
        # check if the next teleport is located in the next position
        # if it's true, then go to the available x-axis first
        else if delta_x = 0 and avoid then
            if (tell1_pos.x = curr.x and tell1_pos.y = (curr.y + delta_y)) or
            (tel2_pos.x = curr.x and tel2_pos.y = (curr.y + delta_y)) then
                if curr.x + 1 < width then
                    delta_y <- 0
                    delta_x <- 1
                else
                    delta_y <- 0

```

```

        delta_x <- -1
    else if delta_y != 0 and delta_x != 0 then
        # if delta_x != 0, delta_y != 0, and avoid is true
        if avoid then
            # check if bot's next position will be blocked by teleport and move
            # accordingly
            if (curr.x + delta_x) = tell1_pos.x and tell1_pos.x = dest.x and
                (curr.y < tell1_pos.y < dest.y or curr.y > tell1_pos.y > dest.y):
                delta_x <- 0
            else if (curr.x + delta_x) = tel2_pos.x and tel2_pos.x = dest.x and
                (curr.y < tel2_pos.y < dest.y or curr.y > tel2_pos.y > dest.y) then
                delta_x <- 0
            else if (curr.y + delta_y) = tell1_pos.y and tell1_pos.y = dest.y and
                (curr.x < tell1_pos.x < dest.x or curr.x > tell1_pos.x > dest.x) then
                delta_y <- 0
            else if (curr.y + delta_y) = tel2_pos.y and tel2_pos.y = dest.y and
                (curr.x < tel2_pos.x < dest.x or curr.x > tel2_pos.x > dest.x) then
                delta_y <- 0
            else if (curr.x + delta_x) = tell1_pos.x and curr.y = tell1_pos.y then
                delta_x <- 0
            else if (curr.x + delta_x) = tel2_pos.x and curr.y = tel2_pos.y then
                delta_x <- 0
            else if curr.x = tell1_pos.x and (curr.y + delta_y) = tell1_pos.y then
                delta_y <- 0
            else if curr.x = tel2_pos.x and (curr.y + delta_y) = tel2_pos.y then
                delta_y <- 0
            else
                # give random effect to bot's movement
                delta_x, delta_y <- randomize_position(delta_x, delta_y)
        else
            # give random effect to bot's movement
            delta_x, delta_y <- randomize_position(delta_x, delta_y)
    -> delta_x, delta_y

# TODO: get shortest distance from a certain diamond to its base and
corresponding diamond
# this function will return list of list of integer, namely arr[N][3] where
n is the amount of diamonds
# 1. arr[i][0] is the distance from ith diamond to jth diamond where jth
diamond is the nearest diamond from ith diamond
# 2. arr[i][1] is the minimum distance from ith diamond to its base
# 3. arr[i][2] is the corresponding diamond's index from arr[i][0], namely
the index-j from the explanation above
function get_all_diamonds_dist(diamonds: arrayofGameObject, teleports:
arrayofGameObject, base: Position, width: int, height: int) ->
arrayof(arrayofInt)
n: int <- len(diamonds)
if n = 0 then
    -> []
res <- [[MAX_INT, MAX_INT, 0] i traversal [0..n - 1]]
i traversal [0..n - 1]
    # for ith diamond, get the minimum distance to base and assign the
    # value to res[i][1]
    base_dist, _, _ <- get_minimum_dist(diamonds[i].position, base,
teleports, width, height)
    res[i][1] <- base_dist
j traversal [i + 1..n - 1]
    dist, _, _ <- get_minimum_dist(diamonds[i].position,
diamonds[j].position, teleports, width, height)
    # if current distance is lesser than the previously computed distance
    # from ith diamond,

```

```

# choose current distance as the minimum distance and assign j to
res[i][2]
if dist < res[i][0] then
  res[i][0] <- dist
  res[i][2] <- j
# do the same thing for jth diamond
if dist < res[j][0] then
  res[j][0] <- dist
  res[j][2] <- i
-> res

# TODO: check if two positions are equal
function is_position_equal(p1: Position, p2: Position) -> bool
-> p1.x = p2.x and p1.y = p2.y

# TODO: check if a point is in the area between two points
function is_position_in_area(base: Position, dest: Position, query_obj: Position) -> bool
  if base.x = query_obj.x and base.y = query_obj.y then
    -> false
  check_x <- base.x <= query_obj.x <= dest.x or base.x >= query_obj.x >=
  dest.x
  check_y <- base.y <= query_obj.y <= dest.y or base.y >= query_obj.y >=
  dest.y
  -> check_x and check_y
# TODO: pick optimal diamond
# this function will return the optimal diamond target's position and a
boolean use_portal
# if the optimal distance is achieved by stepping through the teleports,
the target position will be corresponding teleport's position, use_portal
will be true
# otherwise, the target position will be the optimal diamond's position and
use_portal will be false
function pick_optimal_diamond(curr_pos: Position, diamond_list:
arrayofGameObject, teleport_list: arrayofGameObject, diamond_dist_list:
arrayof(arrayofInt), num_of_diamond_needed: int, width: u, height: int) ->
Position, bool
  min_score <- MAX_INT
  min_base_dist <- MAX_INT # used when number of diamond needed is > 2
  is_portal_used <- false
  selected_diamond <- None
  diamond_len <- len(diamond_list)

  i traversal [0..diamond_len - 1]
  # skip if ith diamond's point is greater than number of diamond needed
  if diamond_list[i].properties.points > num_of_diamond_needed then
    continue
  # get distance from current position to ith diamonds
  curr_dist, curr_target, curr_is_portal_used <-
  get_minimum_dist(curr_pos, diamond_list[i].position, teleport_list,
width, height)
  # calculate current score based on number of diamond needed
  d_needed <- num_of_diamond_needed - diamond_list[i].properties.points
  base_dist <- diamond_dist_list[i][1]
  # if the number of diamond needed is 1,
  # current score is the distance from current position to ith diamond
  plus the distance from ith diamond the the base
  if num_of_diamond_needed = 1 then
    curr_score <- curr_dist + base_dist
  else if num_of_diamond_needed = 2 then

```

```

# if the number of needed diamond is 2 and ith diamond's point is
# equal to 1,
# current score is the distance from current position to ith diamond
# plus the distance from ith diamond to jth diamond (where jth diamond
# is the nearest diamond from i)
# plus the distance from jth diamond to the base
if d_needed != 0 and diamond_len > 1 then
    curr_score <- curr_dist + diamond_dist_list[i][0] +
    diamond_dist_list[diamond_dist_list[i][2]][1]
# if the number of needed diamond is 2 and ith diamond's point is
# equal to 2 or there is no remaning diamond (diamond_len == 1),
# current score is the distance from current position to ith diamond
# plus the distance from ith diamond the the base
else
    curr_score <- curr_dist + base_dist
else
    # if the number of diamond needed is 1 and ith diamond point is equal
    # to 1,
    # current score is the distance from current position to the ith
    # diamond's position plus the distance between ith diamond and jth
    # diamond (where jth diamond is the nearest diamond from i)
    # otherwise, the score is just the distance from current position to
    # the ith diamond
    curr_score <- curr_dist
    if diamond_list[i].properties.points = 1 and diamond_len > 1 then
        curr_score <- curr_dist + diamond_dist_list[i][0]
    # if current score is lesser than minimum score
    # or current score is equal to minimum score and ith diamond distance
    # to base is lesser than minimum base distance (case when number of
    # diamond needed is 2)
    # set current score as the minimum score
    if curr_score < min_score or (curr_score = min_score and base_dist <-
    min_base_dist and num_of_diamond_needed > 2) then
        min_score <- curr_score
        selected_diamond <- curr_target
        is_portal_used <- curr_is_portal_used
        min_base_dist <- base_dist
    -> selected_diamond, is_portal_used

# TODO: get random move from current position
function get_random_move(curr_pos: Position, width: int, height: int) ->
int, int
    if (curr_pos.x + 1) < width then
        -> 1, 0
    else if (curr_pos.x - 1) >= 0 then
        -> -1, 0
    else if (curr_pos.y + 1) < height then
        -> 0, 1
    else
        -> 0, -1

# TODO: take the nearest diamond with some considerations
class MeowNearestDiamond(BaseLogic)
    function __init__(self) -> None:
        self.directions <- [(1, 0), (0, 1), (-1, 0), (0, -1)]
        self.goal_position <- None
        self.current_direction <- 0

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

```

```

# initialize variables needed
curr_pos <- board_bot.position
curr_points <- props.diamonds
width <- board.width
height <- board.height
base_pos <- board_bot.properties.base
remaining_time <- props.milliseconds_left // 1000
inventory_size <- props.inventory_size

# initialize object lists
teleport_list <- []
diamond_list <- []
red_button <- None

# get all object list
obj_traversal board.game_objects
  if obj.type = "TeleportGameObject" then
    teleport_list.append(obj)
  else if obj.type = "DiamondGameObject" then
    diamond_list.append(obj)
  else if obj.type = "DiamondButtonGameObject" then
    red_button <- obj

# get list of shortest distance from a certain diamond to its base and
corresponding diamond (read comments from line 230 - 234)
diamond_dist_list <- get_all_diamonds_dist(diamond_list, teleport_list,
base_pos, width, height)
# get distance from current position to base
base_dist, base_target, base_use_portal <- get_minimum_dist(curr_pos,
base_pos, teleport_list, width, height)
# get optimal diamond (target) position
diamond_picked, is_portal_used <- pick_optimal_diamond(curr_pos,
diamond_list, teleport_list, diamond_dist_list, inventory_size -
curr_points, width, height)
# if remaining time is lesser than the distance from current position
to base (by the assumption of 1 movement/second)
if remaining_time < (base_dist + 2) then
  # if current points is not equal to zero, go to the base
  if (curr_points ≠ 0) then
    # if the optimal diamond's position is in the area between current
    position and base, traverse through the optimal diamond
    if (0 < curr_points < inventory_size and diamond_picked ≠ None and
not is_portal_used and is_position_in_area(curr_pos, base_target,
diamond_picked)) then
      -> get_direction_position(curr_pos, diamond_picked,
      teleport_list, True, width, height)
    # if the red button's position is in the area between current
    position and base, traverse through the red button
    else if (is_position_in_area(curr_pos, base_target,
red_button.position)) then
      -> get_direction_position(curr_pos, red_button.position,
      teleport_list, True, width, height)
    else
      -> get_direction_position(curr_pos, base_target, teleport_list,
      not base_use_portal, width, height)
  # if current points is equal to zero, go to optimal diamond's
  position if possible
  else
    -> get_direction_position(curr_pos, diamond_picked, teleport_list,
    not is_portal_used, width, height)
else

```

```

if (curr_points < inventory_size) then
    # if there's optimal diamond
    if diamond_picked ≠ None then
        # traverse through base if the base' position is in the area
        # between current position and optimal diamond's position
        if is_position_in_area(curr_pos, diamond_picked, base_pos) then
            -> get_direction_position(curr_pos, base_pos, teleport_list,
                true, width, height)
        # traverse through red button if the red button's position is in
        # the area between current position and optimal diamond's position
        else if is_position_in_area(curr_pos, diamond_picked,
            red_button.position) then
            -> get_direction_position(curr_pos, red_button.position,
                teleport_list, true, width, height)
        else
            -> get_direction_position(curr_pos, diamond_picked,
                teleport_list, not is_portal_used, width, height)
        # go to base if there is no optimal diamond
    else
        -> get_direction_position(curr_pos, base_target, teleport_list,
            not base_use_portal, width, height)
    # if inventory is full, go to the base
else
    -> get_direction_position(curr_pos, base_target, teleport_list, not
        base_use_portal, width, height)

```

## 4.2. Struktur Data Bot

Berikut adalah daftar struktur data, baik yang primitif maupun non primitif, yang digunakan dalam program ini.

### 1. Integer

*Integer* (bilangan bulat) digunakan untuk menyimpan informasi seperti posisi sumbu-x, sumbu-y, jarak *object*, skor *diamond* saat ini, tinggi dan lebar papan permainan, dll.

### 2. Boolean

*Boolean* digunakan pada variabel yang menyimpan informasi apakah mendapatkan jarak minimum ke suatu *object* membutuhkan *teleporter* atau tidak.

### 3. String

*String* digunakan pada variabel yang menyimpan tipe dari suatu objek.

### 4. Tuple

*Tuple* digunakan pada *return type* fungsi pada program ini, seperti *get\_minimum\_dist*, *randomize\_position*, *get\_direction\_position*, dan *pick\_optimal\_diamond*.

### 5. Class Position

*Class Position* menyimpan informasi posisi suatu *object* pada sumbu-x dan sumbu-y.

### 6. Class GameObject

*Class GameObject* menyimpan informasi posisi suatu *object*, tipe *object* tersebut, dan (class) *properties* dari *object* tersebut.

### 7. Class Board

*Class Board* menyimpan informasi mengenai papan permainan, seperti *width*, *height*, list *GameObject*, dan lain-lain.

8. *Class Properties*

Class ini menyimpan informasi *properties* suatu object. Class ini menyimpan poin *object* saat ini, skor saat ini, *diamonds* saat ini, ukuran *inventory*, name *object*, dan lain-lain.

9. *List of GameObject*

*List* ini terdapat pada *Class Board* untuk menyimpan *list* dari objek-objek yang terdapat pada suatu papan permainan.

10. *List of List of integer*

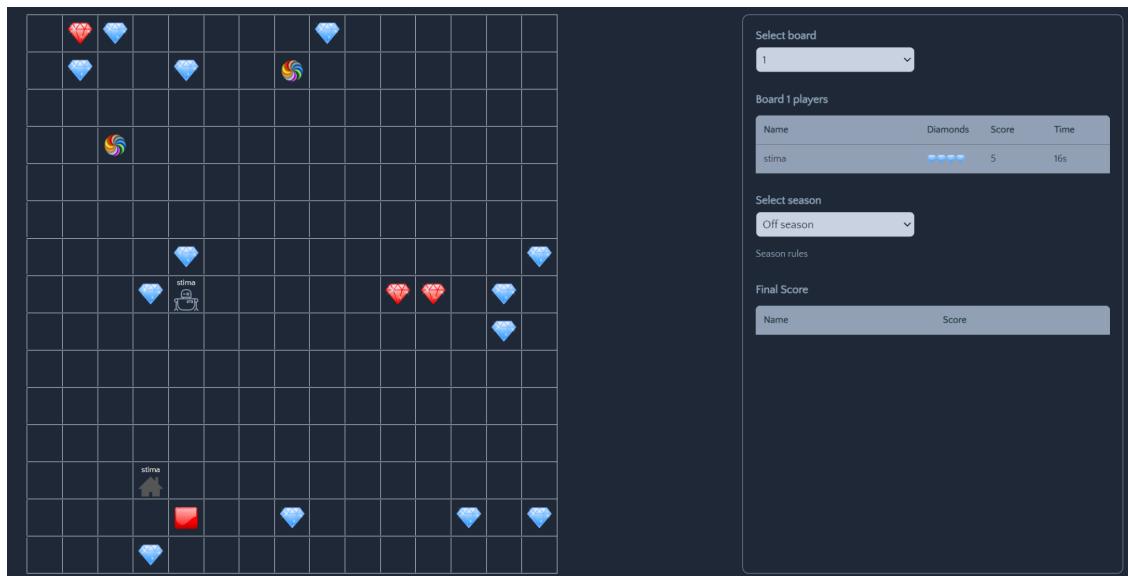
*List* ini digunakan untuk menyimpan informasi jarak *diamond* terdekat relatif terhadap *diamond* ke-*i*, jarak antara *base* dan *diamond* ke-*i*, dan indeks dari *diamond* terdekat relatif terhadap *diamond* ke-*i*. Untuk lebih jelasnya, berikut adalah penjelasannya

- a. *List[i][0]* -> jarak dari *diamond* ke-*i* ke *diamond* ke-*j* dimana *diamond* ke-*j* adalah *diamond* terdekat dari *diamond* ke-*i*.
- b. *List[i][1]* -> jarak dari *diamond* ke-*i* ke *base*
- c. *List[i][2]* -> indeks *j* pada penjelasan bagian (a)

### 4.3. Analisis Desain Solusi Algoritma Greedy

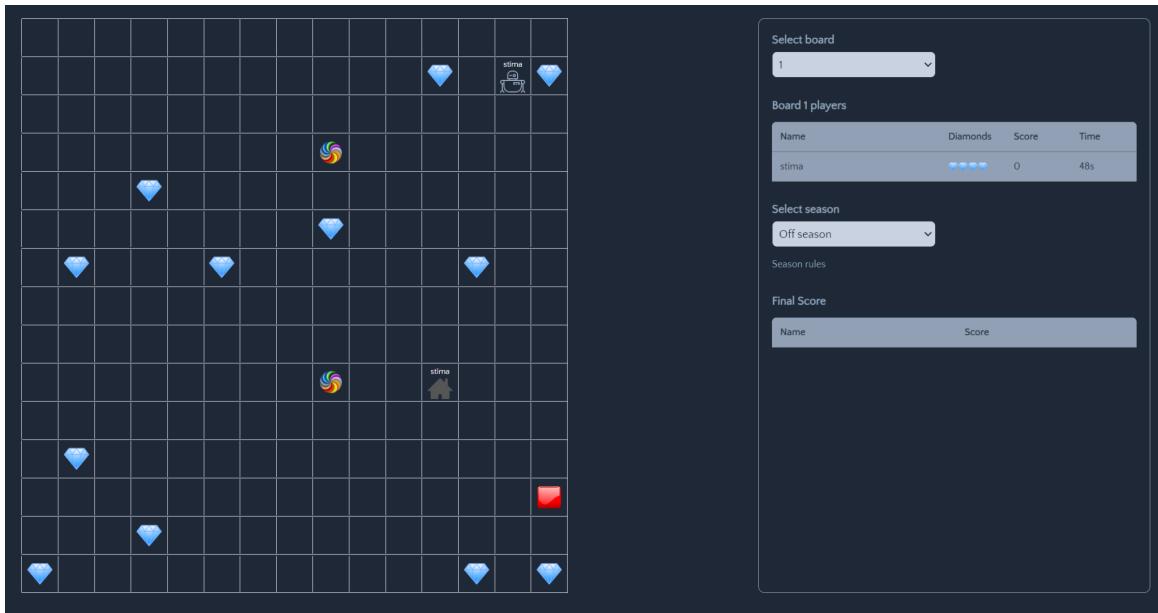
Setelah melakukan analisis dan pengujian secara berulang kali, kami mengamati beberapa perilaku *bot* kami pada berbagai kasus. Adapun kasus-kasus unik yang kamu ujikan adalah sebagai berikut.

1. Kasus 1



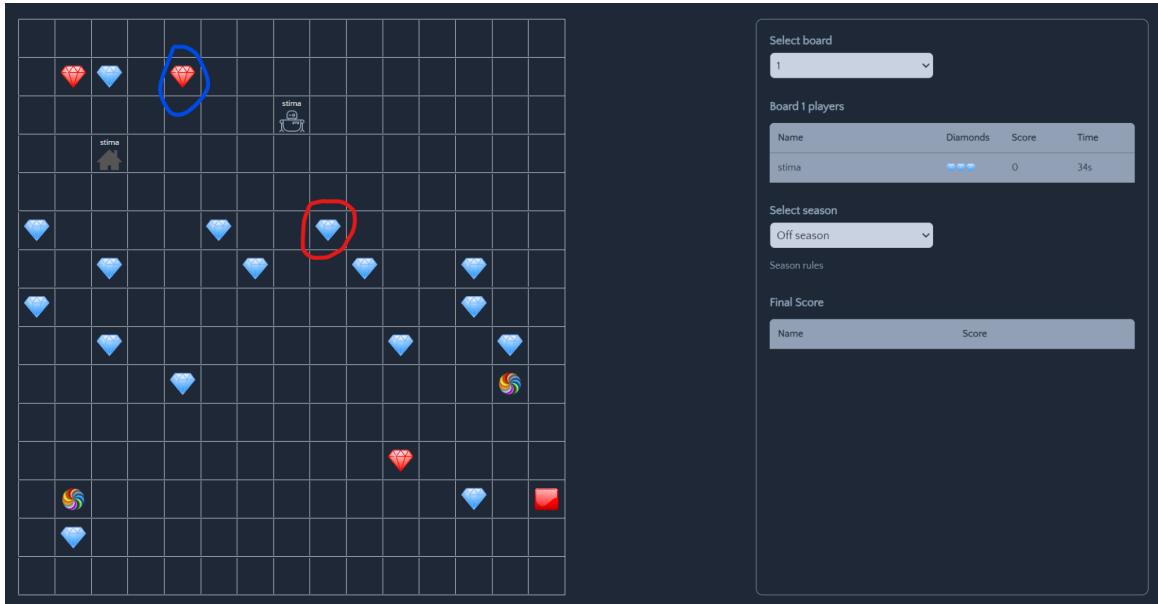
Pada kasus tersebut, *bot* memiliki pilihan untuk mengambil *diamond* di atas atau mengambil *diamond* di kiri karena keduanya memiliki jarak yang sama relatif terhadap posisi *bot* saat ini. Namun, karena *diamond* di sebelah kiri memiliki jarak ke *base* yang lebih kecil dibandingkan dengan *diamond* di sebelah kanan, *bot* memutuskan untuk bergerak ke kiri sesuai dengan strategi *dynamic shortest path*.

## 2. Kasus 2



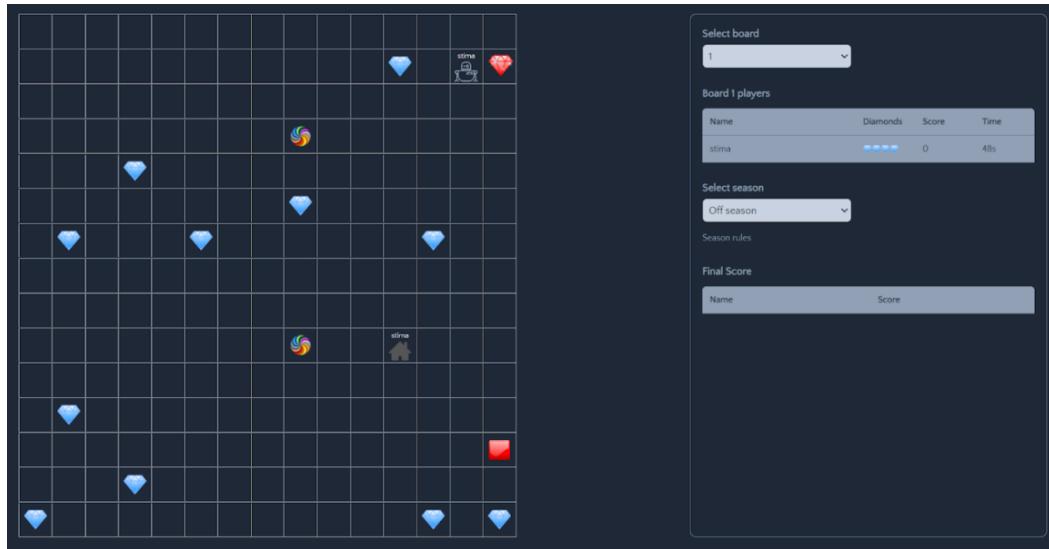
Pada kasus ini, *diamond* di sisi sebelah kanan memiliki jarak relatif yang lebih pendek dibandingkan dengan *diamond* di sebelah kiri bot. Namun, karena *diamond* yang dibutuhkan tinggal satu, jarak *diamond* terhadap *base* juga ikut diperhitungkan sehingga target yang dipilih adalah *diamond* di sebelah kiri.

## 3. Kasus 3



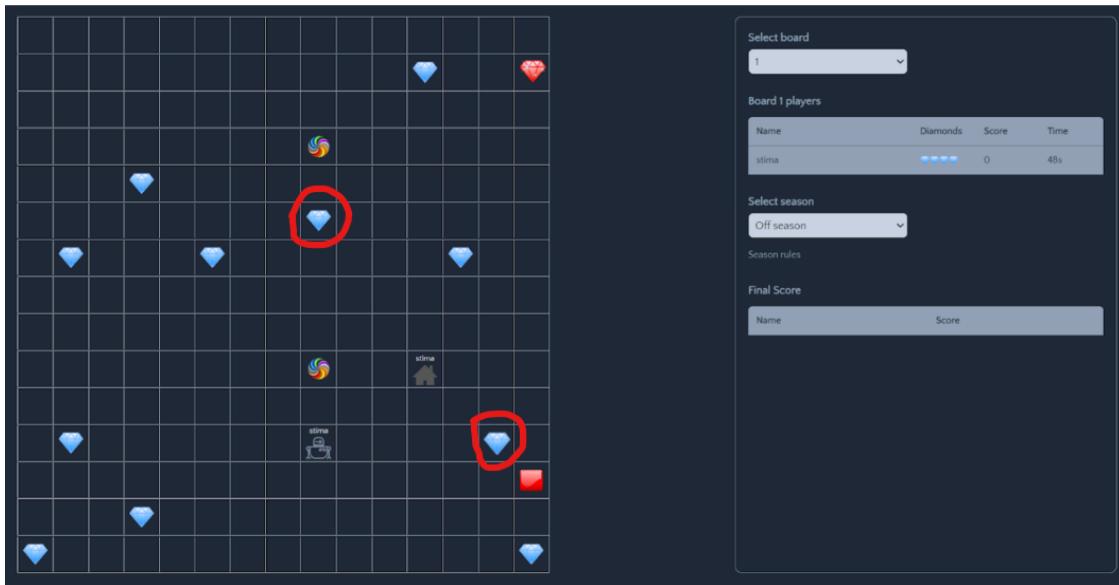
Pada kasus ini, terdapat dua buah *diamond* dengan jarak yang sama. Algoritma *dynamic shortest path* akan memilih *diamond* merah dengan dua alasan. Pertama, *diamond* merah memiliki dua poin, dibandingkan dengan *diamond* biru yang hanya memiliki satu poin. Kedua, jarak *diamond* merah terhadap *base* lebih dekat dibandingkan dengan jarak *diamond* biru terhadap *base*.

#### 4. Kasus 4



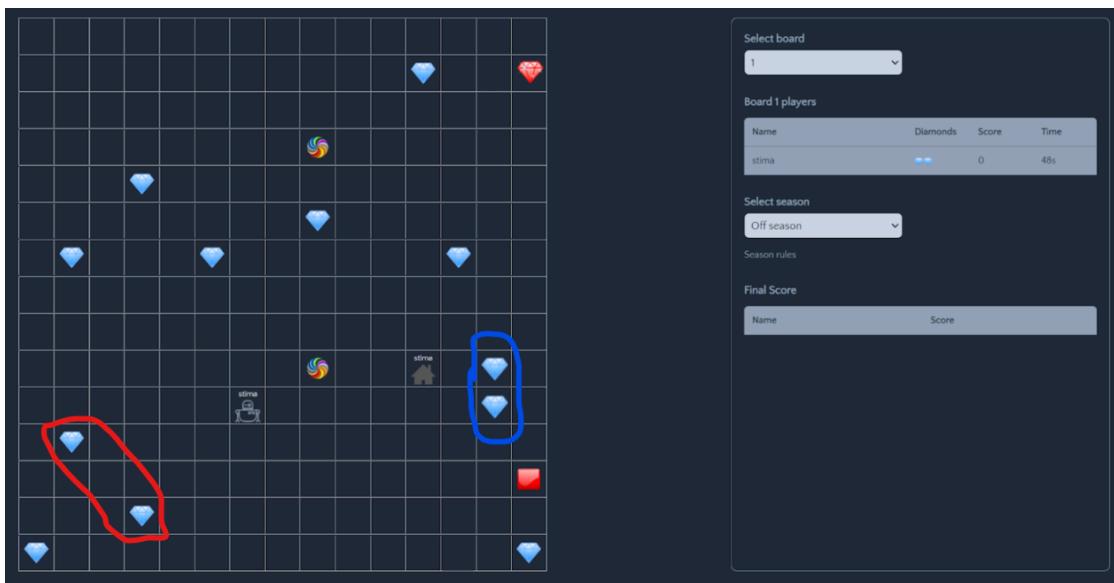
Pada kasus ini, *diamond* terdekat dengan *bot* adalah sebuah *diamond* merah dengan dua poin. Namun, karena *inventory bot* terdapat empat *diamond* berjumlah empat poin dan hanya membutuhkan satu poin lagi, *bot* akan mencari *diamond* biru terdekat, yaitu *diamond* biru di sebelah kirinya.

## 5. Kasus 5



Pada kasus ini, *diamond* terdekat secara langsung dari *bot* ialah *diamond* yang dilingkari merah pada kanan bawah *board*. Namun demikian, jarak tempuh *bot* lebih dekat jika menggunakan *teleporter* untuk berpindah dari *teleporter* bawah ke *teleporter* atas dan mengambil *diamond* yang dilingkari merah pada tengah *board*. Untuk mengambil *diamond* di kanan bawah *board*, *bot* harus menempuh 5 langkah. Untuk mengambil *diamond* di tengah *board*, *bot* hanya harus menempuh 4 langkah. Algoritma *dynamic shortest distance* akan memilih *diamond* dengan jarak tempuh terdekat, yaitu *diamond* di tengah *board*.

## 6. Kasus 6



Pada kasus ini, *diamond* terdekat dari *bot* adalah kedua *diamond* di lingkaran merah (keduanya berjarak sama dari *bot*). Meski demikian, algoritma *dynamic shortest path* akan mencari *diamond* dengan total jarak dari *bot* dan jarak dengan *diamond* terdekat lainnya yang paling kecil. Untuk mengambil kedua *diamond* di lingkaran merah, *bot* harus bergerak 10 langkah. Sementara itu, *bot* hanya perlu mengambil 8 langkah untuk mengambil kedua *diamond* di lingkaran biru. Dengan demikian, *diamond* yang diambil ialah *diamond* bawah pada lingkaran biru, meski bukan merupakan *diamond* dengan jarak terdekat dari *bot*.

## Bab 5: Kesimpulan dan Saran

### 5.1. Kesimpulan

Dari hasil eksplorasi alternatif solusi terkait berbagai strategi algoritma *greedy* yang bisa kami implementasikan untuk menyelesaikan permasalahan dalam permainan Diamonds, kami dapat menentukan strategi yang paling optimal untuk permainan tersebut. Meski demikian, algoritma kami tentu tidak sempurna, masih terdapat celah dan kasus-kasus di mana *bot* kami tidak berhasil memenangkan pertandingan dalam permainan Diamonds.

Dari hasil pengujian, hal tersebut dapat diatribusikan ke faktor-faktor seperti posisi *bot* lawan dan sifat alamiah dari heuristik *greedy* itu sendiri di mana algoritma tidak akan selalu bisa menghasilkan solusi optimal global, tetapi dijamin menghasilkan langkah optimal lokal.

Secara umum, bisa dikatakan strategi dan algoritma *greedy* yang kami implementasikan merupakan algoritma yang cukup baik. Di lihat dari pengujian yang kami lakukan, di mana *bot* kami mampu mengumpulkan poin yang cukup banyak dalam waktu yang ditentukan dan sering memenangkan pertandingan saat diadu dengan *bot* bawaan permainan Diamonds. Meski demikian, tentu terdapat kekurangan-kekurangan dari strategi dan algoritma *greedy* yang perlu diperhatikan dan mungkin bisa diperbaiki di kemudian hari.

### 5.2. Saran

Dengan dibuatnya laporan tugas besar ini, kelompok kami berharap adanya pendalaman lebih lanjut terkait dengan strategi algoritma *greedy* yang paling efektif dan efisien dalam menyelesaikan permasalahan dalam permainan Diamonds. Hal ini tentunya tidak lain dan tidak bukan untuk kemajuan keilmuan informatika di kemudian hari. Kami juga berharap penggerjaan tugas besar ini dapat mengasah kemampuan kami dan menambah minat kami untuk terus mendalami bidang ilmu Informatika di kemudian hari.

Tugas Kecil 1 IF2123  
Pemanfaatan Algoritma *Greedy* dalam Pembuatan *Bot* Permainan Diamonds  
Kelompok Arctica Kenapa Batal?

## Lampiran

*Link Repository GitHub*

[https://github.com/DeltDev/Tubes1\\_Arctica-Kenapa-Batal](https://github.com/DeltDev/Tubes1_Arctica-Kenapa-Batal)

*Link Video*

<https://www.youtube.com/watch?v=zh1mFJI-suo>

## Daftar Pustaka

[https://en.wikipedia.org/wiki/Greedy\\_algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm)

<https://www.freecodecamp.org/news/greedy-algorithms/>

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