

**Tugas Besar 1 IF2211 Strategi Algoritma
Semester II tahun 2023/2024**
**Pemanfaatan Algoritma Greedy dalam Pembuatan
Bot Permainan Diamonds**



Kelompok Tuan Krabs:

Keanu Amadius Gonza Wrahatno 13522082

Muhammad Atpur Rafif 13522086

Muhamad Raflri Rasyiidin 13522088

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023 / 2024**

Daftar Isi

Daftar Isi.....	2
BAB I.....	3
Deskripsi Tugas.....	3
BAB II.....	5
Landasan Teori.....	5
Cara Kerja Program Secara Umum.....	5
BAB III.....	7
Pemetaan Greedy.....	7
Aplikasi Strategi.....	7
1. Jarak Terdekat.....	7
2. Area dengan Diamond Terbanyak.....	8
3. Gradien Terbesar.....	8
Klasifikasi Greedy dan Brute Force.....	9
Analisis Efisiensi dan Efektivitas.....	9
Strategi Greedy Terpilih.....	10
BAB IV.....	11
Implementasi.....	11
Pseudocode.....	12
Pengujian.....	15
BAB V.....	17
Kesimpulan dan Saran.....	17
Lampiran.....	18
Daftar Pustaka.....	18

BAB I

Deskripsi Tugas

Diamonds merupakan suatu *programming challenge* yang mempertandingkan antar bot-bot yang telah dibuat oleh pemain. Setiap pemain akan memiliki sebuah bot dimana tujuan dari bot ini adalah mengumpulkan *diamond* sebanyak-banyaknya. 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. 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

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 *base*.
5. Apabila bot menuju ke posisi home *base*, score bot akan bertambah senilai *diamond* yang tersimpan pada *inventory* dan *inventory* bot akan menjadi kosong kembali.
6. Usahakan agar bot anda tidak bertemu dengan bot lawan. Jika bot A menimpa posisi bot B, bot B akan dikirim ke *home base* dan semua *diamond* pada *inventory* bot B akan hilang, diambil masuk ke *inventory* bot A (istilahnya *tackle*).
7. Selain itu, terdapat beberapa fitur tambahan seperti *teleporter* dan *red button* yang dapat digunakan apabila anda menuju posisi objek tersebut.
8. Apabila waktu seluruh bot telah berakhir, maka permainan berakhir. *Score* masing-masing pemain akan ditampilkan pada tabel *Final Score* di sisi kanan layar.

BAB II

Landasan Teori

Algoritma Greedy adalah algoritma yang digunakan untuk memecahkan permasalahan secara langkah per langkah sedemikian sehingga pada setiap langkahnya memilih pilihan yang terbaik yang dapat diperoleh saat itu tanpa memperhatikan konsekuensi kedepannya dengan harapan pilihan tersebut akan mengantarkan kita pada solusi optimal global. Pada algoritma greedy, setiap pilihan yang telah diambil tidak dapat diubah pada langkah selanjutnya sehingga ketika kita menyadari bahwa pilihan yang diambil saat ini tidak akan mengantarkan kita pada solusi optimal global, kita tidak bisa mundur ke langkah-langkah sebelumnya dan harus tetap melanjutkannya hingga akhir. Pada permainan ini, Elemen-elemen algoritma greedy adalah:

1. Himpunan kandidat : himpunan ini berisi kandidat yang akan dipilih pada setiap langkah
2. Himpunan solusi : himpunan ini berisi kandidat yang sudah dipilih dari himpunan kandidat
3. Fungsi solusi : menentukan apakah himpunan kandidat yang dipilih / himpunan solusi sudah memberikan solusi yang baik.
4. Fungsi seleksi : fungsi untuk memilih kandidat berdasarkan strategi greedy yang dibuat.
5. Fungsi kelayakan : fungsi yang memeriksa apakah kandidat yang terpilih dapat dimasukkan ke dalam himpunan solusi atau tidak.
6. Fungsi obyektif : memaksimumkan atau meminimumkan.

Cara Kerja Program Secara Umum

Pada permainan ini, akan ditandingkan beberapa bot untuk mengambil *diamond*. Bot akan bergerak dalam *board* untuk mendapatkan *diamond*. Bot dapat memasuki teleport dan keluar pada teleport di sisi lainnya. Bot juga dapat menekan *red button* untuk mereset objek yang ada pada *board*.

Cara mengambil *diamond* yaitu dengan mengarahkan bot ke posisi *diamond* maka *diamond* akan otomatis terambil dan masuk pada *inventory*. Apabila bot menekan *red button*, maka komponen komponen pada *board* akan diacak lagi kecuali posisi *base* dan bot. Apabila bot melewati teleport, maka bot akan pindah ke teleport lainnya.

Bot yang memiliki *diamond* terbanyak akan menang dalam permainan ini. Bot memiliki maks *inventory* yaitu 5. Jika *inventory* penuh, bot tidak bisa lagi mengambil *diamond* dan bot harus kembali ke *base* agar *diamond* dapat diubah menjadi point dan mengosongkan *inventory*.

Setelah sampai *base*, maka *inventory* akan menjadi 0 dan bot dapat mengambil *diamond* lagi. Perlu diwaspadai bahwa bot musuh dapat mencuri *diamond* kita apabila posisi bot musuh sama dengan posisi bot kita.

Algoritma greedy dipakai pada permainan ini untuk menentukan arah jalannya bot agar mendapatkan *diamond* terbanyak dalam 60 detik. Pengimplementasiannya dilakukan dengan membuat algoritma yang nantinya digunakan untuk mengubah posisi bot.

BAB III

Pemetaan *Greedy*

Seluruh strategi yang akan dibahas sebagian besar memiliki pemetaan yang sama terhadap pendekatan *greedy*. Berikut merupakan pemetaan secara garis besar:

Aplikasi Strategi

Terdapat banyak pendekatan untuk menyelesaikan masalah ini menggunakan metode *greedy*. Pada tugas besar ini akan dibahas tiga cara yang dapat digunakan. Pada bagian selanjutnya, strategi ini nantinya akan dirujuk dengan nomornya masing-masing. Nantinya akan dipilih salah satu dari ketiga cara tersebut:

- Himpunan Kandidat: Seluruh kemungkinan langkah yang dapat ditempuh, yang dimana setiap langkahnya, bot hanya dapat melangkah satu petak ke atas, bawah, kiri, dan kanan.
- Himpunan Solusi: Langkah yang ditempuh sehingga menghasilkan *score* paling banyak.
- Fungsi Solusi: Banyaknya *diamond* yang terambil
- Fungsi Seleksi: Sesuai strategi yang akan dijelaskan.
- Fungsi Kelayakan: Langkah yang akan diambil tidak melewati batas permainan
- Fungsi Objektif: Memaksimalkan *diamond* yang terambil, sehingga *score* paling banyak.

1. Jarak Terdekat

Pada strategi ini, kita membuat fungsi dimana bot akan bergerak ke *diamond* yang paling dekat dengan posisi bot saat itu. Pertama-tama bot akan mencari semua *diamond* yang dapat ia ambil (tidak lebih dari 5). Apabila *inventory* sudah 4, maka bot hanya akan mencari *diamond* biru. Lalu bot akan mencari *diamond* yang paling dekat sampai jumlah *diamond* pada *inventory* berjumlah 5 lalu bot akan kembali ke *base*. Jika ada 2 *diamond* dengan jarak yang sama, maka bot akan mendekati *diamond* yang diiterasi dahulu / diproses lebih dulu. Algoritma ini juga mempertimbangkan adanya teleport. Apabila ada *diamond* yang lebih dekat jika melewati teleport, maka bot akan masuk ke teleport dan mengambil *diamond* tersebut. Algoritma ini juga digunakan saat balik ke *base*. Apabila jarak balik ke *base* lebih dekat menggunakan teleport, bot akan masuk ke teleport tersebut. Bot juga mempertimbangkan adanya *red button*. Apabila tidak ada lagi *diamond* terdekat dan *red button* ada di dekat bot, maka bot akan menuju *red button* untuk mereset bot.

2. Area dengan *Diamond* Terbanyak

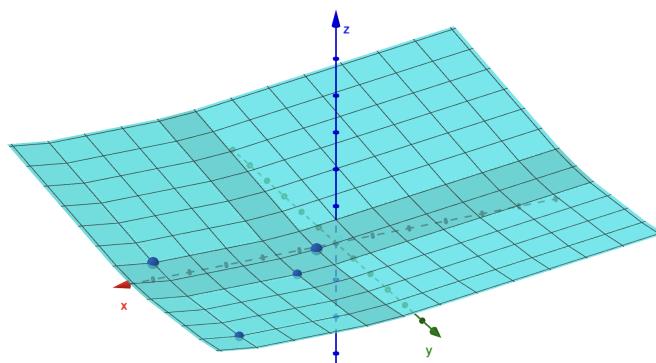
Pada strategi ini, kita akan melakukan pencarian area dengan jumlah *diamond* terbanyak dengan cara membuat area semu berukuran 4x4, menghitung jumlah *diamond* di area tersebut, dan melakukan iterasi ke seluruh area di papan. Setelah kita menemukan area dengan jumlah *diamond* terbanyak, kita akan mengarahkan bot ke area tersebut dan mengambil *diamond* yang berada di sana (pengambilan *diamond* akan menggunakan algoritma jarak terdekat ketika bot telah mencapai area yang ditargetkan). Bot hanya akan menghitung *diamond* yang dapat ditampung olehnya (poin yang dimiliki bot tidak akan melebihi batas maksimal ketika mengambil *diamond* tersebut). Area tujuan yang dipilih oleh bot adalah area dengan jumlah *diamond* terbanyak yang diiterasi terlebih dahulu (jika ada area dengan jumlah *diamond* sama, bot akan memilih area yang pertama kali ditemukan). Jika terdapat *teleporter* yang menuju area dengan jumlah *diamond* terbanyak di dekat posisi bot saat ini, bot akan memasuki *teleporter* tersebut. Jika jarak *red button* dari bot lebih dekat daripada jarak bot ke area dengan *diamond* terbanyak, maka bot akan bergerak ke arah *red button* tersebut. Pada algoritma ini, bot hanya akan kembali ke *base* jika total poin yang telah diperoleh sudah berjumlah 5.

3. Gradien Terbesar

Pada strategi ini, kita membuat sebuah fungsi dalam bentuk dua dimensi yang menerima posisi koordinat x dan y serta seluruh *state* atau nilai dari permainan yang berupa *Game Object* seperti *diamond* ataupun *player* lain. Misalkan, fungsi yang digunakan menghitung jumlah seluruh jarak menuju setiap *diamond*. Apabila kita membuat sebuah grafik pada koordinat 2D, akan terbentuk sebuah lembah. Lembah yang paling dalam memiliki arti sebuah koordinat dimana *diamond* akan terkumpul. Berikut merupakan gambaran grafik menggunakan fungsi sebelumnya (sumbu z diskalakan):

$$f(x, y) = \sum_{(i, j) \in \text{diamond}} (|x - i| + |y - j|)$$

$$\text{diamond} = \{(1, 1), (2, 2), (5, 0), (5, 5)\}$$



Domain dari sumbu x dan y adalah besarnya papan permainan secara diskrit. Selain itu, kita tidak perlu menghitung nilai fungsi pada seluruh papan, namun hanya perlu empat koordinat yang dapat dituju oleh bot saat ini, ditambah satu posisi koordinat bot. Dengan kata lain, kita hanya perlu menghitung nilai fungsi pada posisi bot saat ini, atas, bawah, kiri, dan kanan relatif terhadap bot. Kemudian kita membandingkan gradien berdasarkan nilai yang didapat. Kita mencari gradien negatif yang memiliki nilai absolut terbesar. Arah dari gradien terbesar akan membuat bot menuju posisi dimana nilai fungsi paling kecil, dalam hal ini adalah posisi *diamond*. Tidak hanya itu, semakin banyak *diamond* berkumpul pada sebuah wilayah, maka wilayah tersebut akan memiliki lembah lebih dalam. Implementasi *greedy* pada strategi ini terdapat pada pemilihan gradien terbesar pada setiap saat bot akan bergerak.

Menggunakan strategi ini, maka dengan mudah kita bisa menambahkan konsiderasi selain *diamond* dengan mudah. Misalkan kita ingin menambahkan konsiderasi untuk menghindari (atau bahkan mendekati) bot lain. Hal ini bisa dilakukan dengan memberikan nilai yang merupakan fungsi dari *state* bot lain menuju bilangan real, yang nantinya dijumlahkan pada fungsi utama nanti. Ketika menghindari, kita bisa memberikan nilai positif yang sebanding dengan jarak, sehingga area di dekat bot tersebut akan memiliki ketinggian lebih tinggi, dan dihindari oleh bot yang ingin menuju lembah paling dalam.

Kita tidak harus selalu membuat fungsi yang membuat *diamond* menjadi lembah, kita bisa melakukan kebalikannya, yaitu membuat *diamond* memiliki ketinggian yang tinggi, dan membuat bot memilih langkah yang menuju tempat yang paling tinggi. Kita bisa menuliskan fungsi ini sebagai $f(x, y)$. Lalu misalkan bot berada pada posisi x_0 dan y_0 . Kemudian kita akan mensubtitusi posisi atas, bawah, kiri, dan kanan [misal, pada posisi kanan $f(x_0 + 1, y_0)$]. Nantinya hasil yang memberikan nilai maksimal merupakan langkah yang akan diambil.

Klasifikasi Greedy dan Brute Force

Batas diantara *greedy* dan *brute force* bukanlah merupakan sebuah hal yang hitam dan putih. Misalkan pada strategi pertama, bisa diklasifikasikan menjadi *brute force*, dikarenakan kita memeriksa seluruh *diamond* pada papan permainan, dan mencari yang paling dekat. Namun pada satu sisi lainnya, kita bisa menyebut ini sebagai *greedy*, karena kita hanya memilih kemungkinan yang paling dekat pada waktu itu juga. Hal yang sama juga bisa dikatakan untuk strategi nomor dua dan tiga.

Analisis Efisiensi dan Efektivitas

Pada strategi jarak terdekat, efisiensi sudah baik karena bot akan mencari *diamond* dengan jarak terpendek baik lewat teleport ataupun tidak. Mekanisme kembali ke *base*

dilakukan saat *diamond* pada *inventory* = 5 dan sudah mempertimbangkan juga jarak terdekat ke *base* baik lewat teleport ataupun tidak. Namun, algoritma ini belum optimal karena bot harus membawa 5 *diamond* terlebih dulu baru dia balik.. Algoritma ini juga memiliki kekurangan dalam beberapa kasus. Contoh kasus terburuknya, ada 1 *diamond* di selatan bot dengan jarak dekat namun di utaranya ada sekumpulan 5 *diamond* dan ada *base* dengan jarak yang jauh, maka bot akan mengarah ke selatan mengambil *diamond* terdekat di selatan. Hal ini tidaklah efisien karena membuang buang langkah yang seharusnya ia bisa langsung menuju ke utara mengambil *diamond* yang dekat dengan *base*.

Pada strategi area dengan jumlah *diamond* terbanyak, efisiensi sudah cukup baik karena bot mencari area dengan jumlah *diamond* terbanyak dan mengambil *diamond* di area tersebut sehingga bot memperoleh *diamond* yang banyak dalam sekali jalan. Namun, kekurangan dari algoritma ini adalah bot dapat mengubah target tujuan dan arah gerak secara tiba-tiba. Hal tersebut dikarenakan area dengan jumlah *diamond* terbanyak dapat berubah setiap saat sehingga bot akan memperbarui area tujuan dan mengganti arah pergerakan. Kasus terburuknya adalah bot tidak dapat memperoleh *diamond* karena terlalu sering mengganti target dan arah (bot akan bergerak bolak-balik).

Pada strategi gradien, bot akan menuju ke “titik dengan poin tinggi”. Poin tinggi terjadi apabila disana terdapat banyak *diamond*. Bot musuh juga akan menyebabkan poin menjadi rendah agar bot kami dapat menjauhi bot musuh. Strategi ini paling efisien dari 2 strategi sebelumnya. Kekurangan yang dimiliki strategi jarak terdekat sudah diselesaikan oleh strategi ini. Namun strategi ini tidak luput dari kekurangan juga. Strategi ini juga lemah terhadap red button. Karena apabila di reset maka bot yang awalnya menuju ke arah dengan konsentrasi *diamond* yang banyak menjadi hilang karena di reset. Namun strategi ini juga lebih baik dari strategi ke 2 karena jika di reset, bot akan mencari area dengan konsentrasi tinggi dan selama perjalanan jika ada *diamond* yang sangat dekat, maka akan ia ambil juga.

Strategi Greedy Terpilih

Dari semua kumpulan alternatif solusi, akhirnya kelompok kami memilih algoritma **gradien** karena efisiensi dan efektivitasnya. Algoritma ini memiliki efisiensi tinggi karena:

1. Bot akan bergerak ke area dengan konsentrasi *diamond* tertinggi yang jaraknya paling dekat dan memprioritaskan pengambilan *diamond* dengan poin tertinggi yang dapat ditampung oleh bot (*diamond* merah).
2. Selama perjalanan ke area dengan konsentrasi *diamond* banyak, jika ada *diamond* yang dekat maka bot juga akan mengambil *diamond* itu.
3. Bot akan cenderung kembali ke *base* saat waktu tersisa 10 detik sehingga bot akan mengamankan *diamond* yang ada di *inventory* daripada harus mengambil *diamond* dan tidak ada waktu untuk kembali ke *base*.
4. Bot memiliki mekanisme kembali ke *base* yang cukup baik sehingga dapat meminimalisasi kehilangan poin.

BAB IV

Implementasi

Program diimplementasikan menggunakan bahasa pemrograman python. Pada program bot, kami menggunakan struktur data *array* atau *list* untuk menampung posisi dari *diamond*, *teleporter*, dan *red button*. Pada pengujian, diperlukan sebuah *server* yang menjadi tuan rumah dari permainan. Pada *server*, digunakan sebuah template yang telah diberikan, dan akan dilampirkan. Peraturan *default*, permainan memiliki waktu sebanyak 60 detik untuk setiap sesi, dan seluruh bot/pemain memiliki waktu *cooldown* sebanyak 1 detik untuk setiap langkah.

Ketika melakukan pengujian, peraturan tersebut diubah. Lebih tepatnya waktu *cooldown* dipercepat menjadi 0.1 detik, sedangkan waktu permainan diperpanjang menjadi 600 detik. Hal ini sama seperti melakukan permainan sebanyak 100 sesi. Perlu diperhatikan bahwa tidak terdapat *reset* pada sesi pengetesan. Namun setiap *diamond* habis, atau tombol merah pada papan permainan ditekan, *diamond* akan dibuat kembali. Setiap hal tersebut terjadi, maka kita anggap satu sesi telah selesai (Meskipun hal ini berarti tombol merah tidak termasuk *gameplay* dari permainan). Berikut merupakan penamaan dari setiap implementasi:

1. Strategi 1: *nearest*
2. Strategi 2: *biggest*
3. Strategi 3: *gradient*
4. Strategi memilih langkah selanjutnya secara acak: *random* (sebagai basis)

Pada pengujian nantinya, kita akan memilih strategi 3 untuk dikumpulkan. Oleh karena itu, pada bagian ini akan dibahas lebih lanjut mengenai implementasi pada strategi tersebut. Berikut merupakan beberapa poin yang lebih spesifik mengenai implementasi strategi 3:

1. Fungsi ketinggian dari *diamond* yang memiliki poin 1 sebanding dengan $\frac{1}{r^2}$, dimana r adalah jarak antara titik x, y pada $f(x, y)$, sedangkan untuk *diamond* yang memiliki poin 2 akan memiliki fungsi $\frac{1}{r}$. Sebelum memasukkan *diamond* pada fungsi ini, akan diperiksa apakah *diamond* tersebut masih bisa diambil. Misalkan ketika memiliki *diamond* 4, maka kita akan meninggalkan *diamond* yang memiliki poin 2.
2. Titik yang memiliki objek *red button*, nilainya akan dikurangi sebanyak 1. Karena kita ingin menghindari mengulang penempatan *diamond* ketika posisi bot sudah dekat dengan tempat yang memiliki banyak *diamond*.
3. Apabila kita memiliki rasio *diamond* dan *inventory* yang kecil, maka bot akan berkontribusi terhadap fungsi sebanding dengan banyaknya *diamond* yang dimiliki oleh bot lain, namun berbanding terbalik kuadrat dengan jarak. Sebaliknya, apabila kita memiliki rasio yang besar, maka kita akan menghindari bot lain dengan mengurangi nilai fungsinya. Seluruh fungsi yang mengkonsiderasikan bot hanya berlaku pada jarak dua dari posisi saat ini.

4. Objek *base* dari bot memiliki kontribusi terhadap fungsi sama seperti *diamond* yang memiliki poin satu. Namun kita menambahkan sebuah berat berupa rasio *diamond* dan *inventory*, serta banyak waktu yang bersisi. Semakin banyak *diamond* yang dimiliki, maka bot akan cenderung menuju *base*. Lalu semakin sedikit waktu yang tersisa, bot juga akan mendekati *base* agar tidak kehilangan *diamond* yang dibawanya. Perhatikan bahwa cara pengimplementasian kontribusi *diamond* pada poin satu, membuat program tidak perlu memerintahkan bot untuk pergi ke *base* secara eksplisit. Hal ini dikarenakan seluruh *diamond* akan ditinggalkan karena *inventory* yang tidak cukup.
5. Pada objek *teleporter*, kita menggunakan seluruh fungsi pada poin 1 sampai 4. Misalkan terdapat *teleporter* A dan *teleporter* B. Kita bisa masuk ke A dan keluar di B. Perhitungan fungsi sama seperti sebelumnya, namun kita memberikan *offset* berupa jarak yang dibutuhkan untuk masuk ke *teleporter* A, dan nilai koordinat yang akan digunakan merupakan koordinat dari *teleporter* B. Hal yang sama dilakukan untuk B ke A. Nantinya nilai dari dua cara menggunakan *teleporter* dan langsung (dengan tidak menggunakan *teleporter*) akan dijumlahkan. Selain itu, apabila dua *teleporter* memiliki jarak kurang dari nilai tertentu akan membuat fungsi dan bot untuk tidak melewatinya.

Pseudocode

```

MAX_DIAMOND = 5

{ Fungsi untuk mencari jarak antara dua titik }
FUNCTION manhattan_distance(a: Position, b: Position):
    return |a.x - b.x| + |a.y - b.y|

{ Fungsi untuk menjumlahkan dua titik }
FUNCTION add_position(a: Position, b: Position):
    return Position(a.y + b.y, a.x + b.x)

CLASS GradientLogic(BaseLogic):
    bot: GameObject
    bot_pos: Position
    board: Board
    inventory_size: int

    PROCEDURE __init__(self):
        self.inventory_size = 0;
        self.directions: list[Position] = [
            Position(1, 0),
            Position(0, 1),
            Position(-1, 0),
            Position(0, -1)
        ]

    { Fungsi untuk mengatur pergerakan bot (tidak termasuk pergerakan
     terhadap teleporter) }
    FUNCTION base_fn(self, inp: Position, distance_offset: int):
        total = 0.0

```

```

        inventory_filled = float(self.bot.properties.diamonds) /
float(self.inventory_size)

        { Inisialisasi bot, diamond, dan red button pada papan }
        bots = self.board.bots
        diamonds = self.board.diamonds
        redButton_target: list[GameObject] = list(filter(lambda x: x.type ==
"DiamondButtonGameObject", self.board.game_objects))

        { Pengecekan diamond }
        for diamond in diamonds:
            pos = diamond.position
            points = diamond.properties.points
            if self.inventory_size - self.bot.properties.diamonds < points:
                continue

            distance = manhattan_distance(inp, pos) + distance_offset

            { Fungsi untuk mencari ketinggian titik lokasi diamond
            berada }
            if distance == 0:
                total += points * 100;
            elif points == 1:
                total += distance ** -2
            else:
                total += (distance ** -1) * points

        { Pencarian red button dan mengatur ketinggian agar red
        button selalu dihindari }
        for button in redButton_target:
            pos = button.position
            distance = manhattan_distance(inp, pos) + distance_offset
            if distance == 0:
                total -= 1;

        { Pencarian seluruh bot yang ada di papan }
        for bot in bots:
            pos = bot.position
            if pos == self.bot_pos:
                continue
            distance = max(manhattan_distance(inp, pos) + distance_offset, 1)

            { Jika jarak bot lain lebih dari 2, maka bot akan
            berjalan seperti biasa }
            if distance > 2:
                continue

            { Jika inventory telah terisi lebih dari 3/4 , maka
            ketinggian di sekitar bot lain akan dibuat rendah
            sehingga bot kami menghindarinya }
            if inventory_filled > 3/4:
                total -= (2 * inventory_filled) * (distance ** -2)

            { Jika inventory terisi kurang dari 1/4 , maka bot kami
            akan berusaha memakan bot lain jika memungkinkan }
            elif inventory_filled < 1/4:
                other_base_distance = manhattan_distance(bot.properties.base, pos)
                if other_base_distance < 8:

```

```

        other_inventory = float(bot.properties.diamonds) /
float(self.inventory_size)
            total += (other_inventory * 2) * (distance ** -2)

    { Algoritma untuk kembali ke base, termasuk algoritma untuk
        selalu kembali ke base ketika waktu yang tersisa kurang
        dari 10 detik }
    base = self.bot.properties.base
    distance = max(manhattan_distance(inp, base) + distance_offset, 0.1)
    time_weight = 1
    time_left = self.bot.properties.milliseconds_left / 1000
    if time_left < 10:
        time_weight = max((10 - time_left) * 3, time_weight)
    total += (2 * inventory_filled * time_weight) * (distance ** -2)

    return total

{ Fungsi untuk mengecek jarak bot ke target jika melalui teleporter }
FUNCTION fn(self, inp: Position):
    base_value = self.base_fn(inp, 0)
    teleporter: list[GameObject] = list(filter(lambda x: x.type ==
"TeleportGameObject", self.board.game_objects))
    tel0 = teleporter[0].position
    tel1 = teleporter[1].position
    if manhattan_distance(tel0, tel1) < 5:
        if position_equals(inp, tel0) or position_equals(inp, tel1):
            return 0
        return base_value

    tel0_value = self.base_fn(tel1, manhattan_distance(tel0, inp))
    tel1_value = self.base_fn(tel0, manhattan_distance(tel1, inp))
    return base_value + tel0_value + tel1_value

{ Fungsi untuk menentukan langkah selanjutnya }
FUNCTION next_move(self, board_bot: GameObject, board: Board):
    self.board = board
    self.bot = board_bot
    self.bot_pos = board_bot.position

    bot_provider = next(filter(lambda v: v.name == "BotProvider",
board.features))
    if hasattr(bot_provider, "config") and hasattr(bot_provider.config,
"inventory_size"):
        inv = bot_provider.config.inventory_size
        if type(inv) == int:
            self.inventory_size = inv

        possible_next_pos = list(
            filter(
                lambda pos: 0 <= pos.x and pos.x < board.width and 0 <= pos.y
and pos.y < board.height,
                    map(lambda dir: add_position(self.bot_pos, dir),
self.directions)
            )
        )

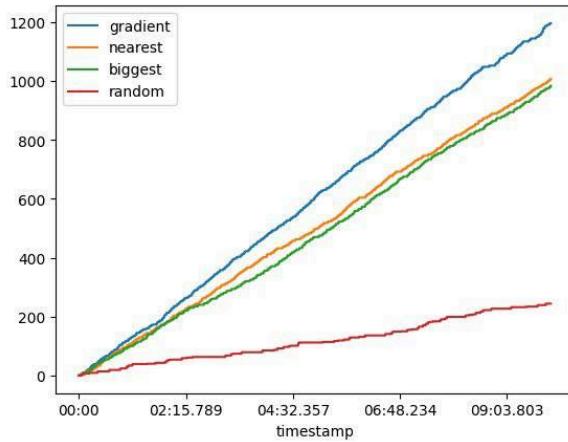
        next_pos = max(possible_next_pos, key=lambda pos: self.fn(pos))

```

```
return get_direction(self.bot_pos.x, self.bot_pos.y, next_pos.x, next_pos.y)
```

Pengujian

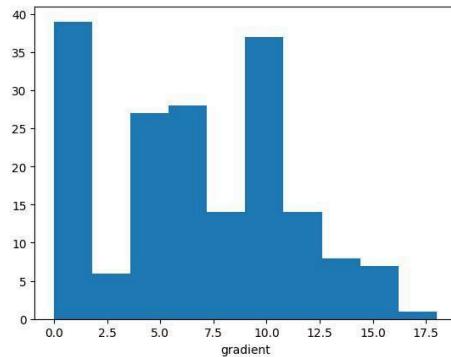
Kami telah melakukan pengujian dengan menggunakan 3 algoritma yang telah kami buat ditambah algoritma random dengan waktu 10 menit dan pergerakan bot delay 0,1 detik. Hasil yang kami dapatkan adalah sebagai berikut:



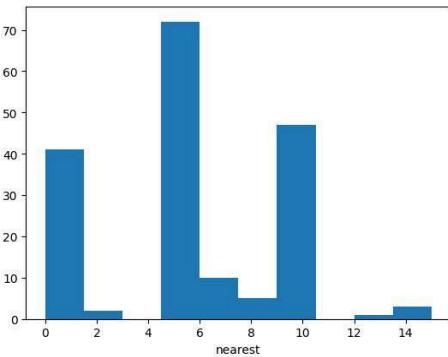
Gambar 3.1 Grafik banyaknya *diamond*

Dapat dilihat bahwa algoritma greedy gradien mengambil *diamond* terbanyak. Pada menit awal, banyaknya *diamond* cenderung sama namun makin lama *diamond* algoritma gradien makin banyak. Kami menggunakan waktu yang lama karena jika waktunya sedikit, pengetesan algoritma cenderung bergantung kepada keberuntungan.

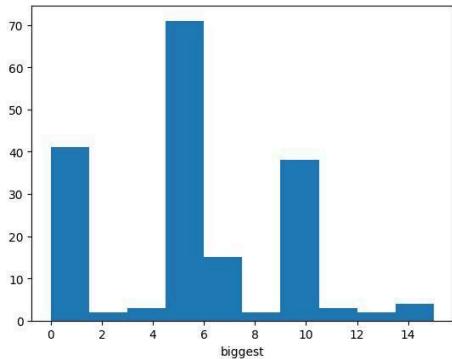
Kami juga melakukan pendataan terhadap banyaknya *diamond* yang diambil dalam 1 *round*. Satu *round* disini dihitung dari permainan dimulai sampai tombol merah ditekan. Jika tombol merah ditekan, *round* akan berakhir dan permainan akan masuk ke *round* berikutnya atau *round* ke 2. Hal ini terus dilang Sampai permainan berakhir. Berikut ini merupakan data dari tiap tiap algoritma mengenai seberapa banyak *diamond* yang diperoleh dalam 1 *round*.



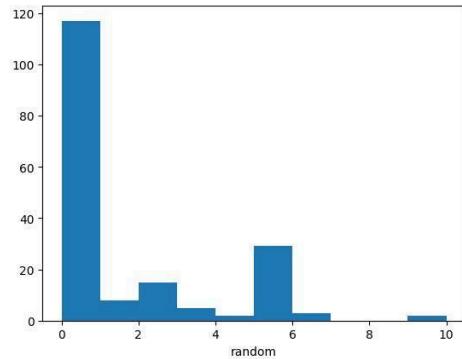
Gambar 3.2 *diamond* yang diambil gradien



Gambar 3.3 *diamond* yang diambil nearest



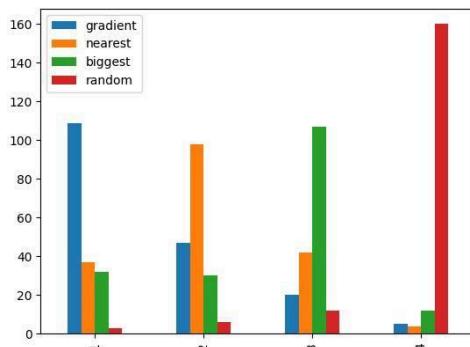
Gambar 3.4 *diamond* yang diambil *biggest*



Gambar 3.5 *diamond* yang diambil *random*

Dari data tersebut, pengambilan *diamond* dari algoritma gradien lebih tersebar karena ia dapat kembali ke *base* meskipun *diamond* yang diperoleh belum berjumlah 5. Namun, pada algoritma “jarak terdekat” dan “area terbanyak”, pengambilan *diamond* cenderung di kelipatan 5, yaitu 5 dan 10 karena mereka hanya akan kembali ke *base* jika *diamond* yang diperoleh sudah berjumlah 5.

Dan berikut ini merupakan data dari urutan peringkat bot dalam beberapa kali percobaan.



Gambar 3.6 Peringkat bot

Yang mendapatkan peringkat pertama paling banyak adalah algoritma gradien dan disusul oleh jarak terdekat lalu area terbanyak.

Pada pengujian ini kami juga menemukan kekurangan strategi gradien. Yang pertama jika posisi spawn *base* ada di samping (tidak di tengah) dan jauh dari mayoritas *diamond*. Maka bot harus menempuh jarak yang jauh dan memakan waktu. Kekurangan kedua yaitu apabila posisi posisi *diamond* di *reset*. Maka bot harus mencari lagi ke tempat yang *diamond*-nya banyak.

BAB V

Kesimpulan dan Saran

Dalam menyelesaikan permasalahan optimasi, terdapat beberapa metode untuk menyelesaiakannya, salah satunya adalah algoritma *greedy*. Untuk menyelesaikan permasalahan tersebut, ada berbagai algoritma *greedy* yang dapat digunakan. Contohnya, ada 3 algoritma *greedy* yang dapat digunakan untuk menyelesaikan permasalahan pada tugas besar ini. Meskipun begitu, ketiga algoritma tersebut memiliki hasil yang berbeda ketika dijalankan. Oleh karena itu, pemilihan algoritma *greedy* yang tepat sangat penting untuk menghasilkan solusi paling optimal.

Pada tugas besar ini, kami melakukan pengujian terhadap ketiga algoritma yang kami buat, yaitu *nearest*, *biggest*, dan *gradient*. Pengujian tersebut dilakukan selama 10 menit dengan delay pergerakan bot sebesar 0,1 detik (setara dengan 100 menit pada kondisi normal). Hasil yang kami dapatkan adalah algoritma *gradient* mampu mengumpulkan lebih banyak *diamond* daripada algoritma lainnya. Hal tersebut dapat dilihat pada Gambar 3.1. Grafik tersebut menunjukkan bahwa algoritma *gradient* mendapatkan *diamond* paling banyak, diikuti dengan algoritma *nearest* dan *biggest*. Selain itu, ada juga Gambar 3.2, 3.3, dan 3.4 yang menunjukkan *diamond* yang diperoleh pada setiap *round*-nya dengan rata-rata perolehan *diamond* tertinggi dimiliki oleh algoritma *gradient*. Dari data-data tersebut, kami memutuskan untuk menggunakan algoritma *gradient* pada dalam tugas besar ini.

Seperti yang telah disebutkan sebelumnya, pemilihan algoritma *greedy* sangat penting untuk mendapatkan hasil yang optimal. Kami bisa saja menggunakan algoritma *biggest* pada tugas besar ini. Namun, algoritma tersebut bukanlah algoritma paling optimal yang bisa digunakan. Oleh karena itu, dapat disimpulkan bahwa tidak semua algoritma *greedy* menghasilkan hasil yang sama, hanya akan ada satu (atau mungkin lebih) algoritma *greedy* yang lebih optimal daripada algoritma *greedy* lainnya. Meskipun begitu, algoritma *greedy* yang paling optimal tersebut tidak selalu lebih baik daripada algoritma yang lain. Terkadang, ada kasus tertentu yang membuat algoritma tersebut kalah dari algoritma *greedy*.

Lampiran

Github: https://github.com/atpur-rafir/Tubes1_Tuan_Krabs

Rilis strategi *gradient*: https://github.com/atpur-rafir/Tubes1_Tuan_Krabs/releases/tag/v1.0

Link video: <https://youtu.be/odKX8wjQ -I>

Daftar Pustaka

1. Slide Bahan Kuliah IF2211 Strategi Algoritma - Semester 2 Tahun 2023/2024