

Tugas Besar
IF2211 Strategi Algoritma
Pemanfaatan Algoritma *Greedy* dalam Pembuatan *Bot*
Permainan *Diamonds*



Oleh :

Justin Aditya Putra Prabakti	13522130
Muhammad Davis Adhipramana	13522157
Muhammad Rasheed Qais Tandjung	13522158

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

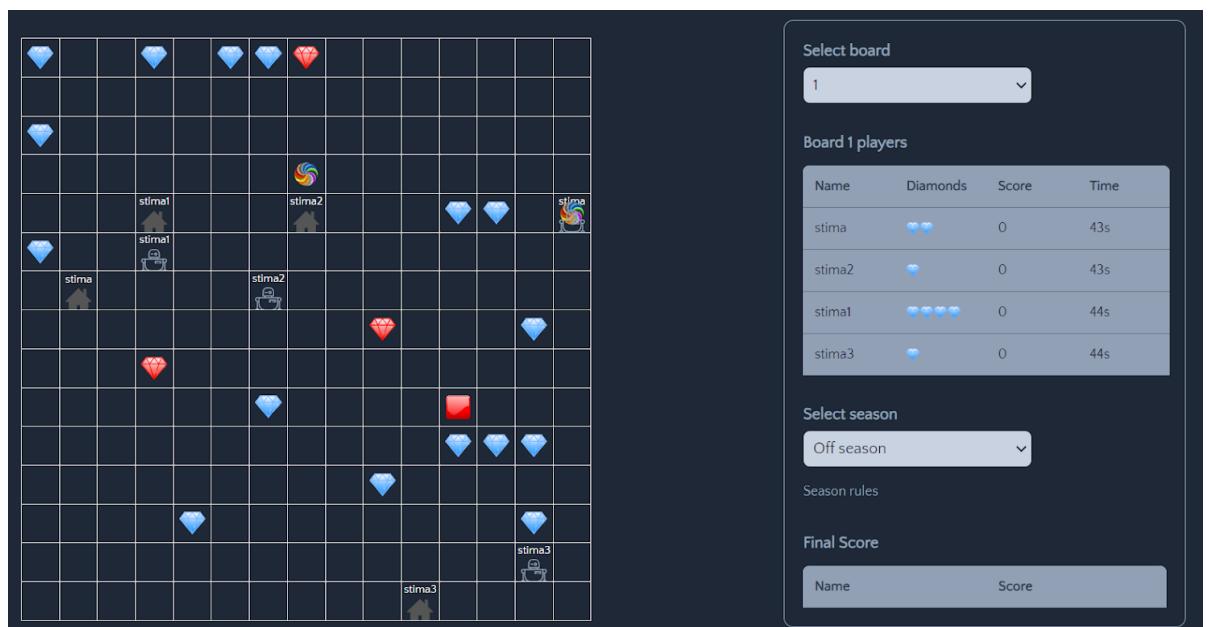
Daftar Isi

Daftar Isi.....	1
Bab I Deskripsi Tugas.....	2
Bab II Landasan Teori.....	5
2.1 Dasar Teori.....	5
2.2 Cara Kerja Bot dan Game Engine.....	7
Bab III Aplikasi Strategi Greedy.....	9
3.1 Mapping Persoalan Menjadi Elemen-Elemen Algoritma Greedy.....	9
3.2 Alternatif Solusi Greedy yang Dirancang.....	10
3.3 Analisis Efisiensi dan Efektivitas Solusi Greedy.....	14
3.4 Strategi Greedy yang dipilih.....	16
Bab IV Implementasi dan Pengujian.....	18
4.1 Implementasi Algoritma dalam Pseudocode.....	18
4.2 Penjelasan Struktur Data.....	22
4.3 Analisis Efektivitas Algoritma Greedy dalam Pengujian.....	25
Bab V Kesimpulan dan Saran.....	31
5.1 Kesimpulan.....	31
5.2 Saran.....	31
Daftar Pustaka.....	31

Bab I

Deskripsi Tugas

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

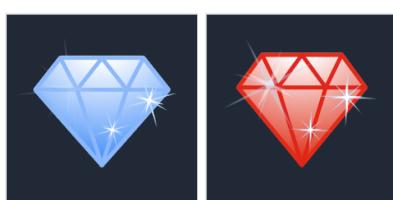


Gambar 1. Ilustrasi Board page game

Pada tugas pertama Strategi Algoritma ini, mahasiswa diminta untuk membuat sebuah bot yang nantinya akan dipertandingkan satu sama lain. Tentunya mahasiswa harus menggunakan **strategi greedy** dalam membuat bot ini.

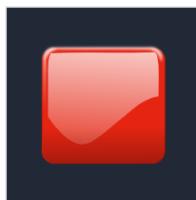
Komponen-komponen dari permainan Diamonds antara lain:

1. Diamonds



Untuk memenangkan pertandingan, kita harus mengumpulkan *diamond* ini sebanyak-banyaknya dengan melewati/melangkahinya. Terdapat 2 jenis *diamond* yaitu *diamond* biru dan *diamond* merah. *Diamond* merah bernilai 2 poin, sedangkan yang biru bernilai 1 poin. *Diamond* akan di-regenerate secara berkala dan rasio antara *diamond* merah dan biru ini akan berubah setiap *regeneration*.

2. Red Button/Diamond Button



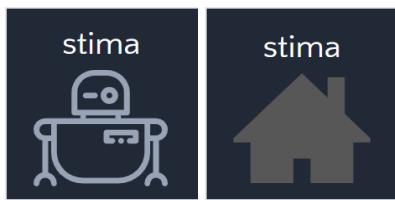
Ketika *red button* ini dilewati/dilangkahi, semua *diamond* (termasuk *red diamond*) akan di-generate kembali pada *board* dengan posisi acak. Posisi *red button* ini juga akan berubah secara acak jika *red button* ini dilangkahi.

3. Teleporters



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

4. Bots and Bases



Pada game ini kita akan menggerakkan bot untuk mendapatkan *diamond* sebanyak banyaknya. Semua bot memiliki sebuah *Base* dimana *Base* ini akan digunakan untuk menyimpan *diamond* yang sedang dibawa. Apabila *diamond* disimpan ke *base*, *score*

bot akan bertambah senilai *diamond* yang dibawa dan *inventory* (akan dijelaskan di bawah) bot menjadi kosong.

5. Inventory

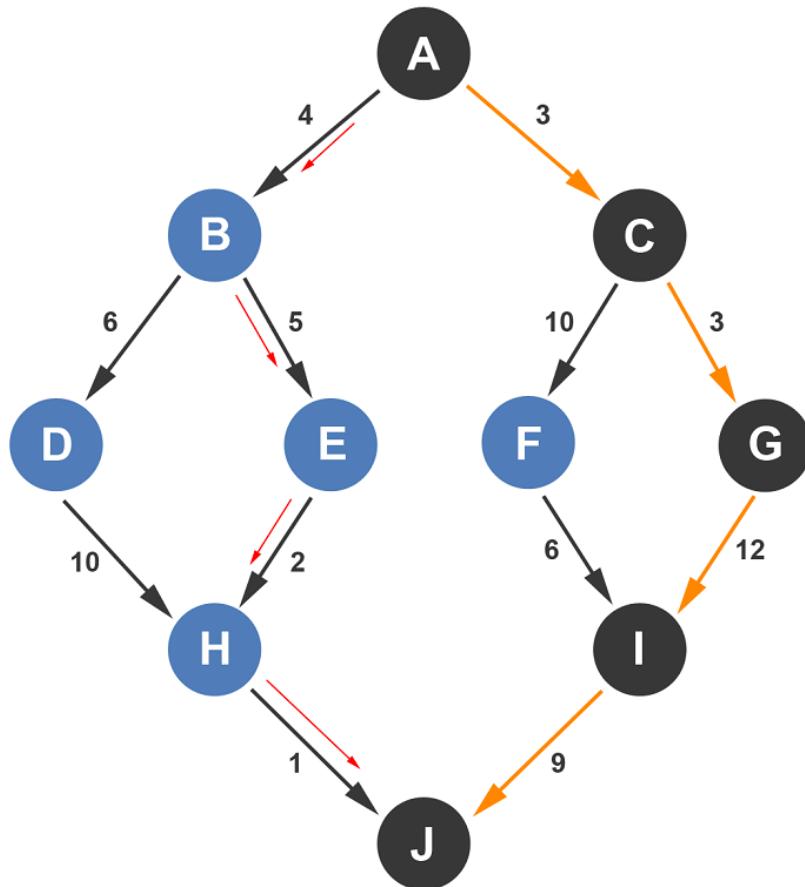
Name	Diamonds	Score	Time
stima	♥♥	0	43s
stima2	♥	0	43s
stima1	♥♥♥♥	0	44s
stima3	♥	0	44s

Bot memiliki *inventory* yang berfungsi sebagai tempat penyimpanan sementara *diamond* yang telah diambil. *Inventory* ini memiliki kapasitas maksimum sehingga sewaktu waktu bisa penuh. Agar *inventory* ini tidak penuh, bot bisa menyimpan isi *inventory* ke *base* agar *inventory* bisa kosong kembali.

Bab II

Landasan Teori

2.1 Dasar Teori



Gambar 2. Ilustrasi algoritma greedy untuk mendapatkan jarak terdekat

Algoritma *greedy* merupakan metode sederhana dalam memecahkan persoalan optimalisasi. Inti dari algoritma *greedy* adalah mengambil solusi optimal pada setiap giliran (optimum lokal) untuk mendapatkan solusi optimal secara keseluruhan (optimum global). Pada algoritma *greedy*, ketika suatu keputusan terbaik diambil, keputusan tersebut bersifat *final* dan tidak dapat kembali ke langkah sebelumnya.

Algoritma *greedy* secara umum memiliki komponen-komponen seperti berikut:

1. Himpunan kandidat (C) : kandidat yang akan dipilih pada tiap langkah
2. Himpunan solusi (S) : kandidat yang sudah dipilih
3. Fungsi solusi : menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi
4. Fungsi seleksi : memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.
5. Fungsi kelayakan : memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak)

6. Fungsi objektif : memaksimumkan atau meminimumkan

Skema dasar dari sebuah algoritma *greedy* umum seperti berikut:

```
function greedy(C: himpunan_kandidat) → himpunan_solusi
KAMUS
    x: kandidat
    S: himpunan_solusi

ALGORITMA
S <-- {}
while (not SOLUSI(S)) and (C /= {} ) do
    x <-- SELEKSI(C)
    C <-- C - {x}
    if LAYAK(S ∪ {x}) then
        S <-- S ∪ {x}
    endif
endwhile

if SOLUSI(S) then
    return S
else
    write('tidak ada solusi')
endif
```

Terdapat beberapa contoh penggunaan algoritma *greedy*, seperti *activity selection problem*, minimisasi waktu dalam sistem, *integer knapsack problem*, *fractional knapsack problem*, *Job Scheduling with Deadlines*, Pohon merentang minimum, lintasan terpendek dalam graf, kompresi data pada kode huffman, *Egyptian Fraction*, dan *Travelling Salesman Problem*.

2.2 Cara Kerja Bot dan Game Engine

```

 9   class RandomLogic(BaseLogic):
10       def __init__(self):
11           self.directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]
12           self.goal_position: Optional[Position] = None
13           self.current_direction = 0
14
15       def next_move(self, board_bot: GameObject, board: Board):
16           props = board_bot.properties
17           # Analyze new state
18           if props.diamonds == 5:
19               # Move to base
20               base = board_bot.properties.base
21               self.goal_position = base
22           else:
23               # Just roam around
24               self.goal_position = None
25
26           current_position = board_bot.position
27           if self.goal_position:
28               # We are aiming for a specific position, calculate delta
29               delta_x, delta_y = get_direction(
30                   current_position.x,
31                   current_position.y,
32                   self.goal_position.x,
33                   self.goal_position.y,
34               )
35           else:
36               # Roam around
37               delta = self.directions[self.current_direction]

```

Gambar 3. Contoh bot source code

Bot dalam permainan diimplementasikan dalam sebuah *class* yang menyimpan algoritma pengambilan keputusan bot. Algoritma utama disimpan dalam method *next_move* yang secara sederhana akan mengembalikan (*delta_x*, *delta_y*) yang merupakan perintah pergerakan bot selanjutnya. *Game engine* akan memanggil *method* tersebut pada permainan sebelum melaksanakan pergerakan bot. Nilai (*delta_x*, *delta_y*) yang di-*return* harus berupa salah satu dari nilai tersebut: [(1, 0), (-1, 0), (0, 1), (0, -1)], dan jika *game engine* mendapatkan nilai yang lain, maka akan dianggap sebagai *invalid move* dan pergerakan bot tidak akan dijalankan pada permainan.

Karena pergerakan bot hanya didasarkan oleh nilai *delta_x*, *delta_y* pada setiap giliran, dapat diimplementasikan sebuah algoritma *greedy* untuk menentukan solusi optimum lokal bagi bot untuk setiap giliran. Algoritma tersebut didapatkan dengan menganalisis komponen-komponen pada *board* (himpunan kandidat), menghitung poin-poin untuk masing komponen melalui sebuah fungsi (fungsi seleksi) untuk mendapatkan komponen dengan poin tertinggi (himpunan solusi), lalu menguji solusi tersebut sebagai keputusan yang optimum secara lokal (fungsi solusi).

```
main.py M X
main.py > ...
1  import argparse
2  from time import sleep
3
4  from colorama import Back, Fore, Style, init
5  from game.api import Api
6  from game.board_handler import BoardHandler
7  from game.bot_handler import BotHandler
8  from game.logic.random import RandomLogic
9  from game.util import *
10 from game.logic.base import BaseLogic
11 | from game.logic.mybot import MyBot ←
12
13 init()
14 BASE_URL = "http://localhost:3000/api"
15 DEFAULT_BOARD_ID = 1
16 | CONTROLLERS = {"Random": RandomLogic, "MyBot": MyBot} ↓
```

Gambar 4. Penambahan bot ke program utama

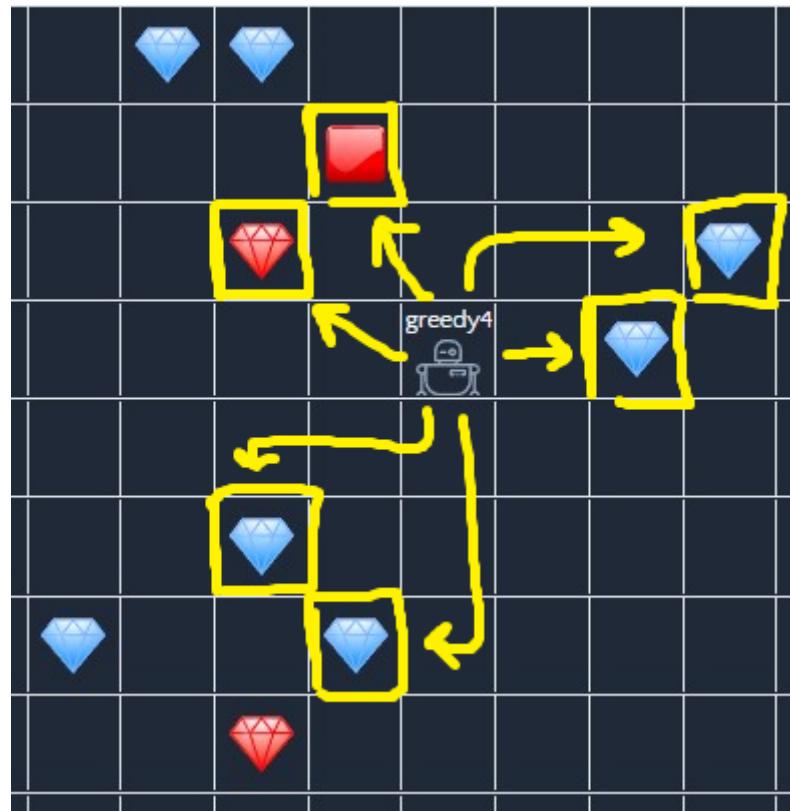
Setelah diimplementasikan algoritma *greedy* pada bot, perlu ditambahkan *logic* dari bot tersebut ke program utama permainan (*main.py*). Penambahannya cukup dengan melakukan *import* dari *class* bot yang sudah dirancang, lalu memasukan *class* tersebut ke *set CONTROLLERS*. Setelah bot sudah ditambahkan di program utama, cukup menjalankan kode tersebut di terminal atau di dalam *file .bat (.sh untuk sistem operasi linux)*.

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

Bab III

Aplikasi Strategi *Greedy*

3.1 Mapping Persoalan Menjadi Elemen-Elemen Algoritma Greedy



Gambar 5. Komponen-komponen greedy dalam permainan Diamonds

Tujuan utama dari permainan *Diamonds* adalah untuk mendapatkan poin sebanyak-banyaknya menggunakan algoritma bot yang sudah dirancang. Waktu yang diberikan untuk mengumpulkan *diamonds* terbatas, dan pada setiap giliran bot diberikan kesempatan untuk menentukan arah gerakannya. Dari konsep permainan tersebut, didapatkan sebuah permasalahan *greedy* sederhana untuk **memutuskan arah gerakan paling optimal pada setiap giliran untuk mendapatkan diamonds terbanyak pada permainan.**

Dalam implementasi greedy ini, terdapat beberapa elemen yang perlu diperhatikan. Elemen-elemen tersebut diantaranya:

1. Himpunan Kandidat

Dalam game *diamonds*, point bisa didapatkan dari mengambil *diamond* biru (berpoin satu), *diamond* merah (berpoin dua), atau bot lain (bernilai berapapun *diamond* yang mereka pegang), sehingga ketiga objek ini akan menjadi himpunan kandidat.

2. Himpunan Solusi

Dalam *game diamonds*, terdapat beberapa solusi dalam mendapatkan *point* terbanyak, yaitu dengan mencari lintasan terpendek menuju *diamond*. Selain itu, dalam *game diamonds* juga terdapat beberapa objek yang dapat membantu bot dalam

mencapai *diamonds* yaitu dengan menggunakan *portal* dan *red button*. Suatu rangkaian gerakan dianggap sebagai solusi jika dia mulai dari *base*, mengambil *diamond*, dan kembali lagi ke *base* tanpa dibunuh oleh bot lain.

3. Fungsi Seleksi

Fungsi seleksi disini merupakan fungsi yang bertugas untuk memilih kandidat-kandidat yang paling mungkin untuk mencapai solusi optimal. Dalam *game diamonds* terdapat beberapa faktor yang dapat digunakan untuk menentukan kepentasan suatu kandidat, dua faktor yang paling umum adalah poin/bobot dari suatu objek, serta jarak-nya dari *player* atau *base*.

4. Fungsi Kelayakan

Kelayakan dalam *game diamonds!* hanya berada pada gerakan individual, dimana gerakan yang menyebabkan bot keluar dari arena dan gerakan yang lebih atau tidak berarah tidak legal (Contoh, gerakan “kanan” ketika bot ada di batas kanan arena dihitung tidak layak, mengirim gerakan kosong [0,0] juga tidak layak)

5. Fungsi Solusi

Fungsi yang memeriksa apakah suatu kandidat yang dipilih dapat memberikan solusi yang layak. Karena permainan *Diamonds!* cukup dinamik (objek bisa bergerak, berubah, atau menghilang sesaat) maka fungsi kelayakan ini cukup menentukan apabila *diamond* yang dipegang sudah cukup sebelum pulang atau masih kurang sehingga harus mencari lagi.

6. Fungsi Objektif

Fungsi yang mengandung metode heuristik untuk meningkatkan efisiensi solusi sehingga lebih optimal, dapat diimplementasikan dengan beberapa cara, seperti memanfaatkan *portal* dan *red button* sehingga bisa meraih objektif dengan efisien.

3.2 Alternatif Solusi *Greedy* yang Dirancang

Dalam penentuan algoritma paling optimal untuk permainan *diamonds* tentunya perlu dilakukan banyak eksperimentasi menggunakan berbagai macam algoritma dengan logika pengambilan keputusan yang berbeda serta nilai-nilai konstanta yang bervariasi. Tim kami masing-masing bereksplorasi dan membuat beberapa implementasi *greedy* kami masing-masing untuk mencari algoritma yang paling optimal. Berikut merupakan deskripsi umum beberapa algoritma yang telah kami rancang, serta hal-hal menarik yang didapatkan masing-masing algoritma.

1. Greedy Qais (*polbot*)

Implementasi *polbot* pada awalnya hanya berdasarkan pemikiran algoritma *greedy* paling simple yaitu bot mencari *diamond* yang paling dekat. Setelah itu, dilakukan beberapa percobaan pada algoritma sederhana yang sudah dibuat, dan didapatkan hasil bahwa terkadang ketika bergerak menuju *diamond* terdekat, posisi dari bot sangat jauh dari *base*. Oleh karena itu, algoritma *greedy* berbasis jarak diubah menjadi berbasis poin prioritas.

Cara kerja algoritma *polbot* secara relatif masih cukup sederhana. Algoritma akan melakukan iterasi seluruh objek pada papan, menghitung poin prioritas objek

tersebut melalui sebuah fungsi, lalu menjadikan objek dengan poin tertinggi sebagai objek tujuan.

Secara umum, fungsi prioritas akan tetap memberikan poin tertinggi pada *diamond* terdekat, namun terdapat beberapa fitur tambahan yang digunakan untuk mengoptimalkan performa algoritma dalam konteks permainan dengan waktu yang terbatas:

- i. Untuk setiap objek dalam papan, algoritma akan memperhitungkan jarak bot ke objek dan jarak objek ke rumah. Maka dengan fitur ini, algoritma akan mementingkan objek yang dekat dengan bot sekaligus dekat dengan rumah.
- ii. Objek-objek unik seperti *diamond* merah, *diamond button*, serta rumah bot akan memiliki pengali yang berbeda dalam perhitungan poin. Contoh: *diamond biru* memiliki pengali 1, sedangkan *diamond merah* memiliki pengali 1.5, sehingga *diamond merah* akan memiliki $1.5 * \text{poin diamond biru}$.
- iii. Dengan ditambahkannya fitur poin dan pengali dalam mengambil keputusan, waktu robot untuk kembali ke rumah juga ditentukan oleh fungsi prioritas. Maka bot dapat langsung kembali ke rumah jika poin rumah cukup besar, meskipun *inventory* bot belum penuh.

Rumus perhitungan untuk masing-masing objek unik adalah berikut:

$$P(obj) = \begin{cases} \frac{1}{dist + (dist_to_base * 0.5)} & \text{if } obj = \text{blue diamond} \\ \frac{1.5}{dist + (dist_to_base * 0.5)} & \text{if } obj = \text{red diamond} \\ \frac{1.2 * diamond_density}{dist} & \text{if } obj = \text{diamond button} \\ \frac{0.12 * current_diamonds}{dist} & \text{if } obj = \text{base} \end{cases}$$

Catatan:

- *dist* = Jarak dari bot ke objek
- *dist_to_base* = Jarak dari objek ke *base*
- *diamond_density* = Jarak rata-rata seluruh *diamond* terhadap *base*
- *current_diamonds* = Jumlah *diamonds* yang ada di *inventory*

Perhitungan poin prioritas secara umum didapatkan dari pembagian dengan variabel *dist* yang bernilai jarak dari bot ke objek, sehingga jarak bot ke objek masih merupakan faktor yang paling signifikan dalam penentuan keputusan. Untuk objek *diamond* pembaginya ditambah dengan jarak objek ke *base* sehingga objek yang lebih dekat ke *base* akan memiliki poin prioritas yang lebih besar.

Poin untuk *diamond* biru dan merah dibedakan oleh *multiplier*-nya. *Diamond* biru memiliki *multiplier* 1, sedangkan *diamond* merah memiliki *multiplier* 1.5, sehingga untuk perbedaan jarak yang tidak terlalu signifikan, *diamond* merah akan memiliki poin yang lebih besar.

Untuk logika objek *diamond button* menggunakan variabel *diamond_density*, yaitu jarak rata-rata seluruh *diamond* terhadap *base*. Jika jumlah *diamond* di dekat *base* berkurang, maka rata-rata jarak *diamonds* ke *base* akan lebih besar, sehingga bobot untuk mengincar *diamond button* akan lebih besar.

2. Greedy Justin (*revelation*)



Gambar 6. Icon Revelation

Di *revelation*, terdapat penekanan memilih objek terdekat dan paling berharga, serta optimisasi seperti penggunaan portal dan pulang secara prematur, kurang lebih poin-poin dari algoritma ini adalah:

- a. Menggunakan *taxis-cab* distance
- b. Menghitung jarak dari player ke objek, serta dari player ke *portal* dan *portal* ke objek
- c. Memungkinkan pulang prematur sehingga jika ada jalur dimana bot melalui rumah jika sedang mengejar *diamond*, bot akan pulang sebentar
- d. “Threat scan” digunakan untuk menghindari daerah ramai, bot lain, atau *obstacle* seperti *teleporter* (ketika *path* tidak menggunakan *teleporter*)

Untuk merealisasikan hal tersebut terdapat 2 perubahan/fitur unik yang bertujuan untuk membantu bot mendapatkan *diamond* sebanyak-banyaknya himpunan kandidat adalah:

- a. *Priority points rework*

Dalam logic revelation, object yang menjadi kandidat merupakan *{Diamond}* merah, *Diamond button*, *Diamond* biru, *Base* sendiri, *Bot* lain} dimana setiap objek mendapat “Priority points” yang akan menentukan apa yang menjadi target bot.

Secara umum, *Priority points* sebuah objek didapatkan dengan: “Jarak * Weight” dimana jarak (untuk *Priority points*) adalah :

$$\text{Jarak} = e^{-\frac{4*dist}{max}}$$

dalam program, rumus ini disimpan sebagai “*p_func*”

Dimana *dist* adalah jarak *taxis-cab* dari bot ke benda atau dari bot ke *portal* (terdekat) dan dari *portal* (terjauh) ke objek. (Jika jarak melalui *portal* merupakan jarak yang lebih dekat, otomatis bot akan disuruh untuk mengejar

portal)

Lalu *weight* berbeda-beda untuk setiap *item*, yakni

- i. Diamond = $5 + 5 * point$, **kecuali** jika *inventory* penuh maka poin = 0
- ii. Base = $30 * (Diamond \text{ di } inventory / Max \text{ diamonds})$
- iii. Diamond Button = 12
- iv. Bot = 1

b. *Threat scan*

Objek yang dianggap threat adalah : *portal* (jika path yang dipilih tidak melalui portal) dan bot lain (jika bukan target). *Threat scan* ini nantinya akan me-return vektor yang merupakan “lawan” dari arah bahaya (contoh jika diatas persis ada *portal*, maka *threat scan* mereturn [0,1] alias BAWAH)

Untuk setiap objek yang berada dalam *fear range*, *threat point* yang dihasilkan adalah :

$$Threat = V * (-1) \times \frac{(dist - max)^2}{max^2}$$

dimana V adalah vektor arah ke musuh (*Position bot - Position musuh*)

Terakhir, perhitungan arah dilakukan dengan memilih *target* dengan *priority points* tertinggi dan menambahkan vektor arah target dengan vektor *threat point* (*Direction + Threat*).

3. Greedy Davis (*GreedyDave*)

Implementasi *GreedyDave* ini menggunakan prinsip pembobotan pada penerapan konsep *greedy*-nya dimana keputusan dengan poin terbanyak akan dijalankan pada *next move*-nya. Hal ini mirip dengan implementasi *greedy* pada *polbot* dimana sama-sama menggunakan konsep pembobotan dalam *greedy*-nya.

Namun, perbedaanya terletak pada jumlah dan cara pengaplikasian elemen penentu yang digunakan. Hal tersebut dapat dilihat pada tabel dibawah ini. Pada tabel tersebut, dapat dilihat bahwa hanya dalam penentuan satu keputusan dibutuhkan hingga 4 sampai 5 kali kalkulasi. Tiap-tiap konsiderasinya memiliki bobotnya masing-masing yang nantinya akan dikalkulasi di akhir. Setelah itu, seluruh perolehan poin akan dibandingkan satu sama lain dan akan dihasilkan suatu objek dengan poin tertinggi sebagai target pada *next move* nya.

Aturan Pembobotan

Name	Consideration	Calculation
Goes to Nearest Diamond	Current Inventory (c) (2) Distance (d) (3) Points (p) (2) Base Distance (b) (2) Time (t) (1)	if(c > p) then → 0 → (c + d + b + p + t) / total_weight
Goes to Base	Current Inventory (c) (3) Base Distance (b) (1) Time (t) (2) Enemy Distance (e) (2) Nearest Diamond (n) (2)	if(c == 5) then → 1 → (c + d + b + p + t) / total_weight
Goes to Blue Diamond	// Used Only if Nearest Diamond are resulting 0 Current Inventory (c) (3) Distance (d) (3) Time (t) (2) Base Distance (b) (2) Optional : Enemy Distance (2)	if(c == 4) then → 1 → (c + d + t + b) / total_weight
Goes to Red Diamond	// Used Only if Nearest Diamond are resulting 0 Current Inventory (c) (3) Distance (d) (4) Time (t) (2) Base Distance (b) (3) Optional : Enemy Distance (2)	→ (c + d + t + b - e) / total_Weight
Goes to Portal	DistancetoNearestDiam(d)(4) Diamond Point(p)(2) Base Distance + inventory (b)(4) Optional: Enemy Distance (e) (2)	→ (d + p + b - e) / total_weight
Goes to Red Button	Inventory (c) (2) Amount of Diamond (a) (4) Current Score to Other (s) (2) Distance (d) (2) Optional : Enemy Distance (e) (2)	→ (a + s + d + c - e) / total_weight

Gambar 7. GreedyDave Decision Table

3.3 Analisis Efisiensi dan Efektivitas Solusi *Greedy*

Berdasarkan beberapa alternatif solusi yang kami dapatkan sebelumnya, berikut merupakan analisis efisiensi dan efektivitas dari algoritma *greedy* yang kami buat:

1. Greedy Qais (*polbot*)

Pada algoritma *greedy* ini, pencarian *diamond* oleh bot efektif dan mampu beradaptasi dengan baik melawan bot lainnya. *management inventory* dan waktu juga teratasi dengan baik. selain itu, pada implementasi *greedy* ini, pergerakan menuju target dan kembali ke *base* cukup bagus. Contohnya, pada beberapa *test*, bot dapat mengambil banyak *diamond* lalu kembali ke *base*.

Algoritma *greedy* ini juga memiliki suatu kelemahan dimana bot ini seringkali mengincar *red button* jika jaraknya cukup dekat, dimana hal ini terkadang justru menguntungkan musuh.

Selebihnya terkait efisiensi dari algoritma ini akan dibahas pada *Bab 4.3*.

2. Greedy Justin (*revelation*)

Algoritma *greedy* ini memiliki keuntungan bahwa jika terdapat base ketika sedang bergerak menuju suatu *diamond* target, maka bot akan dapat bergerak menuju base. Serta algoritma ini juga mengkalkulasi jarak pada musuh yang mengurangi kemungkinan bot terbunuh oleh bot lainnya.

Pada algoritma ini memiliki sebuah *edge case* dimana bot bisa di “Kabedon” jika terlalu dekat dengan bot lain dan dinding



Gambar 8. “Kabedon”, pada posisi seperti ini, bot tidak bisa bergerak dan diam di tempat karena terlalu takut (source google “kabedon”)

3. Greedy Dave

Algoritma *greedy* ini memiliki keuntungan dimana jika jumlah *diamond* masih banyak, algoritma ini akan berjalan dengan lancar dimana keputusan yang diambil seringkali bersifat efektif. keputusan yang diambil juga sangat dinamis karena pengambilan keputusan memiliki banyak konsiderasi yang menyebabkan keputusan yang diambil dapat beradaptasi sesuai situasi *bot* dan *board* saat itu.

Algoritma *greedy* ini juga memiliki kelemahan dimana jika *diamond* yang berada di *board* mulai menipis terkadang bot akan berada dalam state “confused”

dimana bot akan bergerak ke kanan dan kiri secara terus menerus. Hal itu terjadi karena terjadi perpindahan target pada antara 2 objek pada saat bergerak.



Gambar 9. "Confused", pada posisi seperti ini, bot hanya bergerak ke kanan dan kiri karena terjadi perpindahan target antara 2 objek ketika bergerak

3.4 Strategi Greedy yang dipilih

1. Aturan Penilaian

Terdapat beberapa parameter yang kami jadikan sebagai konsiderasi dalam pemilihan strategi *greedy* yang akan digunakan. Hal itu meliputi:

- Jumlah rasio kemenangan jika algoritma dijalankan dalam 1 menit, dengan delay waktu bergerak 0.25 detik dengan algoritma lainnya.
- Perilaku bot dalam kondisi tertentu, seperti ketika bot dekat dengan bot lainnya dan base berada di ujung.
- Kesulitan atau jumlah *edge case* atau *error* yang mungkin terjadi pada bot

2. Analisis algoritma

DATA TURNAMEN WE LOVE MINERS						Condition Table
Match	Greedy Qais	Greedy Justin	Greedy Davis	Condition	HasUnfairBase	
1	56	41	41	unfair-1	justin	fair-1 All the base located on center
2	45	43	7	unfair-1	justin	unfair-1 1 bot has base on corner or side
3	56	49	27	base-1	davis, justin	unfair-2 2 bot has base on corner or side
4	46	45	33	unfair-1	qais	fair-2 All the base located on corner or side
5	44	43	45	unfair-1	qais	base-1 2 bot has base near each other
6	57	48	44	fair-1	-	color of winner
7	55	46	39	fair-1	-	
8	32	53	40	unfair-2	davis, qais	
9	49	27	55	base-1	justin, qais	
10	45	52	30	unfair-1, base-1	unfair(davis), base-1(justin,qais)	
11	55	46	36	fair-1	-	
12	61	33	34	unfair-2, base-1	unfair(davis, qais), base-1(justin,qais)	
13	50	34	37	fair-2, base-1	base-1(justin,davis)	
14	44	56	36	unfair-1	davis	
15	34	51	37	unfair-1, base-1	unfair(qais), base-1(justin,qais)	

Gambar 10. Data turnamen intra WLM

data tersebut merupakan data yang diambil dari 15 kali pertandingan antara ketiga bot yang masing masing mengandung algoritma pada alternatif solusi. Dapat dilihat,

- Greedy Qais

- Sesuai dengan kriteria penilaian a, bot dengan algoritma qais mendominasi pertandingan dengan rasio kemenangan $\frac{9}{15}$ atau 9 kemenangan dari 15 pertandingan.
- Berdasarkan kriteria penilaian b, pertama, dalam keadaan dekat dengan bot lain, bot dengan algoritma qais tidak memiliki algoritma untuk menghindari serangan bot lain. Dalam keadaan base berada di ujung, bot qais masih mampu mendapatkan score yang cukup baik, hal ini dapat dilihat berdasarkan pertandingan ke-4, 10, 12 dimana score perolehan qais mampu meraih kemenangan maupun hampir menang.
- Pada algoritma ini tidak ada *edge case* atau *error* yang terjadi. Namun, bot terkadang tidak sengaja membantu musuh dengan menekankan tombol merah.

b. Greedy Justin

- Sesuai kriteria penilaian a, bot dengan algoritma Justin hanya mendapatkan 4 dari 15 kemenangan.
- Berdasarkan kriteria penilaian b, pertama, dalam keadaan dekat dengan bot lain, bot dengan algoritma qais memiliki sejumlah mekanisme dalam penanganannya yaitu antara menghindari bot tersebut ataupun mencoba untuk menyerang bot lainnya. Mekanisme ini dapat menghasilkan teknik berlindung dan menyerang yang baik dimana pada pertandingan 3, 10, dan 15 dimana bot justin memiliki *score* yang cukup baik karena berdekatan dengan *base* bot lain yang memungkinkan seringnya pertemuan antara dua bot tersebut. Kedua, dalam keadaan *base* berada di ujung atau tepi, pada keadaan ini algoritma justin masih dapat beradaptasi dimana pada putaran ke-1 dan 2, bot justin memiliki *score* yang cukup baik.
- Algoritma ini memiliki *edge case* yang sekaligus menyebabkan *error* dimana jika bot terlalu dekat dengan dinding dan bot lain, bot ini akan mengalami “kabedon” yang menyebabkan *invalid move* menuju luar dinding.

c. Greedy Davis

- Sesuai kriteria penilaian a, bot dengan algoritma Davis mendapatkan rasio kemenangan paling kecil yaitu $\frac{2}{15}$ atau 2 kemenangan dari 15 pertandingan.
- Berdasarkan kriteria penilaian b, pertama, dalam keadaan dekat dengan bot lain, tidak ada implementasi penanganan serangan dengan bot lain. kedua, dalam keadaan base berada di ujung, algoritma Davis hanya dapat menghasilkan score sekitar 30.
- Algoritma ini memiliki *edging case* dimana ketika dua objek memiliki jarak yang sama memungkinkan mereka memiliki jumlah poin yang sama. Hal ini menyebabkan bot mengalami state “confused” dimana bot hanya bergerak kanan-kiri atau maju-mundur hingga keputusan lain dibuat.

3. Penentuan

Berdasarkan analisis yang kami lakukan pada ketiga bot, kami memutuskan memilih bot dengan **algoritma greedy qais (polbot)** sebagai strategi *greedy* kami. Hal ini berdasarkan kriteria penilaian yang sangat cocok pada algoritma qais dimana mendapat kemenangan dalam kebanyakan pertandingan. selain itu, mampu beradaptasi dengan baik dalam keadaan *base* yang berada di ujung. serta, tidak memiliki *error* apapun.

Bab IV

Implementasi dan Pengujian

4.1 Implementasi Algoritma dalam *Pseudocode*

```

// class Greedy4

INCLUDE random

CLASS Greedy4
    (Konstruktor objek)
    CONSTRUCTOR Greedy4()
        {Position attributes untuk menyimpan data lokasi objek dalam
board}
        position THIS.goal_position
        position THIS.buttonPos
        game_object THIS.tela
        game_object THIS.telB

        {Boolean attributes untuk membantu penentuan keputusan algoritma}
        boolean THIS.returnHome = False
        boolean THIS.useTel = False

        {Miscellaneous attributes sebagai data pembantu lainnya}
        time THIS.lastUseButton
        time THIS.lastGoHome
        integer THIS.diamondDensity = 1

    {Menghitung jarak dua posisi dalam board}
    METHOD block_distance(position a, position b) -> integer
        return abs(a.x - b.x) + abs(a.y - b.y)

    {Mendapatkan game_object teleporter pada board}
    METHOD get_teleporters(game_object bot, board board) -> (game_object,
game_object)
        position pos = bot.position

        {Melakukan iterasi seluruh game object untuk mendapatkan
teleporter}
        List[game_object] tel = []
        for obj in board.game_objects do
            if obj.type = "TeleportGameObject" then
                tel += obj

    {Menghitung jarak teleporterA dan teleporterB}
    integer distA = THIS.block_distance(pos, tel[0].position)
    integer distB = THIS.block_distance(pos, tel[1].position)

    {Menentukan teleporter yang lebih dekat ke bot}
    if distA <= distB then

```

```

        return (tel[0], tel[1])
    else
        return (tel[1], tel[0])

    {Menerima input berupa game_object dan menghitung poin yang
     dihasilkan berdasarkan sebuah rumus pembobotan}
    METHOD priorityPoints(game_object bot, game_object obj) -> float
        {Mendapatkan nilai jarak bot ke object dan object ke base, serta
         menghitung jarak menggunakan teleporter}
        properties props = bot.properties
        integer dist = THIS.block_distance(bot.position, obj.position)
        integer dist_to_base = THIS.block_distance(bot.properties.base,
obj.position)
        integer dist_with_tel = THIS.block_distance(bot.position,
THIS.telA.position) + THIS.block_distance(THIS.telB.position,
obj.position)

        {Gunakan teleporter jika jarak dengan teleporter lebih dekat}
        if dist_with_tel < dist and bot.position != THIS.telA.position
then
            THIS.useTel = True
            dist = dist_with_tel
        else
            THIS.useTel = False

        {Melakukan kalkulasi poin}
        if dist = 0 then
            return 0
        else
            {Jika object adalah diamond}
            if obj.type = "DiamondGameObject" then
                if ((props.diamonds + obj.properties.points) >
props.inventory_size) then
                    return 0
                else if obj.properties.points = 2 then
                    return 1.5 / (dist + (dist_to_base * 0.5))
                else
                    return 1 / (dist + (dist_to_base * 0.5))

            {Jika inventory tidak cukup, poin object adalah 0}
            {Jika object adalah red diamond, multiplier bernilai 1.5}
            {Jika object adalah blue diamond, multiplier adalah 1}

            {Jika object adalah button}
            elif obj.type = "DiamondButtonGameObject" then
                if THIS.lastUseButton then
                    return 0
                integer points = (1.2 / dist) * THIS.diamondDensity
                if obj.position != THIS.buttonPos then
                    THIS.buttonPos = obj.position
                    THIS.lastUseButton = bot.properties.milliseconds_left
                return points

```

```

{Menentukan objek yang akan menjadi tujuan bot di giliran selanjutnya}
METHOD nextGoal(game_object bot, board board) -> position
    {Inisialisasi variabel-variabel pembantu}
    boolean useTel = False
    float max_points = THIS.priorityPoints(bot, board.diamonds[0])
    position goal_pos = board.diamonds[0].position
    THIS.teleporters = []

    {Variabel untuk menghitung densitas diamond pada board: total distance, distance maksimal, dan jumlah diamond}
    integer diam_dist_total = 0
    integer diam_dist_max = 0
    integer diam_count = 0

    {Melakukan iterasi terhadap seluruh game_object untuk mendapatkan objek dengan poin tertinggi}
    for obj in board.game_objects
        {Jika object adalah teleporter}
        if obj.type = "TeleportGameObject" then
            THIS.teleporters.append(obj.position)
        {Jika object adalah diamond atau button}
        if obj.type = "DiamondGameObject" or obj.type =
        "DiamondButtonGameObject" then
            {Hitung poin object dan cek apakah nilainya lebih besar dari nilai maksimal}
            points = THIS.priorityPoints(bot, obj)
            if points > max_points then
                max_points = points
                goal_pos = THIS.telA.position if THIS.useTel else
obj.position

            {Jika object adalah diamond, tambahkan data untuk perhitungan densitas diamond}
            if obj.type = "DiamondGameObject" then
                diam_count += 1
                diam_dist = THIS.block_distance(bot.properties.base,
obj.position)
                diam_dist_total += diam_dist
                if diam_dist > diam_dist_max then
                    diam_dist_max = diam_dist

            {Hitung densitas diamond}
            THIS.diamondDensity = (diam_dist_total * 2 / diam_count) /
diam_dist_max

            {Lakukan perhitungan untuk memutuskan apakah bot lebih baik untuk pulang ke base}
            integer home_dist = THIS.block_distance(bot.position,
bot.properties.base)
            integer home_dist_tel = THIS.block_distance(bot.position,

```

```

THIS.telA.position) + THIS.block_distance(THIS.telB.position,
bot.properties.base)

    {Jika jarak ke base lebih dekat dengan teleporter, gunakan
teleporter}
    if home_dist_tel < home_dist and bot.position != 
THIS.telA.position then
        home_dist = home_dist_tel
        useTel = True
    else
        useTel = False

    {Kalkulasi keuntungan pulang ke base}
    if home_dist > 0 and bot.properties.diamonds > 0 then
        float home_points = (0.12 * bot.properties.diamonds) /
home_dist
        float seconds_left = bot.properties.milliseconds_left / 1000
        if (home_points > max_points or bot.properties.diamonds == 
bot.properties.inventory_size or seconds_left < 8) then
            goal_pos = THIS.telA.position if useTel else
bot.properties.base

    {Return posisi objek dengan poin tertinggi}
    return goal_pos

{Memberikan keputusan pergerakan selanjutnya ke permainan}
METHOD next_move(game_object bot, board board) -> (integer, integer)
    {Reset cooldown penggunaan button}
    if THIS.lastUseButton and (THIS.lastUseButton -
bot.properties.milliseconds_left) > 7000 then
        THIS.lastUseButton = None

    {Dapatkan posisi teleporter pada board}
    tel = THIS.get_teleporters(bot, board)
    THIS.telA = tel[0]
    THIS.telB = tel[1]

    {Dapatkan posisi object tujuan}
    THIS.goal_position = THIS.nextGoal(bot, board)

    {Konversi posisi object tujuan menjadi pergerakan
horizontal/vertikal}
    if THIS.goal_position then
        delta_x, delta_y = get_direction(
            bot.position.x,
            bot.position.y,
            THIS.goal_position.x,
            THIS.goal_position.y,
        )

    {Kembalikan pergerakan selanjutnya}
    return delta_x, delta_y

```

4.2 Penjelasan Struktur Data

A. Base Class (Greedy4)

```
class Greedy4(BaseLogic):
    def __init__(self):
        # Position attributes
        self.goal_position: Optional[Position] = None
        self.buttonPos: Optional[Position] = None
        self.telA: Optional[GameObject] = None
        self.telB: Optional[GameObject] = None

        # Boolean attributes
        self.returnHome = False
        self.useTel = False

        # Additional helper attributes
        self.lastGoHome: Optional[int] = None
        self.lastUseButton: Optional[int] = None
        self.diamondDensity = 1
```

Atribut-atribut pada *base class* akan digunakan untuk menyimpan informasi terkait permainan yang tetap ingin disimpan pada iterasi *next_move* selanjutnya. Atribut-atribut kelas dikelompokkan berdasarkan jenis informasi yang disimpan, yaitu atribut posisi, atribut *boolean*, serta atribut tambahan.

Atribut posisi akan menyimpan informasi posisi objek-objek penting pada board, seperti *diamond button* dan *teleporter*, serta data posisi yang ingin dituju berupa *goal_position*. Informasi posisi *diamond button* dan *teleporter* disimpan sebagai atribut agar algoritma dapat mengetahui ketika objek-objek tersebut telah berpindah lokasi.

Contoh dari pemanfaatan informasi posisi sebagai atribut adalah untuk mendeteksi ketika *diamond button* telah ditekan oleh salah satu pemain. Indikasi dari terjadinya *board reset* akibat pemencetan *diamond button* adalah ketika lokasi *diamond button* telah berpindah dari lokasi sebelumnya. Jika informasi lokasi *diamond button* hanya disimpan sebagai variabel lokal, algoritma bot tidak memiliki konteks atas lokasi tombol sebelumnya, sehingga tidak dapat mengetahui ketika papan telah di-*reset*.

B. get_teleporters

```
tel: List[GameObject] = []
for obj in board.game_objects:
    if obj.type == "TeleportGameObject":
        tel.append(obj)
```

```
distA = self.block_distance(pos, tel[0].position)
distB = self.block_distance(pos, tel[1].position)
```

Metode *get_teleporters* digunakan untuk mendapatkan lokasi kedua *teleporter* dalam papan. Kode metode akan melakukan iterasi atas semua *game_object*, dan ketika menemukan objek *teleporter*, objek tersebut akan ditampung sementara di *array tel*.

Program lalu akan menghitung jarak bot terhadap kedua *teleporter* dan menentukan *teleporter* yang terdekat. Setelah *teleporter* terdekat didapatkan, program akan mengembalikan sebuah *tuple* (*teleporter*, *teleporter*) dengan *teleporter* terdekat sebagai *teleporter* pertama pada *tuple*.

C. priorityPoints

```
def priorityPoints(self, bot: GameObject, obj: GameObject) ->
float:
    props = bot.properties
    dist = self.block_distance(bot.position, obj.position)
    dist_to_base = self.block_distance(bot.properties.base,
obj.position)

    dist_with_tel = self.block_distance(bot.position,
self.telA.position) + self.block_distance(self.telB.position,
obj.position)
    if dist_with_tel < dist and bot.position != self.telA.position:
        self.useTel = True
        dist = dist_with_tel
    else:
        self.useTel = False
```

Metode *priorityPoints* akan menerima sebuah objek dalam papan dan melakukan kalkulasi untuk menetapkan prioritas objek tersebut. Nilai prioritas tersebut akan digunakan di metode selanjutnya untuk menentukan objek dengan prioritas tertinggi dari seluruh objek yang ada di papan, sesuai dengan mekanisme algoritma *greedy*.

Program akan memanfaatkan nilai *dist*, *dist_to_base*, *dist_with_tel*, serta atribut boolean *self.useTel* untuk menghitung poin prioritas tersebut. Nilai *dist* adalah jarak bot ke objek, dan nilai *dist_to_base* adalah jarak objek ke rumah. Fungsi secara umum akan memberikan nilai prioritas yang lebih besar ke objek yang lebih dekat ke bot, namun jarak objek ke rumah juga akan diperhitungkan, dan objek-objek yang lebih jauh dari rumah akan mendapatkan pengurangan poin.

Nilai *dist* yang didapatkan akan dibandingkan dengan nilai *dist_with_tel*, yaitu jarak bot ke objek dengan menggunakan *teleporter*. Jika jarak dengan *teleporter* lebih dekat, bot

akan merubah lokasi tujuannya ke *teleporter* terdekat dengan mengubah *self.use_tel* menjadi *True*.

D. nextGoal

```
def nextGoal(self, bot: GameObject, board: Board):
    useTel = False
    max_points = self.priorityPoints(bot, board.diamonds[0])
    goal_pos = board.diamonds[0].position

    diam_dist_total = 0
    diam_dist_max = 0
    diam_count = 0

    # ...
    # ...

    self.diamondDensity = (diam_dist_total * 2 / diam_count) /
diam_dist_max

    # Calculate points for returning to home
    home_dist = self.block_distance(bot.position,
bot.properties.base)
    home_dist_tel = self.block_distance(bot.position,
self.telA.position) + self.block_distance(self.telB.position,
bot.properties.base)
    if home_dist_tel < home_dist and bot.position != self.telA.position:
        home_dist = home_dist_tel
        useTel = True
    else:
        useTel = False
```

Metode *nextGoal* akan menentukan lokasi tujuan bot pada sebuah giliran tertentu. Lokasi tujuan didapatkan dengan melakukan iterasi terhadap seluruh objek yang ada di papan, menghitung poin prioritasnya dengan metode *priorityPoints*, lalu memilih lokasi objek dengan poin prioritas tertinggi sebagai lokasi tujuan.

Metode tersebut akan memulai pencarian dengan pertama menginisialisasikan variabel-variabel *max_points* dan *goal_pos* untuk menyimpan objek dengan prioritas tertinggi, serta variabel *boolean useTel* untuk menentukan apakah bot akan menggunakan

teleporter. Pada tahap awal ini juga akan diinisialisasikan variabel *diam_** yang akan digunakan untuk menghitung densitas *diamond* di dekat rumah.

Setelah mendapatkan *diamond* dengan prioritas tertinggi, program akan mempertimbangkan prioritas mengincar *diamond button*. Pengincaran *diamond button* ditentukan atas jarak bot ke tombol serta densitas *diamond* di dekat rumah. Secara umum, prioritas *diamond button* akan meningkat ketika *diamond density* di dekat rumah mengecil.

Program juga akan mempertimbangkan prioritas kembali ke rumah dengan variabel *home_dist*. Jika jarak bot cukup dekat ke rumah, bot akan memprioritaskan kembali ke rumah walaupun *inventory* bot masih belum penuh. Jika jarak ke rumah lebih dekat menggunakan *teleporter*, bot akan menjadikan *teleporter* sebagai tujuan utama.

4.3 Analisis Efektivitas Algoritma *Greedy* dalam Pengujian

Untuk menguji efektivitas algoritma *greedy* yang dirancang (*polbot*), kami merancang serangkaian *test-run* dengan algoritma-algoritma lainnya untuk melihat performa algoritma pada berbagai kasus-kasus tertentu. Berikut merupakan hasil dari rangkaian percobaan yang kami jalankan.

PENGUJIAN POLBOT						Penjelasan			
No	Base Condition	Move Delay	Score	Highest Score	Score ratio	Base Condition	Kondisi base dari bot saat pertandingan	Move Delay	Delay tiap pergerakan pada bot
1	Base berada di ujung board	1	4	12	0.33	Base berada di ujung board	Base berada di ujung board	Score	Jumlah score bot setelah pertandingan
2	Base berdekatan dengan bot lain	1	15	16	0.9357	Base berada di ujung board	Base berada di ujung board	Highest Score	Jumlah score tertinggi dari 3 bot saat pertandingan
3	Base berada di tepi board	1	13	13	1	Base berada di ujung board	Base berada di ujung board	Score ratio	Perbandingan score saat bot dengan highest score
4	Base berada di tengah	1	11	11	1	Base berada di ujung board	Base berada di ujung board		
5	Base berada di tepi board	1	6	12	0.5	Base berada di ujung board	Base berada di ujung board		
6	Base berada di tengah	1	10	10	1	Base berada di ujung board	Base berada di ujung board		
7	Base berada di tepi board	1	20	20	1	Base berada di ujung board	Base berada di ujung board		
8	Base berada di ujung board	0.5	20	30	0.66	Base berada di ujung board	Base berada di ujung board		
9	Base berada di tepi board	0.5	21	32	0.65	Base berada di ujung board	Base berada di ujung board		
10	Base berada di tengah	0.5	34	34	1	Base berada di ujung board	Base berada di ujung board		
11	Base berdekatan dengan bot lain	0.5	27	27	1	Base berada di ujung board	Base berada di ujung board		
12	Base berada di ujung board	0.5	34	34	1	Base berada di ujung board	Base berada di ujung board		
13	Base berada di tepi board	0.5	20	20	1	Base berada di ujung board	Base berada di ujung board		
14	Base berdekatan dengan bot lain	0.5	19	20	0.95	Base berada di ujung board	Base berada di ujung board		
15	Base berada di tengah	0.1	96	107	0.897	Base berada di ujung board	Base berada di ujung board		
16	Base berdekatan dengan bot lain	0.1	86	107	0.803	Base berada di ujung board	Base berada di ujung board		
17	Base berada di tengah	0.1	115	115	1	Base berada di ujung board	Base berada di ujung board		
18	Base berada di tepi	0.1	113	116	0.974	Base berada di ujung board	Base berada di ujung board		
19	Base berada di tepi	0.1	77	103	0.747	Base berada di ujung board	Base berada di ujung board		
20	Base berdekatan dengan bot lain	0.1	100	100	1	Base berada di ujung board	Base berada di ujung board		
21	Base berada di ujung board	0.1	102	102	1	Base berada di ujung board	Base berada di ujung board		

Gambar 11. Hasil pengujian algoritma *greedy*

Dari data pengujian, didapatkan rata-rata *score ratio* untuk masing-masing kasus serta *move delay* seperti berikut.

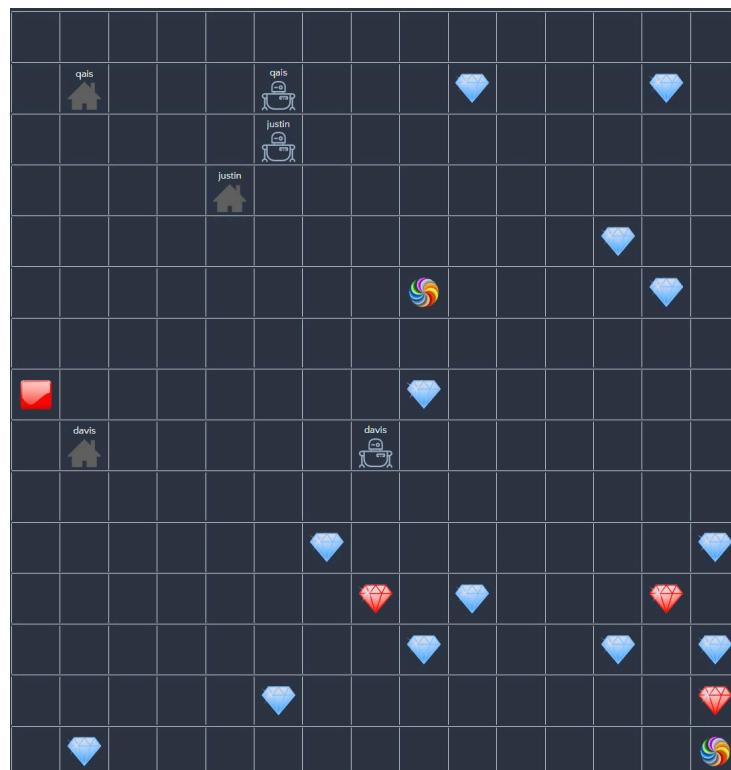
Kasus	Rata-rata score ratio
Base berada di ujung board	0.7475
Base berdekatan dengan bot lain	0.93774
Base berada di tepi board	0.811833333
Base berada di tengah	0.97425

Speed delay	Rata-rata score ratio
1	0.82
0.5	0.894285714
0.1	0.917285714

Gambar 12. Rata-rata score ratio untuk masing-masing kasus

Variabel *score_ratio* adalah rasio skor bot dengan skor maksimal yang didapatkan pada ronde tertentu. Skor 1 menandakan skor yang didapat bot = skor maksimal, yang menandakan botnya menang atau seri dengan bot lain. *Speed delay* adalah *cooldown* bot untuk bergerak. Nilai *speed delay* yang lebih kecil menandakan pergerakan bot yang lebih cepat.

Didapatkan bahwa penurunan speed delay akan menyebabkan *score ratio* yang lebih tinggi. Ini karena *speed delay* yang tinggi menyebabkan permainan lebih dependen kepada keberuntungan dibanding performa algoritma. Untuk lokasi *base*, didapatkan bahwa ketika *base* lebih dekat ke tengah, nilai *score ratio* akan lebih baik, dan semakin jauh dari tengah, nilai *score ratio* akan semakin buruk.



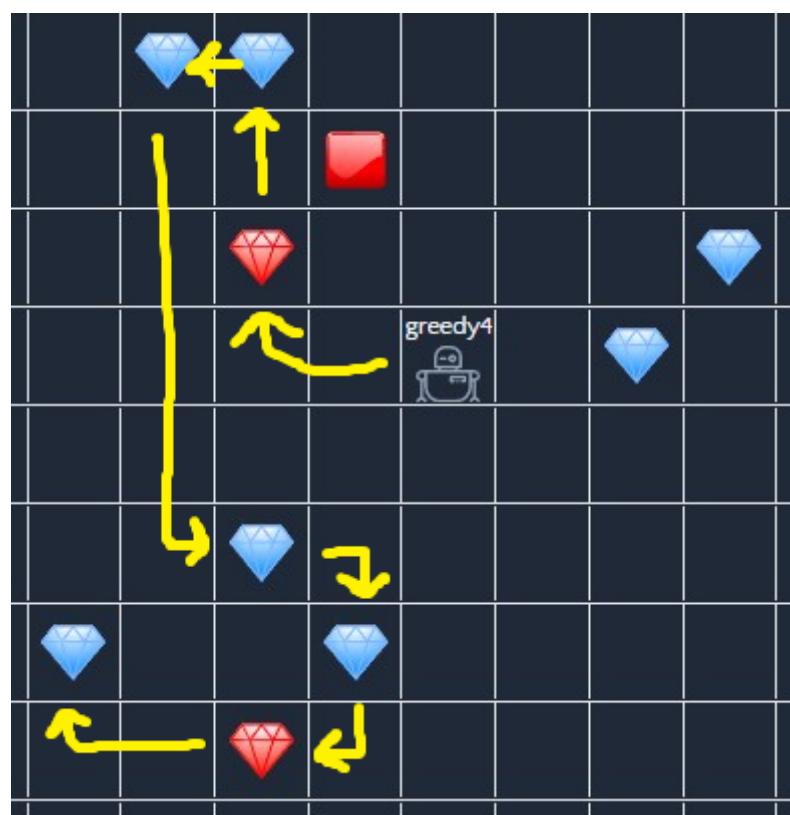
Gambar 13. Contoh lokasi base di pojok papan

Ketika *base* berlokasi di tengah papan, rata-rata jarak setiap kotak ke *base* lebih sedikit, dan kemungkinan munculnya *diamonds* yang lebih dekat ke *base* akan lebih besar, sehingga kemungkinan untuk menang pada kondisi tersebut lebih besar. Ketika *base* makin jauh dari tengah papan, kemungkinan *diamonds* muncul di dekat *base* lebih sedikit, sehingga bot perlu jalan lebih jauh untuk mendapatkan *diamonds*.

Dari hasil pengujian, didapatkan hasil *score ratio* ketika *base* di tengah dibandingkan dengan *score ratio* ketika *base* di pojok memiliki selisih **0.22675**, hasil dari *base* di pojok hampir $\frac{1}{4}$ lebih buruk dari ketika *base* berada di tengah. Penurunan performa *bot* ketika lokasi *base* berubah bukan merupakan konsekuensi dari algoritma yang dirancang, melainkan sebuah *side effect* dari sistematika permainan yang bersifat *random*.

Selain variasi performa akibat lokasi *base*, terdapat beberapa kasus khusus yang sering muncul pada permainan *diamonds* yang memperlihatkan beberapa kekurangan dan kelebihan dari algoritma yang dirancang.

A. Kasus normal



Gambar 14. Pemilihan algoritma pada skenario umum

Pada kasus umum, algoritma ini dapat dikatakan cukup optimal dalam memutuskan tujuan. Bot akan pada umumnya mengutamakan *diamond* yang terdekat, sehingga waktu yang dihabiskan untuk mengumpulkan *diamonds* memiliki nilai yang cukup dekat ke minimum.



Gambar 15. Algoritma memprioritaskan diamond button ketika jumlah diamonds di dekat base sudah sedikit

Pada kasus tertentu seperti di Gambar 15 dimana terdapat sedikit sekali *diamonds* di dekat *base*, maka algoritma akan mengutamakan *diamond button* agar densitas *diamonds* di dekat *base* dapat bertambah, dan bot tidak perlu bergerak jauh dari *base*.

B. Pengincaran *diamond button*



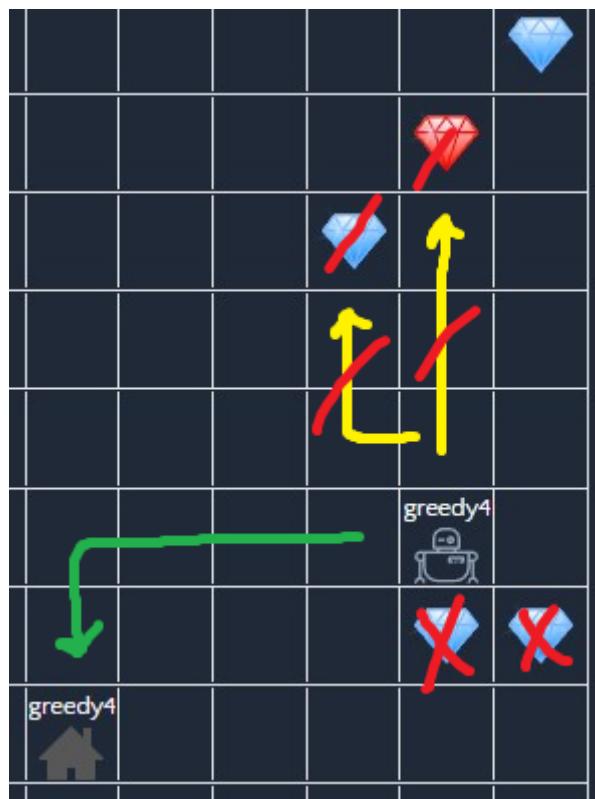
Gambar 16. Contoh pengincaran *diamond button* oleh algoritma

Double-edged sword terbesar dari algoritma *polbot* adalah strateginya untuk banyak mengincar *diamond button*. Fitur ini ditujukan sebagai strategi optimalisasi yang mengharapkan agar *randomizer* memunculkan banyak *diamond* di dekat base dan bot tidak perlu pergi jauh-jauh dari *base* untuk mendapatkan *diamonds*.

Namun sifat *diamond button* yang sangat *random* memunculkan banyak sekali *unpredictability* dalam permainan. Berikut merupakan tabel efek positif dan negatif yang dapat dimunculkan akibat pengincaran *diamond button*:

Efek positif	Efek negatif
Memunculkan banyak <i>diamonds</i> di dekat <i>base</i> pemain	Memunculkan banyak <i>diamonds</i> di dekat <i>base</i> lawan
Mengecoh algoritma lawan dengan menghilangkan <i>diamond</i> yang diincar	Menghilangkan banyak <i>diamonds</i> yang tadinya dekat dengan pemain/ <i>base</i>
	Menghabiskan waktu bot untuk mengincar <i>button</i> yang dapat menguntungkan/merugikan pemain

C. Sistem “pulang di akhir”



Gambar 17. Contoh prosedur “pulang di akhir” pada algoritma

Sistem “pulang di akhir” adalah fitur pada algoritma yang menyuruh bot untuk langsung kembali ke base ketika waktu tersisa < 10 detik. Fitur ini dibuat agar bot tidak

“membuang” *diamonds* yang sudah didapat di akhir karena belum dikembalikan ke *base* sebelum waktu berakhir.

Fitur ini pada umumnya dapat menguntungkan bot karena mengurangi peluang kejadian *diamonds* terbuang pada akhir permainan, namun terkadang akan mengakibatkan pengambilan keputusan yang tidak logis seperti di Gambar 17, ketika terdapat banyak *diamonds* di dekat bot yang tidak diambil karena bot mengutamakan kembali ke *base*.

D. Rapid-storing



Gambar 18. Contoh fitur rapid-storing pada algoritma

Fitur *rapid-storing* adalah salah satu fitur utama lainnya pada algoritma ini, yaitu bot akan kembali ke rumah jika dirasa cukup dekat. Bot akan tetap kembali ke rumah walaupun *inventory* belum penuh, sehingga algoritma akan memperbanyak penyetoran *diamonds* ke rumah dan dapat mengosongkan *inventory* untuk mengumpulkan *diamonds* lain yang lebih jauh. Fitur ini juga mengamankan *diamonds* yang sudah diambil apabila bot ke depannya dibunuh oleh bot lain.

Fitur ini pada umumnya meningkatkan optimalisasi algoritma, karena memperbolehkan bot untuk mendapatkan lebih banyak *diamonds* ketika pergi lebih jauh dari *base*. Namun fitur ini terkadang merugikan bot karena menghabiskan banyak waktu melakukan banyak proses bolak-balik ke *base* jika *diamonds* di dekat *base* cukup berjarak satu sama lain.

Bab V

Kesimpulan dan Saran

5.1 Kesimpulan

Algoritma *greedy* merupakan algoritma yang cukup *versatile* dan simpel. Algoritma ini merupakan algoritma umum yang dapat diterapkan di berbagai masalah. Salah satunya adalah dalam permainan diamond ini dimana algoritma greedy dapat membantu bot dalam mengidentifikasi objek mana yang harus dijadikan target oleh bot untuk mendapatkan score yang tinggi dalam kurun waktu 1 menit.

5.2 Saran

Kami menyarankan pada saat menggunakan algoritma *greedy* banyak hal yang harus diperhatikan selain dari objective *primary* seperti *diamond*. Hal yang perlu diperhatikan selain itu adalah management waktu, *inventory*, jarak antara *base* dengan *diamond*, dan keputusan pengambilan *red button* yang bisa saja malah menguntungkan lawan, serta *handle* perlawanan serangan lawan ketika *base* berdekatan.

Repository program dapat dilihat pada tautan berikut:
https://github.com/trimonuter/Tucil1_IF2211Stima_13522158

Daftar Pustaka

1. Rinaldi Munir. (2024). Algoritma Greedy (Bagian 1).
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm>
2. Rinaldi Munir. (2024). Algoritma Greedy (Bagian 2).
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf)
3. Rinaldi Munir. (2024). Algoritma Greedy (Bagian 3).
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-\(2022\)-Bag3.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf)



Figure 69. A miner (that we love)
source : Cement from arknights



Figure 1337. We Love Miners Merch