IMPLEMENTASI ALGORITMA *GREEDY* PADA BOT GACOR DALAM PERMAINAN *GET STARTED WITH DIAMONDS*

Tugas Besar

Diajukan sebagai syarat menyelesaikan mata kuliah Strategi Algoritma (IF2211) Kelas RA
di Program Studi Teknik Informatika, Fakultas Teknologi Industri, Institut Teknologi Sumatera

Oleh: Kelompok 3 GACOR

Muhammad Fadhillah Akbar	123140003
Sigit Kurnia Hartawan	123140033
Ahmat Prayoga Sembiring	123140053

Dosen Pengampu: Imam Ekowicaksono, S.Si., M.Si.

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS TEKNOLOGI INDUSTRI

INSTITUT TEKNOLOGI SUMATERA

2025

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

Get Started with Diamonds adalah tantangan pemrograman kompetitif di mana bot buatan pemain saling berkompetisi. Tujuan setiap bot adalah mengumpulkan diamond sebanyak mungkin melalui strategi yang diimplementasikan pemain, dengan rintangan yang sengaja didesain kompleks untuk meningkatkan dinamika permainan. Salah satu strategi yang dapat digunakan adalah pendekatan *greedy*, di mana bot akan selalu mengambil keputusan berdasarkan keuntungan jangka pendek terbesar Komponen utama permainan:

1. Diamond
    a. Terdiri dari dua jenis: *biru* (1 poin) dan *merah* (2 poin).
    b. Diregenerasi secara berkala dengan rasio merah-biru yang berubah setiap kali.
    c. Strategi *greedy*: Bot akan prioritaskan diamond merah karena nilai lebih tinggi, dan memilih diamond terdekat untuk efisiensi waktu.
2. Red Button
    a. Menginjaknya akan me-reset semua diamond dan mengacak posisinya.
    b. Strategi *greedy*: Bot akan tekan tombol hanya jika diamond sekitar habis, untuk segera dapatkan diamond baru tanpa pertimbangan jangka panjang.
3. Teleporter
    a. Sepasang portal yang terhubung untuk pemindahan instan.
    b. Strategi *greedy*: Bot akan gunakan teleporter hanya jika menghemat langkah menuju diamond terdekat.
4. Bot dan Base
    a. Bot bergerak mengumpulkan diamond dan menyimpannya di *Bsaase* untuk konversi jadi poin.
    b. Strategi *greedy*: Bot akan langsung kembali ke base begitu inventory penuh, tanpa menunda demi diamond baru.
5. Inventory
    a. Tempat penyimpanan sementara dengan kapasitas terbatas.
    b. Strategi *greedy*: Bot akan selalu isi inventory sampai maksimum sebelum kembali ke base, tanpa menyisakan ruang untuk diamond bernilai lebih tinggi.

BAB II
LANDASAN TEORI

2.1 Dasar Teori

Algoritma *Greedy* dapat menentukan jalur mana yang akan diambil terlebih dahulu atau dapat disebut dengan jalur optimum lokal sehingga sampai seluruh jalur diambil pada akhir perjalanan dan menciptakan rute perjalanan terpendek atau disebut dengan optimum global [1]
Algoritma *greedy* adalah algoritma yang memecahkan masalah langkah demi langkah, pada setiap langkah :

a. Mengambil pilihan yang terbaik yang dapat diperoleh saat itu
b. Berharap bahwa dengan memilih optimum local pada setiap langkah akan mencapai optimum global. Algoritma *greedy* mengasumsikan bahwa optimum lokal merupakan bagian dari optimum global.

Pada setiap langkah, terdapat banyak pilihan yang perlu dieksplorasi. Oleh karena itu, pada setiap langkah harus dibuat keputusan yang terbaik dalam menentukan pilihan. Pada setiap langkahnya merupakan pilihan, untuk membuat langkah optimum local dengan harapan bahwa langkah sisanya mengarah ke solusi optimum global. Sesuai arti harafiah, *Greedy* berarti tamak. Prinsip utama dari algoritma ini adalah mengambil sebanyak mungkin apa yang dapat diperoleh sekarang. Algoritma *Greedy* dapat menentukan sebuah jalur terpendek antara node-node yang akan digunakan dengan mengambil secara terusmenerus dan menambahkannya ke dalam jalur yang akan dilewati. Atau pada notasi big-O dituliskan O(n2log2(n)). Dalam membentuk solusi, algoritma *greedy* pada setiap langkahnya akan mengambil pilihan yang merupakan optimum lokal atau pilihan yang sesuai dengan spesifikasi pembuat algoritma. Dengan pengambilan pilihan yang sesuai pada setiap langkah ini diharapkan solusi yang didapat optimum global atau sesuai keinginan pembuat algoritma. Pada setiap langkah algoritma *greedy*, kita akan mendapat optimum lokal. Bila algoritma berakhir maka diharapkan optimum lokal ini akan menjadi optimum global. Sehingga sebenarnya algoritma *greedy* mengasumsikan bahwa optimum lokal ini merupakan bagian dari optimum global [2]

2.2 Cara Kerja Program

Program ini mengimplementasikan sebuah bot yang bermain dalam permainan dengan menggunakan algoritma *greedy* untuk mengambil keputusan aksinya. Bot ini dirancang untuk mengumpulkan diamond dan berinteraksi dengan objek lain seperti tombol reset dan teleporter di papan permainan untuk mengelola waktu dan inventarisnya.

2.2.1 Cara Implementasi Program

a. Inisialisasi dan Pengaturan Objektif
   Bot menyimpan daftar objektif yang harus dicapai, seperti posisi diamond dan tombol reset. Setiap objektif direpresentasikan sebagai list [posisi, poin, prioritas], di mana :

   - Posisi adalah lokasi objektif di papan permainan
   - Poin adalah nilai atau hadiah (misalnya, poin dari mengumpulkan diamond)
   - Prioritas adalah skor yang dihitung sebagai rasio biaya-manfaat (jarak ke objektif dibagi dengan poinnya).

   Dalam metode set_list_objective, bot membuat daftar sementara objektif dari diamond di papan dan tombol reset. Untuk setiap objektif, bot menghitung pri oritasnya menggunakan metode set_priority. Metode ini menghitung prioritas dengan membagi jarak ke objektif dengan poinnya (yaitu, jarak / poin). Rasio yang lebih rendah menunjukkan bahwa objektif lebih menarik karena lebih dekat relatif terhadap hadiahnya. Jika poin adalah 0 (misalnya, untuk tombol reset), prioritas diatur ke tak hingga (float('inf')), sehingga objektif tersebut hanya dipilih jika tidak ada pilihan lain.

b. Pengurutan Objektif
   Objektif diurutkan berdasarkan skor prioritasnya (self.list_objective.sort(key=lambda e: e[2])), sehingga bot selalu menargetkan objektif dengan rasio biaya-manfaat terendah terlebih dahulu. Ini adalah pilihan *greedy* karena bot memilih opsi yang tampak terbaik pada saat itu

c. Pengambilan Keputusan Langkah Selanjutnya
   Dalam metode next_move, bot menentukan aksinya berdasarkan keadaan saat ini:
   - Jika sudah waktunya pulang (misalnya, waktu hampir habis), bot bergerak menuju basis (self.base_position).
   - Jika inventaris penuh (membawa 5 diamond), bot juga pulang untuk men gosongkan inventaris.
   - Jika tidak, bot memilih objektif dengan prioritas tertinggi dari daftarnya dan bergerak menuju posisi tersebut.

d. Penyesuaian Objektif Berdasarkan Keadaan
Bot menyesuaikan daftar objektifnya berdasarkan inventaris. Misalnya, jika sudah membawa 4 diamond, bot menghapus diamond merah (yang bernilai 2 poin) dari daftar objektifnya (filter(lambda x: x[1] != 2, self.list_objective))un tuk fokus pada objektif lain yang lebih berharga.

e. Tidak Ada Perencanaan Jangka Panjang
Bot tidak melakukan backtracking atau perencanaan jalur jangka panjang. Se tiap langkahnya adalah keputusan *greedy*: bergerak menuju target saat ini tanpa mempertimbangkan konsekuensi jangka panjang

2.2.2 Menjalankan Bot Program

Untuk menjalankan bot, program utama (main.py) mengatur lingkungan permainan, mendaftarkan bot, bergabung dengan papan permainan, dan kemudian secara beru lang memanggil metode next_move sampai permainan berakhir :

- Inisialisasi: Program membaca konfigurasi permainan, seperti papan permainan, posisi awal bot, dan objek seperti diamond dan teleporter.
- Pendaftaran dan Bergabung: Bot terdaftar dan bergabung dengan papan per mainan melalui API permainan.
- Loop Permainan: Program memanggil metode next_move dari bot secara beru lang untuk setiap giliran permainan. Metode ini mengembalikan aksi selanjutnya (misalnya, gerakan ke arah tertentu).
- Aksi Bot: Bot melakukan aksi yang dihasilkan oleh metode next_move, seperti bergerak ke sel tetangga atau menggunakan teleporter.
- Akhir Permainan: Permainan berakhir ketika waktu habis atau kondisi lain terpenuhi, dan program menghentikan loop

2.2.3 Kesimpulan

Algoritma *greedy* adalah pendekatan sederhana namun efektif untuk memecahkan masalah optimasi dengan membuat pilihan optimal lokal pada setiap langkah. Dalam konteks pro gramini, bot menggunakan algoritma *greedy* untuk memprioritaskan objektif berdasarkan rasio biaya-manfaat dan selalu memilih aksi yang paling menguntungkan secara langsung, seperti bergerak menuju objektif terdekat dengan poin tertinggi atau pulang ketika waktu hampir habis. Meskipun efektif untuk skenario sederhana, pendekatan ini mungkin tidak optimal dalam situasi kompleks di mana perencanaan jangka panjang diperlukan.

BAB III

APLIKASI STRATEGI *GREEDY*

Strategi *greedy* dalam game berbasis grid, berdasarkan analisis kode gacor.py dan main.py. Strategi *greedy* digunakan untuk mengoptimalkan pengambilan keputusan bot dalam mengumpulkan diamond dan berinteraksi dengan objek seperti tombol reset dan teleporter. Laporan ini terdiri dari empat bagian: Proses Mapping, Eksplorasi Alternatif Solusi *Greedy*, Analisis Efisiensi dan Efektivitas Solusi *Greedy*, serta Strategi *Greedy* yang Dipilih

3.1 Proses *Mapping*

Proses mapping adalah langkah awal yang dilakukan bot untuk memahami papan per mainan dan menentukan prioritas objek yang akan dikejar. Proses ini dilakukan dalam metode initialize dan set_list_objective dari kelas GacorLogic. Berikut adalah langkah langkah utama:

- Identifikasi Objek

  Bot mengumpulkan informasi tentang posisi semua objek yang dapat dikumpulkan, seperti diamond dan tombol reset, serta posisi teleporter. Setiap objek direpresentasikan sebagai list [posisi, poin, prioritas], di mana:– posisi adalah koordinat objek di papan permainan,– poin adalah nilai hadiah dari objek (misalnya, poin dari diamond),– prioritas adalah skor yang dihitung berdasarkan rasio jarak ke poin.

- Perhitungan Jarak

  Untuk setiap objek, bot menghitung jarak dari posisi saat ini ke objek tersebut menggunakan metode get_distance untuk jalur langsung dan get_distance_teleporter untuk jalur melalui teleporter. Jarak terpendek di antara keduanya dipilih untuk menentukan biaya perjalanan.

- Penetapan Prioritas

  Prioritas dihitung sebagai rasio jarak ke objek dibagi den gan poin yang didapat (jarak / poin). Rasio yang lebih rendah menunjukkan objek yang lebih menarik karena memberikan manfaat besar dengan biaya per jalanan rendah. Jika poin adalah 0 (misalnya, tombol reset), prioritas diatur ke tak hingga untuk menghindari pemilihan kecuali tidak ada pilihan lain.

- Pengurutan Objek

  Daftar objek diurutkan berdasarkan prioritas menggunakan self.list_objective.sort(key=lambda e: e[2]). Objek dengan rasio teren dah menjadi target utama bot

3.2 Eksplorasi Alternatif Solusi *Greedy*

a. *Greedy* Berdasarkan Jarak (Distance-Based *Greedy*)

   Bot selalu memilih objek terdekat dari posisi saat ini, tanpa mempertimbangkan ni lai poin. Pendekatan ini sederhana dan cepat, tetapi bisa mengabaikan objek bernilai tinggi yang sedikit lebih jauh, sehingga total poin yang dikumpulkan mungkin rendah.

b. *Greedy* Berdasarkan Poin (Point-Based *Greedy*)

   Bot memprioritaskan objek dengan nilai poin tertinggi, tanpa mempertimbangkan jarak. Ini dapat memaksimalkan poin per objek, tetapi berisiko menghabiskan waktu untuk perjalanan jauh, yang dapat menyebabkan bot kehabisan waktu.

c. *Greedy* Berdasarkan Rasio Biaya-Manfaat (Ratio-Based *Greedy*)

   Bot memilih objek berdasarkan rasio jarak ke poin (jarak / poin), seperti yang diterapkan dalam kode saat ini. Pendekatan ini menyeimbangkan jarak dan nilai poin, membuatnya lebih efisien dalam berbagai kondisi papan permainan.

d. *Greedy* dengan Kesadaran Waktu (Time-Aware *Greedy*)

   Bot mempertimbangkan waktu yang tersisa dan memprioritaskan objek yang dapat dicapai serta dikumpulkan sebelum waktu habis. Ini memerlukan estimasi waktu tempuh pergi dan pulang, yang lebih kompleks tetapi dapat mencegah bot terjebak jauh dari basis.

e. *Greedy* dengan Kesadaran Inventaris (Inventory-Aware *Greedy*)

   Bot menyesuaikan prioritas objek berdasarkan kapasitas inventaris. Misalnya, jika inventaris hampir penuh, bot menghindari objek bernilai rendah untuk fokus pada objek berharga atau kembali ke basis.

f. *Greedy* dengan Pemanfaatan Teleporter (Teleporter-Heavy *Greedy*)

   Bot secara agresif menggunakan teleporter untuk memperpendek jalur, bahkan jika jalur langsung sedikit lebih pendek dalam langkah. Ini dapat menghemat waktu tetapi berisiko jika teleporter tidak digunakan dengan bijak.

3.3 Analisis Efisiensi dan Efektivitas Solusi *Greedy*

Untuk mengevaluasi alternatif solusi *greedy*, kita mempertimbangkan dua aspek: efisiensi (kompleksitas komputasi dan kecepatan pengambilan keputusan) dan efektivitas (kemampuan memaksimalkan poin yang dikumpulkan)

a. Efisiensi

Semua strategi *greedy* relatif efisien karena hanya melibatkan perhitungan jarak dan pengurutan list. Kompleksitas komputasi utama adalah :

- Perhitungan jarak: O(n) untuk setiap objek, di mana n adalah jumlah objek.
- Pengurutan: O(nlogn) untuk mengurutkan daftar objek berdasarkan priori tas.
- Distance-Based dan Point-Based *Greedy*: Paling sederhana, hanya men gurutkan berdasarkan satu faktor (jarak atau poin).
- Ratio-Based *Greedy*: Sedikit lebih kompleks karena menghitung rasio, tetapi tetap efisien.
- Time-Aware *Greedy*: Memerlukan estimasi waktu tempuh, yang menam bah kompleksitas kecil.
- Inventory-Aware *Greedy*: Memerlukan pembaruan dinamis daftar objek, tetapi tetap dalam batas efisiensi.
- Teleporter-Heavy *Greedy*: Menambah perhitungan jarak melalui tele porter, tetapi masih terjangkau.

b. Efektivitas

- Distance-Based *Greedy*: Efektif pada papan padat dengan banyak objek dekat, tetapi kurang optimal jika objek bernilai tinggi jauh.
- Point-Based *Greedy*: Efektif jika waktu tidak terbatas dan objek bernilai tinggi mudah dijangkau, tetapi berisiko jika waktu terbatas.
- Ratio-Based *Greedy*: Menawarkan keseimbangan yang baik, efektif dalam berbagai kondisi karena mempertimbangkan jarak dan poin.
- Time-Aware *Greedy*: Berpotensi paling efektif karena memastikan bot kembali ke basis tepat waktu, tetapi memerlukan estimasi akurat.
- Inventory-Aware *Greedy*: Penting untuk mengoptimalkan pengumpulan saat inventaris terbatas, mencegah pemborosan gerakan.

- Teleporter-Heavy *Greedy*: Efektif jika teleporter ditempatkan strategis, tetapi bisa suboptimal jika jalur teleporter lebih panjang

3.4 Strategi *Greedy* yang Dipilih

a. Dasar Pemilihan

   Bot memprioritaskan objek berdasarkan rasio jarak ke poin (jarak / poin), memilih objek dengan rasio terendah untuk memaksimalkan man faat per unit jarak. Ini memastikan bot tidak hanya mengejar objek dekat atau bernilai tinggi, tetapi yang memberikan keseimbangan terbaik.

b. Kesadaran Inventaris

   Bot menyesuaikan daftar objek berdasarkan kapasitas inventaris. Misalnya, jika sudah membawa 4 diamond, bot menghapus diamond merah (nilai 2 poin) dari daftar prioritas untuk fokus pada objek bernilai lebih tinggi atau kembali ke basis.

c. Pemanfaatan Teleporter

   Bot membandingkan jarak langsung dengan jarak melalui teleporter menggunakan metode get_distance dan get_distance_teleporter, lalu memilih jalur terpendek. Ini meningkatkan efisiensi pergerakan.

d. Manajemen Waktu

   Bot memiliki mekanisme time_to_go_home untuk memas tikan kembali ke basis sebelum waktu habis, dengan mempertimbangkan jarak langsung dan melalui teleporter

BAB IV
IMPLEMENTASI DAN PENGUJIAN

Bab ini bertujuan untuk memaparkan secara rinci implementasi algoritma *greedy* yang diterapkan pada *bot* Gacor. Pembahasan mencakup detail pseudocode, struktur data yang digunakan, serta analisis mendalam mengenai desain solusi *greedy* melalui berbagai skenario pengujian. Analisis ini akan mengevaluasi apakah strategi yang diimplementasikan berhasil mencapai hasil optimal dan mengidentifikasi kondisi-kondisi di mana strategi tersebut mungkin tidak memberikan solusi terbaik.

4.1 Implementasi Algoritma *Greedy* pada Program Bot Gacor

Pengembangan Gacor didasarkan pada penerapan strategi *greedy* yang fokus pada pengambilan keputusan lokal optimal di setiap langkah permainan. Tujuannya adalah memaksimalkan perolehan skor dengan cara seefisien mungkin.

4.1.1 Prinsip Desain dan Strategi Inti Gacor

Strategi inti Gacor dibangun di atas beberapa pilar utama:

1. Prioritas Target Berbasis Rasio Jarak-Poin
   Bot memilih target (diamond atau *red button*) dengan rasio terkecil antara estimasi jarak tempuh efektif dan poin yang akan diperoleh. Jarak efektif mempertimbangkan rute langsung dan rute melalui teleporter.
2. Manajemen Kondisi Kritis
   Bot secara otomatis memprioritaskan kembali ke *base* jika inventaris penuh atau jika sisa waktu permainan dianggap tidak cukup untuk tindakan lain selain kembali ke *base*.
3. Pemanfaatan Fitur Permainan
   Teleporter secara aktif dipertimbangkan untuk mempersingkat perjalanan, dan *red button* diberi nilai artifisial untuk dipertimbangkan sebagai target potensial.
4. Adaptasi Dinamis
   Keputusan selalu didasarkan pada kondisi terkini *board* permainan, yang dievaluasi ulang pada setiap giliran.

4.1.2 Pseudocode Detail GacorLogic

Berikut adalah pseudocode yang mengilustrasikan logika operasional GacorLogic, dirancang agar mudah dipahami alurnya:

```
# Import library standar Python
import time  # Digunakan untuk menghitung waktu eksekusi
import random  # Digunakan jika bot harus bergerak secara acak

# Import tipe data untuk penjelasan tipe variabel (type hinting)
from typing import Optional, List, Tuple
```

```python
# Import dari game engine
from game.logic.base import BaseLogic  # Kelas dasar semua bot
from game.models import GameObject, Board, Position  # Model utama dalam game
from ..util import get_direction  # Fungsi bantu untuk menentukan arah
gerakan

# GACORLOGIC adalah bot dengan strategi greedy berbasis jarak dan poin
class GacorLogic(BaseLogic):
    def __init__(self):
        # Inisialisasi variabel internal bot
        self.goal_position: Optional[Position] = None  # Posisi target utama
(bisa None jika tidak ada)
        self.is_returning_to_base: bool = False  # Menandakan apakah bot
sedang dalam perjalanan pulang
        self.list_objective = []  # Daftar target (diamond, tombol) dengan
format: [posisi, poin, rasio prioritas]
        self.target_position: Optional[Position] = None  # Posisi target
aktif saat ini
        self.teleporter = []  # Daftar teleporter (format: [(posisi,
dummy_val)])
        self.base_position: Optional[Position] = None  # Posisi base (tempat
kembali)
        self.current_position: Optional[Position] = None  # Posisi bot saat
ini
        self.is_teleporter_move = False  # Menandakan apakah barusan
menggunakan teleporter
        self.start_time = None  # Waktu mulai (opsional, untuk logging)

    def initialize(self, board_bot: GameObject, board: Board):
        # Update posisi penting saat ini dari board dan bot
        self.current_position = board_bot.position
        self.base_position = board_bot.properties.base
        # Ambil semua teleporter di board
        self.teleporter = [(tp.position, 0) for tp in board.game_objects if
tp.type == "TeleportGameObject"]
        # Update daftar target
        self.set_list_objective(board)

    def get_distance(self, a: Position, b: Position) -> int:
        # Menghitung jarak Manhattan antara dua titik (tanpa diagonal)
        return abs(a.x - b.x) + abs(a.y - b.y)

    def get_distance_teleporter(self, a: Position, b: Position) -> int:
        # Menghitung jarak a → teleporter1 → teleporter2 → b
```

```
        if len(self.teleporter) < 2:
            return 999  # Teleporter tidak tersedia atau tidak cukup, beri
nilai besar
        return self.get_distance(a, self.teleporter[0][0]) +
self.get_distance(self.teleporter[1][0], b)

    def set_priority(self, obj) -> List:
        # Hitung prioritas berdasarkan rasio (jarak / poin)
        distance = min(
            self.get_distance(self.current_position, obj[0]),  # Jarak
langsung
            self.get_distance_teleporter(self.current_position, obj[0])  #
Jarak lewat teleporter
        )
        obj[2] = distance / obj[1] if obj[1] != 0 else float('inf')  # Rasio:
makin kecil makin prioritas
        return obj

    def time_to_go_home(self, board_bot: GameObject) -> bool:
        # Mengecek apakah sudah saatnya pulang (berdasarkan waktu sisa)
        time_left = board_bot.properties.milliseconds_left // 1000  # Waktu
tersisa dalam detik
        dist_normal = self.get_distance(self.current_position,
self.base_position)
        dist_teleporter = self.get_distance_teleporter(self.current_position,
self.base_position)

        # Jika jarak + buffer waktu (2 detik) lebih besar dari waktu sisa,
maka pulang
        if dist_teleporter < dist_normal:
            return dist_teleporter + 2 >= time_left
        else:
            return dist_normal + 2 >= time_left

    def set_list_objective(self, board: Board):
        # Mengambil semua diamond dan tombol reset yang bisa dijadikan target
        temp_objective = [[x.position, x.properties.points, 0] for x in
board.diamonds]
        reset_buttons = [x for x in board.game_objects if x.type ==
"DiamondButtonGameObject"]
        if reset_buttons:
            # Tambahkan tombol reset ke dalam daftar target dengan nilai poin
0.75
            temp_objective.append([reset_buttons[0].position, 0.75, 0])
        # Hitung prioritas semua target
```

```
        self.list_objective = list(map(self.set_priority, temp_objective))
        # Urutkan berdasarkan prioritas terkecil (rasio jarak/poin)
        self.list_objective.sort(key=lambda e: e[2])

    def should_use_teleporter(self, target: Position) -> bool:
        # Menentukan apakah harus menggunakan teleporter untuk menuju target
        if len(self.teleporter) < 2:
            return False  # Tidak ada cukup teleporter

        dist_direct = self.get_distance(self.current_position, target)
        dist_tp = self.get_distance_teleporter(self.current_position, target)

        return dist_tp + 1 < dist_direct  # Gunakan teleporter jika secara
signifikan lebih cepat

    def next_move(self, board_bot: GameObject, board: Board) -> Tuple[int,
int]:
        # Fungsi utama yang menentukan langkah bot di setiap giliran

        time_start = time.time()  # Logging waktu mulai
        self.initialize(board_bot, board)  # Update posisi dan target

        if self.is_teleporter_move:
            self.start_time = time.time()  # Catat waktu teleportasi
            self.is_teleporter_move = False

        # Cetak informasi penting (debug)
        print("bot time remaining:", board_bot.properties.milliseconds_left)
        print("current position = ", self.current_position)
        print("inventory: ", board_bot.properties.diamonds)

        # Keputusan strategis: pulang jika waktu hampir habis atau inventory
penuh
        if self.time_to_go_home(board_bot):
            print("=== TIME TO GO HOME")
            self.target_position = self.base_position
        elif board_bot.properties.diamonds == 5:
            print("=== INVENTORY PENUH")
            self.target_position = self.base_position
        else:
            if board_bot.properties.diamonds == 4:
                # Hindari diamond merah (poin 2) jika sudah punya 4 diamond
                print("== DIAMOND MERAH DIHAPUS")
                self.list_objective = list(filter(lambda x: x[1] != 2,
self.list_objective))
```

```
            # Ambil target terbaik (prioritas tertinggi = rasio terkecil)
            if self.list_objective:
                self.target_position = self.list_objective[0][0]
            else:
                self.target_position = None  # Tidak ada target

        # Debugging
        print("Teleporter position:", self.teleporter)
        print("Target position:", self.target_position)

        # Menentukan arah gerakan
        if self.target_position:
            if self.should_use_teleporter(self.target_position):
                # Arahkan ke teleporter pertama jika lebih cepat
                tp_entry = self.teleporter[0][0]
                print("USING TELEPORTER to reach target")
                delta_x, delta_y = get_direction(
                    self.current_position.x,
                    self.current_position.y,
                    tp_entry.x,
                    tp_entry.y
                )
                if self.current_position == tp_entry:
                    self.is_teleporter_move = True  # Tandai bahwa
teleportasi sedang terjadi
            else:
                # Gerak langsung ke target
                delta_x, delta_y = get_direction(
                    self.current_position.x,
                    self.current_position.y,
                    self.target_position.x,
                    self.target_position.y
                )
        else:
            # Jika tidak ada target, gerak acak
            delta_x, delta_y = random.choice([(1, 0), (0, 1), (-1, 0), (0, -
1)])

        print("Elapsed Time:", time.time() - time_start)  # Waktu proses 1
langkah bot
        return delta_x, delta_y  # Kembalikan gerakan yang akan dijalankan
```

4.1.3 Penjelasan Komponen Kunci Implementasi GacorLogic

a. initialize(): Metode ini memastikan *bot* selalu memiliki data terbaru tentang posisinya, posisi *base*, lokasi teleporter, dan daftar target yang relevan sebelum membuat keputusan.

b. calculate_priority_ratio() (dalam set_priority di kode Python): Merupakan jantung strategi *greedy*. Fungsi ini mengevaluasi setiap objek potensial (diamond, *red button*) dengan menghitung rasio antara jarak efektif untuk mencapainya (mempertimbangkan rute langsung dan via teleporter) dan poin yang ditawarkan. Semakin kecil rasio ini, semakin tinggi prioritas objek tersebut.

c. set_list_objective(): Mengumpulkan semua diamond dan *red button* yang ada, kemudian menggunakan calculate_priority_ratio untuk menilai masing-masing. Hasilnya adalah daftar target yang sudah terurut, memungkinkan *bot* dengan mudah memilih target "terbaik" saat ini.

d. is_time_to_return_to_base() (dalam time_to_go_home di kode Python): Fungsi krusial untuk manajemen risiko. *Bot* memperkirakan waktu yang dibutuhkan untuk kembali ke *base* (termasuk *buffer*) dan membandingkannya dengan sisa waktu permainan. Jika waktu kritis, *bot* akan memprioritaskan pulang untuk mengamankan skor.

e. should_bot_use_teleporter(): Meningkatkan efisiensi pergerakan dengan mengevaluasi apakah penggunaan teleporter akan mempersingkat waktu tempuh ke target secara signifikan. Ini mencegah penggunaan teleporter yang tidak efisien.

f. Logika Keputusan dalam next_move():

   1. Prioritas Kondisi Mendesak: Pemeriksaan untuk kembali ke *base* (karena waktu atau inventaris penuh) dilakukan pertama kali. Jika salah satu kondisi terpenuhi, semua target lain diabaikan.
   2. Pemilihan Target Objektif: Jika tidak ada kondisi mendesak, *bot* memilih target dari list_objective yang sudah diprioritaskan. Terdapat penanganan khusus jika inventaris berisi 4 diamond untuk menghindari diamond merah, memaksimalkan potensi pengisian slot terakhir. Penentuan Arah Gerakan: Setelah target final ditentukan (bisa jadi *base*, diamond, *red button*, atau pintu masuk teleporter), fungsi get_direction digunakan untuk menghitung langkah selanjutnya. Jika tidak ada target sama sekali, *bot* bergerak acak.

4.2 Struktur Data yang Digunakan dalam Program Bot GacorLogic

Pemilihan struktur data dalam GacorLogic bertujuan untuk mendukung efisiensi dan kesederhanaan logika *greedy*:

1. self.current_position: Position
   - Tipe: Objek Position (menyimpan koordinat x, y).
   - Deskripsi: Menyimpan posisi aktual *bot* di *grid* permainan pada setiap giliran. Menjadi titik referensi utama untuk semua perhitungan jarak.

2. self. base_position: Position
   - Tipe: Objek Position.
   - Deskripsi: Menyimpan koordinat *home base* milik *bot*. Digunakan sebagai target ketika *bot* memutuskan untuk kembali menyimpan diamond.

3. self.teleporter: List[Position]
   - Tipe: Daftar (list) dari objek Position. (Dalam kode Python Anda, List[Tuple[Position, int]], namun integer kedua tidak digunakan).
   - Deskripsi: Menyimpan posisi dari semua objek teleporter yang aktif di *board*. Digunakan untuk menghitung jarak alternatif dan memutuskan penggunaan teleporter.

4. self.list_objective: List[Objective]
   - Tipe: Daftar (list) dari objek/struktur Objective. Setiap Objective idealnya berisi position: Position, points: Float, dan ratio: Float. (Dalam kode Python Anda, ini adalah List[List] dengan format [Position, poin, rasio]).
   - Deskripsi: Struktur data inti untuk strategi *greedy*. Berisi semua target potensial (diamond dan *red button*) yang telah dievaluasi dan diurutkan berdasarkan rasio prioritasnya. Target dengan rasio terkecil (paling diinginkan) berada di awal daftar.

5. self.target_position: Optional[Position]
   - Tipe: Objek Position atau None.

- Deskripsi: Menyimpan koordinat target akhir yang telah diputuskan oleh *bot* untuk dikejar pada giliran tersebut. Bisa berupa posisi diamond, *red button*, *base*, atau None jika tidak ada target yang dipilih.

Variabel boolean seperti is_returning_to_base atau is_teleporter_move dalam kode Python Anda lebih berfungsi sebagai *flag* status sementara dalam satu siklus next_move atau untuk *logging*, dan keputusan utama lebih banyak bergantung pada evaluasi kondisi secara langsung. Struktur data yang ada telah memadai untuk kebutuhan algoritma *greedy* yang diterapkan, yang tidak memerlukan penyimpanan histori atau status yang kompleks.

4.3 Analisis Desain Solusi Algoritma *Greedy* Melalui Skenario Pengujian

Bagian ini menganalisis desain solusi algoritma *greedy* GacorLogic dengan memprediksi perilakunya dalam berbagai skenario permainan yang unik. Tujuannya adalah untuk menilai potensi keberhasilan dalam mencapai nilai optimal dan mengidentifikasi kondisi di mana strategi ini mungkin kurang efektif. Pendahuluan untuk Analisis Skenario Strategi *greedy* GacorLogic dirancang untuk membuat pilihan yang tampak terbaik pada setiap langkah berdasarkan informasi lokal. Analisis berikut akan mengeksplorasi bagaimana pendekatan ini bekerja dalam situasi permainan yang beragam.

4.3.1 Skenario Pengujian

Skenario Pengujian dan Analisis Kinerja:

a) Skenario 1: Papan Permainan dengan Distribusi Diamond Merata, Minim Interaksi Lawan
   - Deskripsi Skenario: Sejumlah diamond biru dan merah tersebar relatif merata di seluruh papan. Bot lawan sedikit atau tidak aktif di sekitar GacorLogic.
   - Prediksi Perilaku GacorLogic:
     1. GacorLogic akan menghitung rasio jarak/poin untuk semua diamond yang terlihat.
     2. Akan bergerak menuju diamond dengan rasio terkecil, memanfaatkan teleporter jika should_bot_use_teleporter mengindikasikan itu lebih efisien.
     3. Secara berulang akan mengambil diamond terdekat/terbaik secara lokal, kembali ke *base* saat inventaris penuh atau waktu kritis.
   - Analisis Kinerja dan Optimalitas:
     1. Keberhasilan: Cukup berhasil dalam mengumpulkan poin secara konsisten dan cepat karena fokus pada efisiensi lokal.
     2. Optimalitas: Kemungkinan besar tidak optimal secara global. GacorLogic mungkin mengambil serangkaian diamond yang mudah dijangkau namun total poinnya lebih rendah dibandingkan rute lain yang mungkin memerlukan sedikit pengorbanan di awal untuk mencapai *cluster* diamond bernilai lebih tinggi.

3. Kondisi Tidak Optimal: Ketika ada kelompok diamond bernilai tinggi yang sedikit lebih jauh, namun pilihan *greedy* lokal mengarahkan *bot* ke diamond tunggal yang lebih dekat namun bernilai lebih rendah, sehingga melewatkan kesempatan skor yang lebih besar dalam jangka waktu yang sama.

b) Skenario 2: Manajemen Inventaris Mendekati Penuh (4 dari 5 Diamond)
   - Deskripsi Skenario: GacorLogic telah mengumpulkan 4 diamond. Terdapat pilihan diamond biru dan merah di papan.
   - Prediksi Perilaku GacorLogic:
     1. Strategi khusus akan aktif: list_objective akan disaring untuk mengabaikan diamond merah (poin 2).
     2. GacorLogic akan memprioritaskan diamond biru (poin 1) terdekat/terbaik berdasarkan rasio, untuk memaksimalkan kapasitas inventaris menjadi 5.
     3. Jika tidak ada diamond biru yang layak, dan kondisi lain tidak memaksa pulang, ia mungkin akan mempertimbangkan *red button* atau bergerak acak.
   - Analisis Kinerja dan Optimalitas:
     1. Keberhasilan: Berhasil dalam upaya memaksimalkan penggunaan slot inventaris menjadi 5 jika diamond biru tersedia.
     2. Optimalitas: Optimal dalam konteks memaksimalkan jumlah diamond per trip jika diamond biru ada.
     3. Kondisi Tidak Optimal: Jika hanya tersisa diamond merah di papan dan *bot* memiliki 4 diamond. *Bot* akan mengabaikan diamond merah tersebut. Jika waktu masih banyak, ini sub-optimal dibandingkan kembali ke *base* dengan 4 diamond atau mengambil risiko mengambil diamond merah jika itu satu-satunya pilihan sebelum waktu habis (logika saat ini tidak mendukung pengambilan risiko ini jika ada diamond merah).

c) Skenario 3: Kondisi Waktu Kritis (Permainan Akan Segera Berakhir)
   - Deskripsi Skenario: Sisa waktu permainan sangat terbatas.
   - Prediksi Perilaku GacorLogic:
     1. Fungsi is_time_to_return_to_base akan mengembalikan True.
     2. target_position akan diatur secara paksa ke base_position.
     3. *Bot* akan bergerak lurus menuju *base*, mempertimbangkan rute tercepat (termasuk via teleporter jika lebih efisien untuk mencapai *base*).
   - Analisis Kinerja dan Optimalitas:
     1. Keberhasilan: Sangat berhasil dalam mengamankan poin yang sudah ada di inventaris.
     2. Optimalitas: Ya, ini adalah perilaku optimal dalam situasi waktu kritis untuk mencegah kehilangan poin.

3. Kondisi Tidak Optimal: Tidak ada, selama estimasi REQUIRED_TIME_BUFFER dalam is_time_to_return_to_base dikalibrasi dengan baik. Jika *buffer* terlalu besar, *bot* pulang terlalu dini; jika terlalu kecil, berisiko tidak sampai.

d) Skenario 4: Papan Permainan Sepi (Sedikit Diamond, Jarak Jauh, atau Hanya Ada Red Button)
   - Deskripsi Skenario: Jumlah diamond di papan sedikit dan/atau jaraknya sangat jauh. Mungkin hanya *red button* yang terlihat sebagai objek interaktif.
   - Prediksi Perilaku GacorLogic:
     1. Jika ada diamond (meskipun jauh), GacorLogic tetap akan menghitung rasionya. Jika itu satu-satunya pilihan, ia akan mengejarnya.
     2. Jika *red button* menawarkan rasio yang lebih baik (berdasarkan poin artifisial 0.75) dibandingkan diamond yang sangat jauh atau tidak ada diamond, *bot* mungkin akan menuju *red button*.
     3. Jika list_objective kosong (setelah filter jika ada) dan tidak ada kondisi pulang, *bot* akan bergerak acak.
   - Analisis Kinerja dan Optimalitas:
     1. Keberhasilan: Hasilnya sangat bergantung pada situasi. Mengejar diamond jauh bisa memakan banyak waktu. Menginjak *red button* bersifat spekulatif – bisa menghasilkan papan yang lebih baik atau lebih buruk.
     2. Optimalitas: Sulit ditentukan. Bergerak acak adalah upaya terakhir dan jarang optimal.
     3. Kondisi Tidak Optimal: Jika *red button* menghasilkan konfigurasi diamond yang lebih buruk. Jika gerakan acak menjauhkan *bot* dari diamond yang baru muncul atau area strategis.

e) Skenario 5: Penggunaan Teleporter Secara Strategis
   - Deskripsi Skenario: Terdapat target diamond yang jauh, namun ada pasangan teleporter yang dapat mempersingkat perjalanan secara signifikan.
   - Prediksi Perilaku GacorLogic:
     1. calculate_priority_ratio akan menghitung jarak efektif via teleporter.
     2. should_bot_use_teleporter akan menentukan apakah jalur via teleporter (dengan mempertimbangkan TELEPORTER_ADVANTAGE_THRESHOLD) lebih menguntungkan.
     3. Jika ya, actual_move_destination akan diatur ke pintu masuk teleporter terdekat yang relevan.
   - Analisis Kinerja dan Optimalitas:
     1. Keberhasilan: Berhasil meningkatkan mobilitas dan efisiensi dalam mencapai target jauh.

2. Optimalitas: Optimal secara lokal untuk mencapai satu target tersebut. TELEPORTER_ADVANTAGE_THRESHOLD membantu menghindari penggunaan teleporter untuk penghematan jarak yang minimal.
3. Kondisi Tidak Optimal: Jika penggunaan teleporter membawa *bot* ke area papan yang terisolasi atau jauh dari *cluster* diamond berikutnya, meskipun optimal untuk target saat itu, mungkin kurang optimal untuk strategi keseluruhan multi-langkah (yang tidak dimiliki GacorLogic).

f) Skenario 6: Implikasi Tanpa Logika Lawan Eksplisit
- Deskripsi Skenario: Terdapat *bot* lawan lain yang aktif di papan, mungkin mengincar diamond yang sama atau berpotensi melakukan *tackle*.
- Prediksi Perilaku GacorLogic:
  1. GacorLogic akan tetap beroperasi seolah-olah ia sendirian, fokus pada perhitungan rasio jarak/poin untuk diamond dan *red button*.
  2. Tidak ada mekanisme menghindar, memblokir, atau secara khusus bersaing dengan lawan.
- Analisis Kinerja dan Optimalitas:
  1. Keberhasilan: Sangat rendah dalam menghadapi lawan yang cerdas atau agresif.
  2. Optimalitas: Jelas tidak optimal. Mengabaikan lawan adalah kelemahan fatal dalam permainan kompetitif.
  3. Kondisi Tidak Optimal: Setiap situasi di mana ada interaksi atau potensi interaksi dengan *bot* lawan. GacorLogic rentan di-*tackle* (kehilangan semua diamond di inventaris), atau diamond yang dituju diambil lebih dulu oleh lawan.

4.3.2 Ringkasan Kekuatan dan Kelemahan Desain *Greedy* GacorLogic

Berdasarkan implementasi dan analisis skenario di atas, desain solusi algoritma *greedy* GacorLogic memiliki kekuatan dan kelemahan sebagai berikut:

Kekuatan:

a) Kecepatan Pengambilan Keputusan: Algoritma *greedy* secara inheren cepat karena tidak melakukan eksplorasi mendalam, hanya mengevaluasi opsi berdasarkan kondisi saat ini.

b) Efisiensi Lokal: Baik dalam menemukan solusi yang "cukup baik" dengan cepat untuk target-target individual, terutama dengan adanya perhitungan rasio jarak-poin dan pemanfaatan teleporter.

c) Adaptif terhadap Perubahan Langsung: Dengan menginisialisasi ulang status dan daftar objektif di setiap langkah, *bot* dapat merespons perubahan dinamis di papan.

d) Manajemen Risiko Dasar: Implementasi is_time_to_return_to_base dan logika inventaris penuh adalah bentuk manajemen risiko yang penting untuk mengamankan skor.

Kelemahan:

- Tidak Ada Perencanaan Global atau Jangka Panjang: Pilihan selalu bersifat lokal optimal, yang tidak menjamin solusi global optimal. *Bot* tidak dapat "melihat" beberapa langkah ke depan untuk merencanakan rute yang lebih strategis.
- Tidak Ada Kesadaran Terhadap Lawan: Ini adalah batasan paling signifikan. *Bot* tidak dapat menghindar, memprediksi, atau bereaksi terhadap tindakan *bot* lawan, membuatnya sangat rentan dalam permainan kompetitif.
- Potensi Terjebak di Lokal Optimum: *Bot* dapat terus menerus mengambil pilihan yang baik secara lokal, namun melewatkan peluang yang jauh lebih besar yang mungkin memerlukan investasi langkah awal yang kurang menarik menurut kriteria *greedy* saat itu.
- Penanganan *Red Button* yang Spekulatif: Meskipun diberi nilai, keputusan untuk mengambil *red button* masih bersifat untung-untungan karena hasilnya tidak dapat diprediksi.

Kesimpulannya, GacorLogic adalah implementasi yang solid dari algoritma *greedy* untuk tugas pengumpulan diamond dengan beberapa heuristik cerdas. Namun, kinerjanya dalam kompetisi nyata akan sangat dipengaruhi oleh kemampuannya mengatasi kelemahan-kelemahan tersebut, terutama yang berkaitan dengan interaksi lawan dan perencanaan strategis.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Bot GACOR menggunakan strategi *greedy* untuk membuat navigasi papan permainan, mengumpulkan diamond dan objek lain seperti tombolreset,mengelola batasan waktu dan kapasitas inventaris. Strategi ini memprioritaskan objek berdasarkan rasio jarak terhadap poin (jarak / poin), memastikan bot memilih target yang memberikan manfaat maksimal dengan biaya perjalanan minimal.Bot juga memanfaatkan teleporter untuk memperpendek jalur perjalanan dan memiliki mekanisme untuk kembali ke base ketika waktu hampir habis atau inventaris penuh (mencapai 5 diamond). Selain itu, bot ini juga menyesuaikan prioritas objek secara dinamis, seperti menghindari diamond merah (nilai 2 poin) ketika inventaris hampir penuh.

Main utama pada program ini berfungsi sebagai entry point untuk menjalankan bot, menangani pendaftaran bot ke server permainan, konfigurasi parameter (seperti token, nama, email, dan ID papan), dan mengelola loop permainan. Dalam loop ini, program terus memperbarui state papan permainan, memanggil metode next_move dari bot untuk menentukan langkah selanjutnya, dan mengeksekusi langkah tersebut melalui API permainan.

Secara keseluruhan, strategi *greedy* yang digunakan pada bot ini sangat efektif untuk skenario permainan di mana keputusan cukup untuk memaksimalkan poin dalam waktu terbatas. Namun, pendekatan ini memiliki keterbatasan dalam situasi kompleks yang memerlukan perencanaan jangka panjang, seperti ketika distribusi diamond.Meskipun demikian,bot ini menunjukkan efisiensi dalam pengambilan keputusan dan kemampuan untuk beradaptasi dengan batasan waktu dan inventaris

5.2 Saran

Untuk meningkatkan performa dan fleksibilitas bot,terdapat saran dan pengembangan pada game ini yaitu:

a. Implementasi Pathfinding Advanced

   Strategi saat ini menggunakan jarak Manhattan untuk menghitung jalur, yang tidak mempertimbangkan rintangan seperti bot lain atau area yang tidak dapat dilewati. Mengimplementasikan algoritma pathfinding seperti A* atau Dijkstra dapat membantu bot menemukan jalur terpendek yang lebih akurat, terutama dalam papan permainan yang kompleks. Algoritma ini dapat mempertimbangkan bobot tambahan, seperti kepadatan bot lawan di area tertentu.

b. Prioritas Objektif Dinamis

   Saat ini, bot menggunakan rasio jarak/poin untuk menentukan prioritas objek. Untuk meningkatkan adaptabilitas, bot dapat mempertimbangkan faktor dinamis seperti:

   - Jumlah diamond yang tersisa di papan.
   - Waktu tersisa dalam permainan.
   - Keadaan inventaris (misalnya, lebih memprioritaskan diamond bernilai tinggi saat inventaris hampir penuh).

   Dengan pendekatan ini, bot dapat membuat keputusan yang lebih kontekstual dan meningkatkan total poin yang dikumpulkan.

c. Manajemen Inventaris yang Lebih Baik

   Bot saat ini hanya menghindari diamond merah ketika inventaris mencapai 4 diamond. Sistem manajemen inventaris dapat ditingkatkan dengan:

   - Mengoptimalkan urutan pengumpulan diamond berdasarkan nilai dan jarak ke base.
   - Mempertimbangkan biaya waktu untuk kembali ke base saat menentukan apakah akan mengumpulkan diamond bernilai rendah sehingga akan meminimalkan waktu yang terbuang untuk perjalanan bolak-balik ke base.

d. Strategi Teleporter yang Lebih Optimal

   Bot saat ini menggunakan teleporter jika jalur melalui teleporter lebih pendek dari jalur langsung. Strategi ini dapat ditingkatkan dengan :

   1. Mempertimbangkan penggunaan teleporter berantai (jika ada beberapa tele-porter yang saling terhubung).
   2. Menghindari teleporter di area yang padat atau berisiko (misalnya, dekat bot lawan) sehingga akan meningkatkan efisiensi pergerakan bot.

e. Koordinasi Multi-Bot

   Jika permainan memungkinkan beberapa bot dari tim yang sama, bot dapat ditingkatkan dengan strategi koordinasi, seperti:

   1. Membagi area papan untuk menghindari tumpang tindih dalam pengumpulan diamond.
   2. Berbagi informasi tentang lokasi diamond atau ancaman dari bot lawan. Ini akan meningkatkan efisiensi tim secara keseluruhan.

f. Optimasi Kinerja

   1. Perhitungan jarak yang berulang untuk setiap objek.
   2. Pengurutan daftar objektif yang memakan waktu.
   3. Optimasi seperti caching hasil perhitungan jarak atau mengurangi frekuensi pengurutan dapat meningkatkan kecepatan pengambilan keputusan.

g. Pengujian dan Simulasi

   1. Papan dengan distribusi diamond yang berbeda (padat, jarang, atau acak).
   2. Permainan dengan waktu sangat terbatas.
   3. Interaksi dengan bot lawan yang agresif atau strategis.

LAMPIRAN

A. Repository Github

https://github.com/Ahmatsembiring/Tubes1_GACOR

B. Video Penjelasan

https://drive.google.com/drive/folders/14Die3mPnjeew3kP9vv1jIYTvKi6x_gwW

DAFTAR PUSTAKA

[1] H. F. d. Y. Safitri, "Rancang Bangun aplikasi pencarian wisata kota bogor dengan *greedy* berbasis android," *jurnal tekno mandiri,* vol. XI, 2014.

[2] E. Y. Hayati, "Pencarian Rute Terpendek Menggunakan Algoritma *greedy*," *Seminar Nasional IENACO,* pp. 391-397, 2014.