IMPLEMENTASI ALGORITMA GREEDY DALAM PEMECAHAN BOT PERMAINAN DIAMOND

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: Andri

Vebri Yanti	123140056
Doni Agus Setiawan	123140009
Handialrizky	123140012

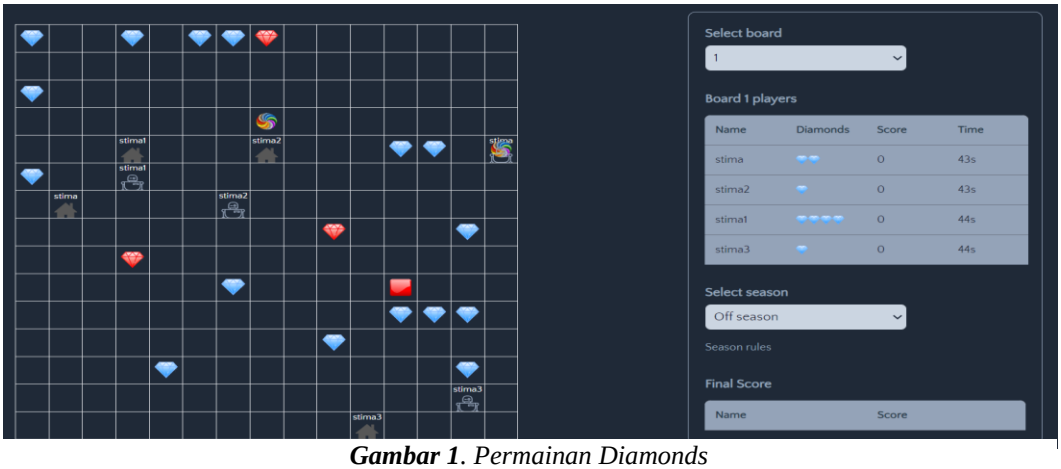
Dosen Pengampu: Imam Eko Wicaksono, S.Si., M.Si.

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INDUSTRI
INSTITUT TEKNOLOGI SUMATERA
2025

DAFTAR ISI

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. *Permainan Diamonds*

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

Program permainan Diamonds terdiri atas:

1. *Game engine*, yang secara umum berisi:
    a. Kode *backend* permainan, yang berisi *logic* permainan secara keseluruhan serta API yang disediakan untuk berkomunikasi dengan *frontend* dan program bot
    b. Kode *frontend* permainan, yang berfungsi untuk memvisualisasikan permainan
2. *Bot starter pack*, yang secara umum berisi:

a. Program untuk memanggil API yang tersedia pada *backend*
b. Program *bot logic* (bagian ini yang akan kalian implementasikan dengan algoritma *greedy* untuk bot kelompok kalian)
c. Program utama (*main*) dan utilitas lainnya

Terdapat pula komponen-komponen dari permainan Diamonds antara lain, yakni sebagai berikut:

1. Diamonds

Gambar 2*.Diamond Biru dan Merah*

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. RedButton/Diamond Button

Gambar 3*. Red/Diamond Button*

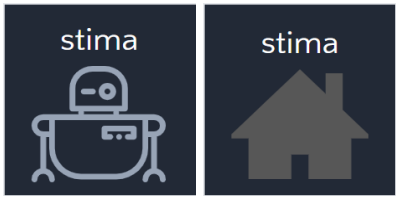
Ketika red button ini dilewati/dilangkahi, semua diamond (termasuk red diamond) akan di-generate kembali pada board dengan posisi acak. Posisi red button ini juga akan berubah secara acak jika red button ini dilangkahi.

3. Teleporters

Gambar 4*. Teleporters*

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

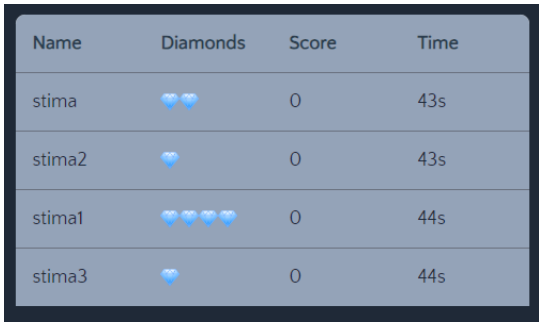
4. Bots and Bases



Gambar 5. *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. Posisi Base tidak akan berubah sampai akhir game.

5. Inventory



Gambar 6. *Inventory*

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

BAB II
LANDASAN TEORI

2.1 Dasar Teori

[1] Algoritma greedy adalah algoritma yang memecahkan persoalan secara langkah per langkah (step by step) sedemikian sehingga pada setiap langkah itu mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan dan "berharap" bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global. [2]Berikut elemen-elemen yang terdapat dalam Algoritma Greedy yang harus ditentukan dan dipertimbangkan.

1. Himpunan kandidat, C : berisi kandidat yang akan dipilih pada setiap Langkah (misal: simpul/sisi di dalam graf, job, task, koin, benda, karakter, dsb)
2. Himpunan solusi, S : berisi kandidat yang sudah dipilih
3. Fungsi solusi: menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi
4. Fungsi seleksi (selection function): memilih kandidat berdasarkan strategi greedy tertentu. Strategi greedy ini bersifat heuristik.
5. Fungsi kelayakan (feasible): memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak)
6. Fungsi obyektif : memaksimumkan atau meminimumkan

Namun, optimum global belum tentu merupakan solusi optimum (terbaik), bisa jadi merupakan solusi sub-optimum atau pseudo-optimum. Hal ini dikarenakan Algoritma greedy tidak beroperasi secara menyeluruh terhadap semua kemungkinan solusi yang ada dan terdapat beberapa fungsi seleksi yang berbeda sehingga harus memilih fungsi yang tepat jika ingin algoritma menghasilkan solusi optimal. [3]

2.2 Cara Kerja Program

Program bot ini bekerja dengan membaca kondisi papan permainan (*board*) dan memilih aksi terbaik secara lokal menggunakan strategi algoritma Greedy. Bot mengevaluasi semua objek diamond yang terlihat dan menghitung *value per distance* untuk menentukan diamond mana yang paling menguntungkan untuk diambil. Jika inventory penuh atau waktu tersisa tidak cukup untuk mengambil diamond baru, bot akan segera kembali ke base untuk menyimpan diamond.

Secara umum, program bekerja dengan menghubungkan bot ke papan permainan (*board*) menggunakan API yang telah disediakan oleh *game engine*. Setelah terhubung, bot akan secara otomatis melakukan aksi berdasarkan informasi kondisi papan saat ini. Logika pengambilan keputusan ini dibangun menggunakan pendekatan **algoritma Greedy**, di mana bot selalu memilih aksi terbaik lokal berdasarkan kriteria tertentu, seperti jarak terpendek ke diamond bernilai tinggi atau peluang skor tertinggi dalam waktu sesingkat mungkin. Setiap aksi bot dikirim ke server permainan melalui request HTTP, dan server akan merespons dengan kondisi terbaru papan. Proses ini dilakukan secara terus-menerus hingga waktu permainan habis. Berikut adalah rincian langkah kerja program:

1. **Cara Implementasi Program**

   Cara kerja program ini dimulai dengan membangun kelas turunan dari BaseLogic bernama GreedyValueDistanceBot. Bot membaca semua objek pada board, menyaring diamond, dan memilih target berdasarkan nilai terbaik yang dihitung dengan rumus: **nilai/(jarak+1)**

   Pemilihan arah gerak dilakukan dengan fungsi get_direction_towards, yang mengembalikan vektor gerak horizontal atau vertikal menuju target.

2. **Menjalankan Bot Program**

   Bot dijalankan dengan perintah python main.py atau menggunakan runner yang tersedia di starter pack. Program akan:

   - Mendaftar/memulihkan bot menggunakan API.
   - Masuk ke board permainan.
   - Mengambil aksi terus-menerus hingga waktu habis.

3. **Penanganan Aksi dan Respons**

   Setiap aksi bot dikirim ke server dalam bentuk HTTP POST ke endpoint move, yang mencakup arah gerakan ("NORTH", "SOUTH", "EAST", atau "WEST"). Server akan memberikan respons berupa status dan kondisi terbaru board. Program bot kemudian:

   - Memperbarui posisi dan informasi terkini dari GameObject.
   - Mengevaluasi kembali kondisi, seperti sisa waktu, posisi diamond, dan kapasitas inventory.
   - Mengambil keputusan aksi berikutnya berdasarkan strategi Greedy. Loop ini terus berlangsung selama waktu permainan aktif.

4. **Integrasi dengan Frontend**

   Frontend permainan disediakan oleh game engine dan berjalan secara otomatis. Visualisasi papan diperbarui secara berkala melalui request GET ke endpoint /api/boards/{id}. Hal ini memungkinkan:

   - Pemantauan jalannya permainan secara visual.
   - Debugging strategi bot dan pergerakan real-time.
   - Evaluasi keputusan yang diambil bot pada setiap langkah.

   Frontend akan menampilkan skor, posisi bot, posisi diamond, dan fitur-fitur lain seperti red button atau teleporter secara interaktif, sehingga mempermudah proses pengujian maupun kompetisi.

5. **Logika Pengambilan Keputusan**

   - Jika inventory penuh atau waktu tidak cukup: kembali ke base.
   - Jika ada diamond: pilih diamond dengan skor *value per distance* tertinggi.
   - Jika tidak ada target valid: tetap di tempat atau kembali ke base.

BAB III
APLIKASI STRATEGI GREEDY

Dalam pengembangan bot pada permainan *Diamonds*, penerapan algoritma greedy dilakukan dengan terlebih dahulu memetakan komponen-komponen permainan ke dalam elemen dasar dari algoritma greedy. Hal ini bertujuan untuk menyusun strategi pengambilan keputusan yang efisien dan adaptif terhadap kondisi permainan. Bot harus mampu memilih aksi terbaik secara lokal, berdasarkan nilai manfaat maksimum pada setiap langkahnya. Untuk itu, elemen seperti diamond, inventory, waktu, dan base dikaji dalam kerangka greedy sebagai kandidat solusi, fungsi seleksi, dan fungsi objektif.

3.1 Proses *Mapping*

- **Himpunan Kandidat**: Semua diamond di papan permainan.
- **Fungsi Seleksi**: Memilih diamond dengan rasio nilai per jarak tertinggi.
- **Fungsi Kelayakan**: Memastikan kapasitas inventory masih cukup dan waktu masih memungkinkan.
- **Fungsi Objektif**: Memaksimalkan total poin sebelum waktu habis.
- **Himpunan Solusi**: Urutan aksi bot dari mengambil diamond hingga menyimpan ke base.

Dalam proses perancangannya, beberapa alternatif strategi greedy dipertimbangkan. Setiap pendekatan memiliki fokus berbeda terhadap prioritas pergerakan bot, yang akan berdampak pada skor akhir. Eksplorasi dilakukan untuk memahami kelebihan dan kekurangan dari tiap strategi sebelum menentukan pilihan utama.

3.2 Eksplorasi Alternatif Solusi Greedy

- **Greedy Jarak Terpendek**: Selalu mengambil diamond terdekat tanpa mempertimbangkan nilai.
- **Greedy Nilai Tertinggi**: Mengejar diamond merah meskipun jaraknya jauh.
- **Greedy Value per Distance (yang diimplementasikan)**: Mengutamakan rasio nilai terhadap jarak.
- **Greedy Zona Aman**: Menghindari musuh dan hanya mengambil diamond di area aman.

Setelah beberapa strategi diuji dan dianalisis secara teoritis, dilakukan evaluasi terhadap efisiensi waktu dan efektivitas dalam menghasilkan skor. Strategi terbaik adalah yang mampu menyeimbangkan risiko, waktu, dan hasil yang diperoleh.

3.3 Analisis Efisiensi dan Efektivitas Solusi Greedy

- **Jarak Terpendek**: Efisien waktu, tetapi tidak selalu optimal skor.
- **Nilai Tertinggi**: Menghasilkan skor tinggi jika berhasil, tetapi boros waktu.
- **Value per Distance**: Seimbang dan fleksibel, cocok untuk kondisi dinamis.
- **Zona Aman**: Aman dari tackle, tetapi kurang agresif dan potensi skor rendah.

Berdasarkan eksplorasi dan analisis yang telah dilakukan, strategi yang dipilih untuk diimplementasikan adalah *Greedy Value per Distance*. Strategi ini menawarkan keseimbangan antara nilai diamond dan jarak tempuh, serta mampu beradaptasi dengan dinamika permainan seperti regenerasi diamond, kapasitas inventory, dan waktu tersisa.

3.4 Strategi Greedy yang Dipilih

Strategi yang dipilih adalah **Greedy Value per Distance**. Strategi ini menggabungkan efisiensi waktu dan nilai diamond, serta menjaga agar bot tetap produktif selama permainan berlangsung. Pemilihan target mempertimbangkan *trade-off* antara nilai diamond dan jarak, serta memperhatikan kapasitas inventory dan waktu tersisa. Strategi ini cukup adaptif untuk kondisi papan yang terus berubah.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma Greedy

1. Pseudocode

```
from typing import Optional
from game.logic.base import BaseLogicMore actions
from game.models import Board, GameObject, Position
from game.util import get_direction


# LIST OF IMPROVEMENTS
# TODO: Implement AVOID TELEPORT ON THE WAY
# TODO: Implement back to base if already have 2 diamonds and time is near to end <TESTING>
# TODO: Implement back to base if base is near and have 3 diamonds <TESTING>
# TODO: Implement AVOID ENEMY ON THE WAY TO DIAMOND
# TODO: Implement back to base if enemy is near (distance 2) and have 3 diamonds

# TODO: Implement Simple Diamond Greedy Algorithm
class GreedyDiamondLogic(BaseLogic):
    static_goals : list[Position] = []
    static_goal_teleport : GameObject = None
    static_temp_goals : Position = None
    static_direct_to_base_via_teleporter : bool = False

    def __init__(self) -> None:
        self.directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        self.goal_position: Optional[Position] = None
        self.current_direction = 0
        self.distance = 0

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

```
        self.board_bot = board_bot
        self.diamonds = board.diamonds
        self.bots = board.bots
        self.teleporter = [d for d in self.board.game_objects if d.type == "TeleportGameObject"]
        self.redButton = [d for d in self.board.game_objects if d.type == "DiamondButtonGameObject"]
        self.enemy = [d for d in self.bots if d.id != self.board_bot.id]
        self.enemyDiamond = [d.properties.diamonds for d in self.enemy]

        # REMOVE ALL STATIC WHEN IN BASE
        if (self.board_bot.position == self.board_bot.properties.base):
            self.static_goals = []
            self.static_goal_teleport = None
            self.static_temp_goals = None
            self.static_direct_to_base_via_teleporter = False

        # REMOVE STATIC GOALS IN TELEPORT
        if (self.static_goal_teleport and self.board_bot.position ==
self.find_other_teleport(self.static_goal_teleport)):
            self.static_goals.remove(self.static_goal_teleport.position)
            self.static_goal_teleport = None
        if (not self.static_goal_teleport and self.board_bot.position in self.static_goals):
            self.static_goals.remove(self.board_bot.position)

        # Remove temp goal if already reached
        if (self.board_bot.position == self.static_temp_goals):
            self.static_temp_goals = None

        # Analyze new state
        if props.diamonds == 5 or (props.milliseconds_left < 5000 and props.diamonds > 1):
            # Move to base
            self.goal_position = self.find_best_way_to_base()
            if not self.static_direct_to_base_via_teleporter:
                self.static_goals = []
                self.static_goal_teleport = None
        else:
            if (len(self.static_goals) == 0):
```

```
            self.find_nearest_diamond()
        self.goal_position = self.static_goals[0]


    if (self.calculate_near_base() and props.diamonds > 2):
        self.goal_position = self.find_best_way_to_base()
        if not self.static_direct_to_base_via_teleporter:
            self.static_goals = []
            self.static_goal_teleport = None

    if self.static_temp_goals: # If there is a temp goal, use it
        self.goal_position = self.static_temp_goals

    # Calculate next move
    current_position = board_bot.position
    if self.goal_position:
        # Check if there is a teleporter on the path
        if (not self.static_temp_goals):
            self.obstacle_on_path(
                'teleporter',
                current_position.x,
                current_position.y,
                self.goal_position.x,
                self.goal_position.y,
            )


        # Check if there is a red diamond on the path
        if (props.diamonds == 4):
            self.obstacle_on_path(
                'redDiamond',
                current_position.x,
                current_position.y,
                self.goal_position.x,
                self.goal_position.y,
            )
```

```
        # We are aiming for a specific position, calculate delta
        delta_x, delta_y = get_direction(
            current_position.x,
            current_position.y,
            self.goal_position.x,
            self.goal_position.y,
        )
    else:
        # Roam around
        delta = self.directions[self.current_direction]
        delta_x = delta[0]
        delta_y = delta[1]
        self.current_direction = (self.current_direction + 1) % len(
            self.directions
        )

    if (delta_x == 0 and delta_y == 0):
        # Reset goal
        self.static_goals = []
        self.static_direct_to_base_via_teleporter = False
        self.static_goal_teleport = None
        self.static_temp_goals = None
        self.goal_position = None
        tempMove = self.next_move(board_bot, board)
        delta_x, delta_y = tempMove[0], tempMove[1]

    return delta_x, delta_y

#Calculate the best way to base
def find_best_way_to_base(self):
    current_position = self.board_bot.position
    base = self.board_bot.properties.base
    base_position = Position(base.y, base.x)

    # Calculate distance to base with direct and teleporter
```

```
        base_distance_direct = abs(base.x - current_position.x) + abs(base.y - current_position.y)
        nearest_teleport_position, far_teleport_position, nearest_tp = self.find_nearest_teleport()

        if (nearest_teleport_position == None and far_teleport_position == None):
            return base_position

        # Find the best way to base
        base_distance_teleporter = abs(base.x - far_teleport_position.x) + abs(base.y -
far_teleport_position.y) + abs(nearest_teleport_position.x - current_position.x) +
abs(nearest_teleport_position.y - current_position.y)
        if (base_distance_direct < base_distance_teleporter):
            return base_position
        else:
            self.static_direct_to_base_via_teleporter = True
            self.static_goal_teleport = nearest_tp
            self.static_goals = [nearest_teleport_position, base]
            return nearest_teleport_position

    def calculate_near_base(self):
        current_position = self.board_bot.position
        base = self.board_bot.properties.base

        # Calculate distance to base with direct and teleporter
        base_distance = abs(base.x - current_position.x) + abs(base.y - current_position.y)
        base_distance_teleporter = self.find_base_distance_teleporter()
        distance = base_distance_teleporter if base_distance_teleporter < base_distance else
base_distance

        if (distance == 0):
            return False

        return distance < self.distance

    def find_base_distance_teleporter(self):
        current_position = self.board_bot.position
```

```
        # Calculate distance to base with teleporter
        nearest_teleport_position, far_teleport_position, nearest_teleport = self.find_nearest_teleport()

        if (nearest_teleport_position == None and far_teleport_position == None and nearest_teleport
== None):
            return float("inf")

        base = self.board_bot.properties.base
        base_distance_teleporter = abs(base.x - far_teleport_position.x) + abs(base.y -
far_teleport_position.y) + abs(nearest_teleport_position.x - current_position.x) +
abs(nearest_teleport_position.y - current_position.y)
        return base_distance_teleporter

    def find_nearest_diamond(self) -> Optional[Position]:
        direct = self.find_nearest_diamond_direct() # distance, position
        teleport = self.find_nearest_diamond_teleport() # distance, [teleportPosition, diamondPosition]
        redButton = self.find_nearest_red_button() # distance, position
        if (direct[0] < teleport[0] and direct[0] < redButton[0]):
            self.static_goals = [direct[1]]
            self.distance = direct[0]
        elif (teleport[0] < direct[0] and teleport[0] < redButton[0]):
            self.static_goals = teleport[1]
            self.static_goal_teleport = teleport[2]
            self.distance = teleport[0]
        else:
            self.static_goals = [redButton[1]]
            self.distance = redButton[0]

    # Find the nearest red button
    def find_nearest_red_button(self):
        current_position = self.board_bot.position
        distance = abs(self.redButton[0].position.x - current_position.x) +
abs(self.redButton[0].position.y - current_position.y)
        return distance, self.redButton[0].position

    # Find the nearest teleport
```

```
    def find_nearest_teleport(self):
        nearest_teleport_position, far_teleport_position, nearest_tp = None, None, None
        min_distance = float("inf")
        for teleport in self.teleporter:
            distance = abs(teleport.position.x - self.board_bot.position.x) + abs(teleport.position.y -
self.board_bot.position.y)
            if distance == 0:
                return None, None, None
            if distance < min_distance:
                min_distance = distance
                nearest_teleport_position,far_teleport_position = teleport.position,
self.find_other_teleport(teleport)
                nearest_tp = teleport
        return nearest_teleport_position,far_teleport_position, nearest_tp

    # Find the other teleport
    def find_other_teleport(self, teleport: GameObject):
        for t in self.teleporter:
            if t.id != teleport.id:
                return t.position

    # Find the nearest diamond with teleport
    def find_nearest_diamond_teleport(self) -> Optional[Position]:
        current_position = self.board_bot.position
        nearest_teleport_position, far_teleport_position, nearest_teleport = self.find_nearest_teleport()

        if (nearest_teleport_position == None and far_teleport_position == None and nearest_teleport
== None):
            return float("inf")

        min_distance = float("inf")
        nearest_diamond = None

        # Calculate distance to diamond with teleport
        for diamond in self.diamonds:
```

```
            distance = abs(diamond.position.x - far_teleport_position.x) + abs(diamond.position.y - far_teleport_position.y) + abs(nearest_teleport_position.x - current_position.x) + abs(nearest_teleport_position.y - current_position.y)
            distance /= diamond.properties.points
            if distance < min_distance and ((diamond.properties.points == 2 and self.board_bot.properties.diamonds != 4) or (diamond.properties.points == 1)):
                min_distance = distance
                nearest_diamond = [nearest_teleport_position, diamond.position]
        return min_distance, nearest_diamond, nearest_teleport

    # Find the nearest diamond with direct
    def find_nearest_diamond_direct(self) -> Optional[Position]:
        current_position = self.board_bot.position
        min_distance = float("inf")
        nearest_diamond = None
        for diamond in self.diamonds:
            distance = abs(diamond.position.x - current_position.x) + abs(diamond.position.y - current_position.y)
            distance /= diamond.properties.points
            if distance < min_distance and ((diamond.properties.points == 2 and self.board_bot.properties.diamonds != 4) or (diamond.properties.points == 1)):
                min_distance = distance
                nearest_diamond = diamond.position
        return min_distance, nearest_diamond

    def obstacle_on_path(self, type, current_x, current_y, dest_x, dest_y):
        if type == 'teleporter':
            object = self.teleporter
        elif type == 'redDiamond':
            object = [d for d in self.diamonds if d.properties.points == 2]
        elif type == 'redButton':
            object = self.redButton

        for t in object:
            if current_x == t.position.x and  current_y == t.position.y:
                continue
```

```
        # Kondisi saat redDiamond sejajar dengan destinasi dalam sumbu y dan berada pada jalur current->dest
        if t.position.x == dest_x and (dest_y < t.position.y <= current_y or current_y <= t.position.y < dest_y):

            # Kondisi saat current tidak sejajar dengan destinasi pada sumbu y
            if (dest_x != current_x):
                self.goal_position = Position(dest_y,dest_x-1) if dest_x > current_x else Position(dest_y,dest_x+1)

            # Kondisi saat current sejajar dengan destinasi pada sumbu y
            else:
                # Handle kalo dipinggir kiri/kanan
                if (dest_x <= 1):
                    self.goal_position = Position(dest_y,dest_x+1)
                else:
                    self.goal_position = Position(dest_y,dest_x-1)
            self.static_temp_goals = self.goal_position

        # Kondisi saat redDiamond sejajar dengan destinasi dalam sumbu x dan berada pada jalur current->dest (Tidak akan pernah terjadi)
        elif t.position.y == dest_y and (dest_x < t.position.x <= current_x or current_x <= t.position.x < dest_x):

            # Kondisi saat current tidak sejajar dengan destinasi pada sumbu x
            if (dest_y != current_y):
                self.goal_position = Position(dest_y-1,dest_x) if dest_y > current_y else Position(dest_y+1,dest_x)

            # Kondisi saat current sejajar dengan destinasi pada sumbu x
            else:
                # Handle kalo dipinggir atas/bawah
                if (dest_y <= 1):
                    self.goal_position = Position(dest_y+1,dest_x)
                else:
                    self.goal_position = Position(dest_y-1,dest_x)
```

```
            self.static_temp_goals = self.goal_position

        # Kondisi saat redDiamond sejajar dengan current dalam sumbu x dan berada pada jalur current->dest
        elif t.position.y == current_y and (dest_x < t.position.x <= current_x or current_x <= t.position.x < dest_x):

            # Kondisi saat current tidak sejajar dengan destinasi pada sumbu x
            if (dest_y != current_y):
                self.goal_position = Position(dest_y,current_x)

            # Kondisi saat current sejajar dengan destinasi pada sumbu y
            else:
                # Handle kalo dipinggir kiri/kanan
                if (current_y <= 1):
                    self.goal_position = Position(current_y+1,current_x)
                else:
                    self.goal_position = Position(current_y-1,current_x)

            self.static_temp_goals = self.goal_position
```

2. **Penjelasan Alur Program**

Setelah strategi ditentukan, langkah selanjutnya adalah mengimplementasikan logika greedy ke dalam kode program bot. Bot dikembangkan dalam bentuk kelas Python GreedyValueDistanceBot, yang melakukan analisis kondisi papan permainan pada setiap langkahnya. Informasi seperti lokasi diamond, kapasitas inventory, waktu tersisa, dan posisi base digunakan untuk menentukan langkah berikutnya. Jika kondisi inventory sudah penuh atau waktu tidak cukup untuk mengambil diamond baru, maka bot akan segera diarahkan ke base.

Penjelasan Alur Program

- Bot membaca kondisi papan dari objek board.
- Mengambil semua objek diamond dan menyimpan dalam list.

- Mengecek kondisi inventory dan waktu:
    - Jika penuh atau waktu hampir habis, bot kembali ke base.
- Jika masih bisa mengambil diamond:
    - Bot menghitung rasio nilai per jarak untuk semua diamond.
    - Menentukan target dengan skor tertinggi.
- Bot mengarahkan diri menuju target menggunakan fungsi get_direction_towards.

Untuk mendukung alur logika tersebut, digunakan beberapa struktur data yang memungkinkan pengolahan data posisi dan properti game secara efisien. Struktur ini disusun agar mudah diakses dan dimanipulasi selama permainan berlangsung.

4.2 Struktur Data yang Digunakan

- List[GameObject]: Menyimpan semua diamond dan objek lainnya.
- Position: Menyimpan koordinat x dan y bot, base, dan diamond.
- GameObject.properties: Menyimpan informasi properti seperti poin, inventory, base, waktu tersisa.
- Tuple (int, int): Digunakan untuk arah gerakan (dx, dy).

Pengujian dilakukan dengan berbagai skenario permainan, mulai dari kondisi ideal (diamond dekat dan banyak), kondisi penuh inventory, waktu hampir habis, hingga kondisi tidak ada diamond di sekitar. Tujuan dari pengujian ini adalah memastikan bot mampu berperilaku optimal dan sesuai dengan strategi yang telah dirancang.

4.3 Pengujian Program

1. **Skenario Pengujian**
    - Papan dengan banyak diamond merah terdekat: Apakah bot mengejar dengan efisien?
    - Inventory hampir penuh: Apakah bot segera kembali ke base?
    - Waktu hampir habis: Apakah bot berhenti atau kembali ke base tepat waktu?
    - Tidak ada diamond di sekitar: Apakah bot tetap diam atau kembali ke base?
2. **Hasil Pengujian dan Analisis**
    - Pada kondisi diamond tersebar merata, bot mampu memilih target dengan nilai tertinggi secara efisien.
    - Bot berhasil kembali ke base tepat waktu sebelum inventory penuh atau waktu habis.

- Jika tidak ada diamond dalam jangkauan, bot mengambil keputusan aman dengan kembali ke base.
- Dalam beberapa kasus, bot tidak selalu optimal jika posisi musuh sangat agresif (tackle), karena strategi belum mempertimbangkan posisi lawan secara langsung.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Tugas besar ini menunjukkan bagaimana penerapan algoritma greedy dapat digunakan untuk membangun strategi pengambilan keputusan dalam permainan kompetitif seperti *Diamonds*. Melalui proses pemetaan elemen permainan ke dalam kerangka greedy, eksplorasi alternatif strategi, serta implementasi dalam bentuk bot, diperoleh solusi yang efektif dan efisien untuk mencapai tujuan permainan, yaitu mengumpulkan poin sebanyak-banyaknya.

Strategi **Greedy Value per Distance** yang dipilih mampu memberikan hasil yang baik dalam berbagai kondisi permainan. Bot dapat menentukan target diamond yang menguntungkan, mempertimbangkan jarak dan nilai, serta mengatur pengelolaan inventory dan waktu secara mandiri. Pengujian yang dilakukan menunjukkan bahwa bot mampu bekerja optimal saat tidak ada gangguan dari lawan, dan dapat menyelesaikan permainan dengan performa yang stabil.

Namun, meskipun strategi yang digunakan cukup adaptif terhadap kondisi papan, masih terdapat keterbatasan dalam hal respons terhadap kehadiran dan pergerakan bot musuh. Hal ini menjadi titik penting yang dapat ditingkatkan untuk mencapai performa yang lebih kompetitif.

5.2 Saran

Pengembangan lebih lanjut terhadap bot ini dapat difokuskan pada beberapa aspek, di antaranya:

- **Integrasi strategi defensif** dengan memperhitungkan posisi bot musuh untuk menghindari tackle, seperti dengan membuat zona berbahaya yang dihindari oleh bot.
- **Peningkatan adaptivitas** dengan mempertimbangkan kondisi real-time, seperti regenerasi diamond dan dinamika pergerakan lawan, agar strategi menjadi lebih kontekstual.
- **Penggabungan strategi hibrida**, misalnya kombinasi greedy dengan pendekatan probabilistik atau prediktif, untuk memberikan fleksibilitas lebih tinggi dalam pengambilan keputusan.
- **Pengoptimalan pengelolaan waktu**, seperti membuat fungsi evaluasi kapan saat terbaik untuk berhenti mencari diamond dan segera kembali ke base sebelum waktu habis.

Dengan peningkatan tersebut, diharapkan bot dapat menjadi lebih tangguh dalam menghadapi lawan dan lebih optimal dalam berbagai kondisi permainan kompetitif yang dinamis.

LAMPIRAN

A. Repository Github

Link: https://github.com/handi18/Tubes_Stigma_TheAndri_direct.git

B. Video Penjelasan (link GDrive)

Link: https://drive.google.com/drive/folders/19ShVzapfyH7pnNsHvp-XjwDRGa_XBXnu?usp=drive_link

DAFTAR PUSTAKA

[1] R. Munir, “Algoritma Greedy (Bagian 1)” (online)., https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf.

[2] R. Munir, Algoritma Greedy (Bagian 2)” (online), https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf.

[3] R. Munir, Munir, Rinaldi. “Algoritma Greedy (Bagian 3)” (online)., https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf.

[4] https://www.geeksforgeeks.org/greedy-algorithms/, "Greedy Algorithms," 07 Apr 2025.