IMPLEMENTASI ALGORITMA GREEDY DALAM PEMECAHAN BOT PERMAINAN DIAMOND

Tugas Besar

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

Oleh: Kelompok 8 (3J)

Cikal Galih Nur Arifin	123140109
Ragil Bayu Saputra	123140128
Rafael Abimanyu Ratmoko	123140134

Dosen Pengampu: Winda Yulita, M.Cs.

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.

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

Untuk mengimplementasikan algoritma pada bot tersebut, mahasiswa dapat menggunakan *game engine* dan membuat bot dari bot starter pack yang telah tersedia pada pranala berikut.

- *Game engine*: https://github.com/haziqam/tubes1-IF2211-game-engine/releases/tag/v1.1.0
- *Bot starter pack*: https://github.com/haziqam/tubes1-IF2211-bot-starter-pack/releases/tag/v1.0.1

Komponen-komponen dari permainan Diamonds antara lain:

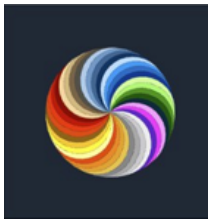
1. *Diamonds*

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*

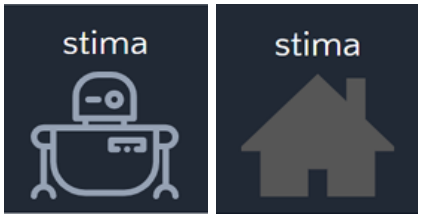
Ketika *red button* ini dilewati/dilangkahi, semua *diamond* (termasuk *red diamond*) akan di-*generate* kembali pada board dengan posisi acak. Posisi *red button* ini juga akan berubah secara acak jika *red button* ini dilangkahi.

3. *Teleporters*



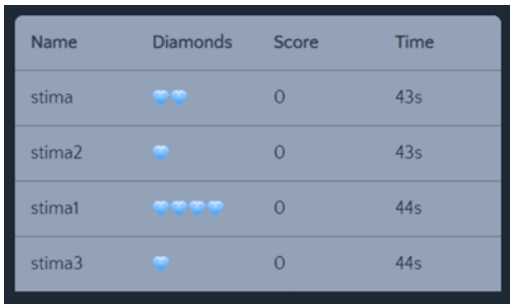
Terdapat 2 *teleporter* yang saling terhubung satu sama lain. Jika *bot* melewati sebuah *teleporter* maka *bot* akan berpindah menuju posisi *teleporter* yang lain.

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*



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.

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

1. Pertama, setiap pemain (bot) akan ditempatkan pada board secara random. Masing-masing bot akan mempunyai home base, serta memiliki score dan inventory awal bernilai nol.
2. Setiap bot diberikan waktu untuk bergerak, waktu yang diberikan semua sama untuk setiap pemain.
3. Objektif utama bot adalah mengambil diamond-diamond yang ada di peta sebanyak-banyaknya. Seperti yang sudah disebutkan di atas, diamond yang berwarna merah memiliki 2 poin dan diamond yang berwarna biru memiliki 1 poin.
4. Setiap bot juga memiliki sebuah inventory, dimana inventory berfungsi sebagai tempat penyimpanan sementara diamond yang telah diambil. Inventory ini sewaktu-waktu bisa penuh, maka dari itu bot harus segera kembali ke home base.
5. Apabila bot menuju ke posisi home base, score bot akan bertambah senilai diamond yang tersimpan pada inventory dan inventory bot akan menjadi kosong kembali.

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

Panduan Penggunaan

Adapun panduan mengenai cara instalasi, menjalankan permainan, membuat bot, melihat visualizer/frontend, dan mengatur konfigurasi permainan dapat dilihat melalui tautan berikut.

https://docs.google.com/document/d/1L92Axb89yIkom0b24D350Z1QAr8rujvHof7-kXRAp7c/edit?tab=t.0

Mekanisme Teknis Permainan Diamonds

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

1. Program bot akan mengecek apakah bot sudah terdaftar atau belum, dengan mengirimkan POST request terhadap endpoint /api/bots/recover dengan body berisi email dan password bot. Jika bot sudah terdaftar, maka backend akan memberikan response code 200 dengan body berisi id dari bot tersebut. Jika tidak, backend akan memberikan response code 404.
2. Jika bot belum terdaftar, maka program bot akan mengirimkan POST request terhadap endpoint /api/bots dengan body berisi email, name, password, dan team. Jika berhasil, maka backend akan memberikan response code 200 dengan body berisi id dari bot tersebut.
3. Ketika id bot sudah diketahui, bot dapat bergabung ke board dengan mengirimkan POST request terhadap endpoint /api/bots/{id}/join dengan body berisi board id yang diinginkan (preferredBoardId). Apabila bot berhasil bergabung, maka backend akan memberikan response code 200 dengan body berisi informasi dari board.
4. Program bot akan mengkalkulasikan move selanjutnya secara berkala berdasarkan kondisi board yang diketahui, dan mengirimkan POST request terhadap endpoint /api/bots/{id}/move dengan body berisi direction yang akan ditempuh selanjutnya (“NORTH”, “SOUTH”, “EAST”, atau “WEST”). Apabila berhasil, maka backend akan memberikan response code 200 dengan body berisi kondisi board setelah move tersebut.

Langkah ini dilakukan terus-menerus hingga waktu bot habis. Jika waktu bot habis, bot secara otomatis akan dikeluarkan dari board.

5. Program frontend secara periodik juga akan mengirimkan GET request terhadap endpoint /api/boards/{id} untuk mendapatkan kondisi board terbaru, sehingga tampilan board pada frontend akan selalu ter-update.

BAB II
LANDASAN TEORI

2.1 Dasar Teori

Algoritma greedy (serakah) adalah pendekatan dalam pemrograman yang digunakan untuk menyelesaikan masalah optimasi dengan cara memilih solusi lokal terbaik pada setiap langkah, dengan harapan bahwa keputusan lokal tersebut akan menghasilkan solusi global yang optimal. Strategi ini tidak mengevaluasi semua kemungkinan, tetapi lebih memilih keputusan yang paling menguntungkan saat itu juga (locally optimal choice), lalu melanjutkan ke langkah berikutnya hingga selesai. Pendekatan greedy akan memberikan solusi yang optimal hanya jika masalah yang dihadapi memiliki dua sifat utama, yaitu *greedy choice property* (pilihan lokal optimal mengarah ke solusi global optimal) dan *optimal substructure* (solusi dari suatu masalah dapat dibangun dari solusi submasalahnya):

a. Greedy Choice Property : Solusi optimal global dapat dibentuk dari keputusan optimal lokal.
b. Optimal Substructure : Solusi dari masalah dapat disusun dari solusi masalah-masalah sub bagian (submasalah) yang optimal.

Sebagaimana dijelaskan oleh Cormen dkk. dalam buku *Introduction to Algorithms*, *"sebuah algoritma greedy memperoleh solusi optimal untuk suatu masalah dengan membuat serangkaian pilihan. Pada setiap titik keputusan, algoritma membuat pilihan yang tampak terbaik saat itu"* [1, hlm. 423]. Dengan kata lain, algoritma ini mengambil keputusan terbaik yang tersedia di setiap langkah, dengan harapan bahwa serangkaian keputusan lokal tersebut akan menghasilkan solusi global yang benar-benar optimal.

2.2 Cara Kerja Program

Program kami terdiri dari dua bagian utama, yaitu Game Engine dan Bot. Komunikasi antara bot dan game engine dilakukan melalui HTTP API, di mana bot melakukan serangkaian request untuk mendaftar, bergabung ke permainan, serta mengirimkan perintah pergerakan secara berkala. Bot dibuat dengan menggunakan bahasa Python, dan strategi utamanya berbasis pada algoritma greedy.

2.2.1. Inisialisasi dan Registrasi Bot

Pada awal program dijalankan, bot akan mengecek apakah sudah terdaftar pada sistem backend. Jika belum, bot akan mengirimkan POST request ke endpoint /api/bots disertai informasi seperti email, nama, password, dan tim. Jika berhasil, backend akan mengembalikan ID bot. Setelah itu, bot mengirimkan request ke /api/bots/{id}/join untuk bergabung ke board

permainan. Saat berhasil, bot menerima data board dan informasi awal yang diperlukan untuk memulai pengambilan keputusan.

2.2.2. Pengambilan Keputusan Bot Menggunakan Strategi Greedy

Setelah bot berada di dalam permainan, setiap tick waktu, bot akan menerima informasi tentang kondisi terkini dari papan permainan (seperti posisi diamond, lawan, dan posisi red button atau teleporter). Berdasarkan informasi tersebut, bot akan menghitung gerakan selanjutnya yang dianggap paling menguntungkan secara lokal. Bot kemudian akan mengirimkan request ke endpoint /api/bots/{id}/move untuk bergerak ke arah yang telah dipilih (utara, selatan, timur, atau barat).

2.2.3. Manajemen Inventory dan Penyimpanan ke Base

Bot memiliki inventory terbatas. Jika inventory hampir penuh, maka prioritas greedy akan dialihkan untuk kembali ke home base dan menyimpan diamond. Skor akan bertambah dan inventory dikosongkan saat menyentuh base.

2.2.4. Menghindari Risiko Tackle

Selain mencari diamond, bot juga menghindari posisi lawan untuk mencegah tackle yang menyebabkan kehilangan semua isi inventory. Dalam kondisi berisiko, bot dapat memprioritaskan jalan yang lebih aman meskipun jaraknya sedikit lebih jauh.

2.2.5. Fitur Khusus: Red Button dan Teleporter

Bot juga dapat memanfaatkan red button untuk me-reset posisi diamond secara acak bila diperlukan (misalnya saat tidak ada diamond dekat). Selain itu, teleporter dapat digunakan sebagai jalur cepat menuju area dengan diamond lebih banyak atau untuk kabur dari lawan.

2.2.6. Siklus Berulang Hingga Permainan Berakhir

Proses pengambilan keputusan dan pergerakan akan terus diulang selama permainan berlangsung. Bot akan terus mengamati kondisi papan, mengkalkulasi opsi terbaik secara lokal, dan bergerak sesuai keputusan greedy hingga waktu habis. Setelah permainan selesai, skor akhir bot akan ditampilkan.

BAB III
APLIKASI STRATEGI *GREEDY*

3.1 Proses *Mapping*

Strategi greedy dalam permainan *Diamonds* dirancang untuk mengambil keputusan secara bertahap dengan fokus pada keuntungan lokal terbesar di setiap langkah, sambil mempertimbangkan batasan dan kondisi permainan yang berubah-ubah. Untuk dapat diterapkan secara sistematis, persoalan permainan ini dimodelkan ke dalam komponen-komponen utama dari algoritma greedy, yang terdiri dari:

- **Himpunan Kandidat:** Kumpulan objek atau aksi yang tersedia untuk dipilih oleh bot pada satu waktu tertentu. Dalam permainan ini, kandidat dapat berupa posisi diamond, base, red button, teleporter, hingga bot lawan yang sedang membawa diamond.
- **Himpunan Solusi:** Merupakan himpunan dari kandidat-kandidat yang telah dipilih. Dalam konteks permainan, ini dapat diartikan sebagai jalur atau urutan tindakan yang telah diambil oleh bot selama permainan berlangsung.
- **Fungsi Solusi:** Fungsi ini memeriksa apakah langkah-langkah (kandidat) yang telah diambil telah membentuk sebuah solusi lengkap, seperti ketika bot berhasil mengumpulkan dan menyimpan diamond di base.
- **Fungsi Seleksi (Selection Function):** Bertugas menentukan kandidat terbaik dari himpunan kandidat berdasarkan kriteria tertentu. Dalam strategi greedy, fungsi seleksi bersifat heuristik, misalnya memilih diamond dengan nilai tertinggi per langkah, atau memilih musuh terdekat yang membawa diamond untuk ditackle.
- **Fungsi Kelayakan (Feasibility Function):** Mengecek apakah kandidat yang terpilih memenuhi syarat untuk menjadi bagian dari solusi, seperti memastikan inventory belum penuh sebelum mengambil diamond, atau memastikan waktu tersisa cukup untuk kembali ke base.
- **Fungsi Objektif**: Tujuan akhir dari strategi ini adalah untuk mengoptimalkan skor, yaitu memaksimalkan jumlah poin yang diperoleh melalui pengambilan diamond dan penyimpanan ke base, sekaligus meminimalkan kerugian seperti tertackle atau kehabisan waktu.

Dengan menerapkan pemetaan ini, strategi greedy yang dibangun dapat dijalankan secara terstruktur, logis, dan sesuai dengan dinamika permainan *Diamonds*. Pendekatan ini memungkinkan bot untuk mengambil keputusan cepat dan efisien berdasarkan informasi yang tersedia saat itu, tanpa perlu mempertimbangkan seluruh kemungkinan jalur secara menyeluruh.

3.2 Eksplorasi Alternatif Solusi Greedy

Dalam mengembangkan strategi greedy untuk permainan Diamonds, terdapat berbagai pendekatan yang dapat dieksplorasi untuk menentukan aksi bot yang paling menguntungkan secara lokal. Strategi-strategi ini memiliki fokus dan kelebihan masing-masing tergantung pada kondisi permainan, seperti distribusi diamond, posisi musuh, serta sisa waktu. Berikut adalah beberapa alternatif solusi greedy yang dipertimbangkan:

3.2.1. Greedy by Distance

Pada pendekatan ini, bot selalu memilih untuk bergerak ke diamond yang memiliki jarak paling dekat dari posisinya saat ini. Strategi ini bertujuan untuk memaksimalkan jumlah pengambilan diamond dalam waktu terbatas dengan meminimalkan langkah per target. Kelebihannya adalah efisien dari segi waktu tempuh, tetapi kelemahannya adalah bot bisa saja lebih sering mengambil diamond bernilai kecil (biru), dan mengabaikan diamond merah yang sedikit lebih jauh namun bernilai lebih tinggi.

3.2.2. Greedy by Point

Strategi ini membuat bot memprioritaskan diamond dengan nilai tertinggi, yaitu diamond merah. Selama masih tersedia di papan, bot akan mengabaikan diamond biru dan hanya bergerak menuju diamond merah, meskipun letaknya relatif lebih jauh. Pendekatan ini cocok jika waktu masih cukup panjang, namun dapat menjadi tidak efisien saat diamond merah tersebar sangat jauh, sehingga waktu tempuhnya tidak sebanding dengan poin yang diperoleh.

3.2.3. Greedy by Point per Distance

Gabungan dari dua pendekatan sebelumnya, strategi ini memilih diamond dengan rasio nilai dibagi jarak tertinggi. Dengan kata lain, bot akan menilai semua diamond berdasarkan seberapa besar poin yang bisa didapatkan dibandingkan dengan jumlah langkah yang diperlukan untuk mencapainya. Strategi ini memberikan

keseimbangan antara kecepatan dan keuntungan, dan cenderung lebih fleksibel terhadap kondisi permainan yang dinamis.

3.2.4. Greedy by Tackling

Dalam strategi ini, bot akan mengincar bot lawan yang sedang membawa diamond, khususnya jika lawan berada dalam jangkauan dekat dan membawa banyak poin. Jika berhasil melakukan tackle, bot dapat mencuri seluruh isi inventory lawan, yang bisa menjadi cara cepat untuk mendapatkan poin besar. Meskipun berisiko, strategi ini efektif untuk mengganggu lawan sekaligus meningkatkan skor sendiri dalam waktu singkat.

3.2.5. Greedy by Base Awareness

Strategi ini mempertimbangkan jarak antara bot dan base dalam pengambilan keputusan. Bot cenderung tidak bergerak terlalu jauh dari base, agar bisa menyimpan diamond dengan lebih aman dan cepat. Pendekatan ini mengurangi risiko kehilangan diamond saat waktu hampir habis atau ketika inventory penuh, namun bisa membuat bot kehilangan peluang mengambil diamond bernilai tinggi yang letaknya agak jauh.

3.3 Analisis Efisiensi dan Efektivitas Solusi Greedy

No	Alternatif Solusi (Greedy by)	Kelebihan	Kekurangan
1.	Distance (Jarak Terdekat)	● Memungkinkan bot mengambil banyak diamond dalam waktu singkat. ● Efisien dalam waktu tempuh karena selalu menuju diamond terdekat.	● Cenderung mengabaikan diamond merah yang hanya sedikit lebih jauh. ● Skor per diamond relatif rendah karena dominan mengambil diamond biru.
2.	Point (Nilai Diamond)	● Memprioritaskan diamond merah yang bernilai lebih tinggi.	● Tidak efisien jika diamond merah tersebar jauh.

		● Berpotensi meningkatkan skor lebih cepat jika berhasil dikumpulkan.	● Waktu habis sebelum mendapatkan banyak poin jika terlalu fokus pada diamond merah yang jauh.
3.	Point per Distance (Efisiensi Nilai/Langkah)	● Menyeimbangkan antara nilai diamond dan jarak tempuh. ● Fleksibel dan adaptif terhadap kondisi permainan.	● Bot bisa menjauh dari base demi diamond bernilai tinggi. ● Butuh perhitungan rasio yang konsisten agar tidak salah prioritas.
4.	Tackling (Menargetkan Lawan)	● Dapat memperoleh banyak poin secara instan dari inventory lawan. ● Mengganggu strategi dan perolehan skor lawan.	● Berisiko tinggi jika tackle gagal. ● Bot bisa masuk dalam situasi kejar-kejaran yang menghabiskan waktu.
5.	Base Awareness (Kesadaran Posisi Base)	● Aman dari kehilangan diamond karena dekat dengan base. ● Efektif saat inventory hampir penuh atau waktu hampir habis.	● Mengorbankan potensi diamond bernilai tinggi yang letaknya jauh. ● Kurang agresif, sehingga bisa kalah dari bot yang lebih ofensif.

3.4 Strategi Greedy yang Dipilih

Berdasarkan hasil analisis kami, strategi Greedy yang kami pilih adalah gabungan antara beberapa algoritma Greedy , yaitu sebagai berikut :

3.4.1. Greedy by Distance

Ini adalah strategi inti. Bot secara konsisten menghitung jarak efektif (mempertimbangkan jalur langsung dan teleporter) ke semua berlian yang tersedia dan memprioritaskan berlian dengan jarak terpendek

3.4.2. Greedy by Points

Jika terdapat beberapa berlian dengan jarak efektif yang sama atau sangat mirip, bot akan lebih memilih berlian merah (2 poin) daripada berlian biru (1 poin)

3.4.3. Greedy by Tackle

Greedy by Tackling kami pilih karena berdasarkan percobaan-percobaan yang kami lakukan, melakukan Tackle terhadap bot lawan sangatlah menguntungkan. Selain mendapatkan poin dengan jumlah yang besar dalam waktu yang singkat, Greedy by Tackling juga akan sangat meningkatkan peluang kita untuk menang karena lawan yang kita tackle akan kehilangan Diamond-nya, dan biasanya sangat sulit untuk kembali menjadi kompetitif pada permainan tersebut. Selain itu, dengan melakukan tackle juga kita mencegah bot kita untuk di-tackle oleh bot lain.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma Greedy

1. Pseudocode

```
import random
import math
from typing import Optional, List, Tuple, Dict, Any

from game.logic.base import BaseLogic
from game.models import GameObject, Board, Position

MIN_DIAMONDS_TO_TACKLE = 5
MAX_DISTANCE_TO_CONSIDER_TACKLE = 1
TACKLE_SCORE_PENALTY = 10000.0

SAFE_RETURN_BUFFER_SECONDS = 3
INVENTORY_SIZE_DEFAULT = 5
LOW_DIAMOND_THRESHOLD_FOR_RED_BUTTON = 4



class SuperBot(BaseLogic):
    def __init__(self):
        super().__init__()
        self.goal_position: Optional[Position] = None

    def _clamp(self, n, smallest, largest) -> int:
        return max(smallest, min(n, largest))

    def _position_equals(self, a: Position, b: Optional[Position]) -> bool:
        if b is None: return False
        return a.x == b.x and a.y == b.y

    def _get_manhattan_distance(self, pos1: Optional[Position], pos2: Optional[Position]) -> int:
        if pos1 is None or pos2 is None: return float('inf')
        return abs(pos1.x - pos2.x) + abs(pos1.y - pos2.y)
```

```
  def _get_teleporter_pair(self, teleport_obj: GameObject, all_teleporters: List[GameObject]) ->
Optional[GameObject]:
      if len(all_teleporters) < 2: return None
      pair_id = getattr(teleport_obj.properties, 'pair_id', None)
      if pair_id is None:
          if len(all_teleporters) == 2:
              return next((t for t in all_teleporters if t.id != teleport_obj.id), None)
          return None
      return next((t for t in all_teleporters if t.id == pair_id), None)

  def _get_direction_advanced(self, current_pos: Position, dest_pos: Optional[Position],
                              avoid_positions_tuples: List[Tuple[int,int]],
                              board_width: int, board_height: int) -> Tuple[int, int]:
      if dest_pos is None: return 0,0
      current_x, current_y = current_pos.x, current_pos.y
      dest_x, dest_y = dest_pos.x, dest_pos.y
      delta_x_ideal = self._clamp(dest_x - current_x, -1, 1)
      delta_y_ideal = self._clamp(dest_y - current_y, -1, 1)
      if delta_x_ideal == 0 and delta_y_ideal == 0: return 0, 0

      preferred_moves = []
      if delta_x_ideal != 0: preferred_moves.append((delta_x_ideal, 0))
      if delta_y_ideal != 0: preferred_moves.append((0, delta_y_ideal))

      fallback_moves = [(1, 0), (0, 1), (-1, 0), (0, -1)]
      random.shuffle(fallback_moves)
      for move in fallback_moves:
          if move not in preferred_moves: preferred_moves.append(move)

      for dx, dy in preferred_moves:
          if dx != 0 and dy != 0: continue
          next_x, next_y = current_x + dx, current_y + dy
          if 0 <= next_x < board_width and 0 <= next_y < board_height and \
             (next_x, next_y) not in avoid_positions_tuples:
              return dx, dy
      return 0, 0

  def _calculate_path(self, start_pos: Position, end_pos: Optional[Position], teleporters:
List[GameObject]) \
```

```
        -> Tuple[int, Optional[GameObject], Optional[GameObject]]:
    if end_pos is None:
        return float('inf'), None, None

    dist_direct = self._get_manhattan_distance(start_pos, end_pos)
    dist_teleport = float('inf')
    best_tp_entry, best_tp_exit = None, None

    if len(teleporters) >= 2:
        for tp_e in teleporters:
            tp_x = self._get_teleporter_pair(tp_e, teleporters)
            if tp_e and hasattr(tp_e, 'position') and tp_e.position and \
               tp_x and hasattr(tp_x, 'position') and tp_x.position:
                d = self._get_manhattan_distance(start_pos, tp_e.position) + \
                    self._get_manhattan_distance(tp_x.position, end_pos)
                if d < dist_teleport:
                    dist_teleport, best_tp_entry, best_tp_exit = d, tp_e, tp_x

    if dist_teleport < dist_direct:
        return dist_teleport, best_tp_entry, best_tp_exit
    return dist_direct, None, None

def _evaluate_diamonds_by_distance_priority(self, current_pos: Position, diamonds:
List[GameObject],
                                            teleporters: List[GameObject]) -> List[Dict[str, Any]]:
    diamond_targets = []
    for diamond in diamonds:
        diamond_points = getattr(diamond.properties, 'points', 0)
        if diamond_points == 0: continue

        dist_to_diamond, tp_entry_to_diamond, _ = self._calculate_path(current_pos,
diamond.position, teleporters)
        if dist_to_diamond == float('inf'): continue

        score = -dist_to_diamond + (diamond_points * 0.01)

        diamond_targets.append({
            "obj": diamond, "distance": dist_to_diamond, "points": diamond_points,
            "score_value": score,
```

```
            "type": "diamond",
            "teleporter_entry": tp_entry_to_diamond
        })

    diamond_targets.sort(key=lambda x: x["score_value"], reverse=True)
    return diamond_targets

  def _evaluate_tackle_targets_deprioritized(self, board_bot: GameObject, other_bots:
List[GameObject],
                                teleporters: List[GameObject]) -> List[Dict[str, Any]]:
    my_props = board_bot.properties
    current_pos = board_bot.position
    inventory_size = getattr(my_props, 'inventory_size', INVENTORY_SIZE_DEFAULT)
    my_current_diamonds = getattr(my_props, 'diamonds', 0)
    tackle_targets = []

    for enemy_bot in other_bots:
        if not hasattr(enemy_bot, 'properties') or enemy_bot.properties is None: continue
        enemy_diamonds = getattr(enemy_bot.properties, 'diamonds', 0)

        if enemy_diamonds >= MIN_DIAMONDS_TO_TACKLE and my_current_diamonds <
inventory_size:
            dist_to_enemy, tp_entry_to_enemy, _ = self._calculate_path(current_pos,
enemy_bot.position, teleporters)
            if 0 < dist_to_enemy <= MAX_DISTANCE_TO_CONSIDER_TACKLE:

                tackle_score = (enemy_diamonds) - dist_to_enemy - TACKLE_SCORE_PENALTY
                tackle_targets.append({
                    "obj": enemy_bot, "distance": dist_to_enemy, "points": enemy_diamonds,
                    "score_value": tackle_score, "type": "tackle",
                    "teleporter_entry": tp_entry_to_enemy
                })
    tackle_targets.sort(key=lambda x: x["score_value"], reverse=True)
    return tackle_targets

  def _should_return_to_base(self, board_bot: GameObject, board: Board,
                        teleporters: List[GameObject],
                        best_next_item_overall: Optional[Dict[str,Any]]) -> Tuple[bool,
Optional[Position], List[Tuple[int,int]]]:
```

```
        props = board_bot.properties
        current_pos = board_bot.position
        inventory_size = getattr(props, 'inventory_size', INVENTORY_SIZE_DEFAULT)
        my_current_diamonds = getattr(props, 'diamonds', 0)

        if my_current_diamonds == 0:
            return False, None, []

        dist_to_base, tp_entry_for_base_obj, _ = self._calculate_path(current_pos, props.base,
teleporters)
        time_left_seconds = props.milliseconds_left / 1000 if props.milliseconds_left is not None else
float('inf')

        must_return = False
        if my_current_diamonds >= inventory_size: must_return = True
        elif (dist_to_base + SAFE_RETURN_BUFFER_SECONDS) >= time_left_seconds: must_return
= True
        elif my_current_diamonds == inventory_size - 1 and best_next_item_overall:
            if best_next_item_overall["type"] == "diamond" and best_next_item_overall["points"] == 2:
must_return = True

        if not must_return: return False, None, []

        # Tentukan jalur dan hindaran untuk kembali ke base
        goal_for_base_return: Optional[Position] = props.base
        avoid_for_base_return_tuples: List[Tuple[int,int]] = []
        other_bots_on_board = [bot for bot in board.bots if bot.id != board_bot.id]
        for ob in other_bots_on_board: avoid_for_base_return_tuples.append((ob.position.x,
ob.position.y))

        if tp_entry_for_base_obj and tp_entry_for_base_obj.position:
            # Hitung ulang jarak efektif via teleporter untuk kepastian
            tp_exit_for_base = self._get_teleporter_pair(tp_entry_for_base_obj, teleporters)
            if tp_exit_for_base and tp_exit_for_base.position:
                path_to_base_via_tp_dist = self._get_manhattan_distance(current_pos,
tp_entry_for_base_obj.position) + \
                                           self._get_manhattan_distance(tp_exit_for_base.position, props.base)
                if path_to_base_via_tp_dist < dist_to_base:
                    if not self._position_equals(current_pos, tp_entry_for_base_obj.position):
```

```
                goal_for_base_return = tp_entry_for_base_obj.position

        # Atur hindaran teleporter
        if goal_for_base_return == (tp_entry_for_base_obj.position if tp_entry_for_base_obj else None)
:
            for t in teleporters:
                if not self._position_equals(t.position, goal_for_base_return):
                    avoid_for_base_return_tuples.append((t.position.x, t.position.y))
        else:
            for t in teleporters:
                avoid_for_base_return_tuples.append((t.position.x, t.position.y))

        if goal_for_base_return and goal_for_base_return != props.base:
            if (goal_for_base_return.x, goal_for_base_return.y) in avoid_for_base_return_tuples:
                avoid_for_base_return_tuples.remove((goal_for_base_return.x, goal_for_base_return.y))

        return True, goal_for_base_return, avoid_for_base_return_tuples

    def next_move(self, board_bot: GameObject, board: Board) -> Tuple[int, int]:
        props = board_bot.properties
        current_pos = board_bot.position

        red_buttons = [obj for obj in board.game_objects if obj.type == "DiamondButtonGameObject"]
        teleporters = [obj for obj in board.game_objects if obj.type == "TeleportGameObject"]
        diamonds = board.diamonds
        other_bots = [bot for bot in board.bots if bot.id != board_bot.id]

        diamond_targets = self._evaluate_diamonds_by_distance_priority(current_pos, diamonds,
teleporters)
        tackle_targets = self._evaluate_tackle_targets_deprioritized(board_bot, other_bots, teleporters)

        all_targets = diamond_targets + tackle_targets
        all_targets.sort(key=lambda x: x["score_value"], reverse=True)

        best_overall_target: Optional[Dict[str, Any]] = None
        if all_targets:
            best_overall_target = all_targets[0]

        must_return, base_goal_pos, base_avoid_tuples = \
```

```
            self._should_return_to_base(board_bot, board, teleporters, best_overall_target)

        if must_return and base_goal_pos is not None:
            self.goal_position = base_goal_pos
            return self._get_direction_advanced(current_pos, self.goal_position, base_avoid_tuples,
board.width, board.height)

        current_action_avoid_tuples = [(b.position.x, b.position.y) for b in other_bots]

        if best_overall_target and best_overall_target["score_value"] > - (board.width + board.height) * 2
:
            target_object = best_overall_target["obj"]
            self.goal_position = target_object.position

            tp_entry_for_target_obj = best_overall_target.get("teleporter_entry")
            if tp_entry_for_target_obj and tp_entry_for_target_obj.position and \
              not self._position_equals(current_pos, tp_entry_for_target_obj.position):
               self.goal_position = tp_entry_for_target_obj.position

            if best_overall_target["type"] == "tackle":
                current_action_avoid_tuples = [(b.position.x, b.position.y) for b in other_bots if b.id !=
target_object.id]
                for t in teleporters: current_action_avoid_tuples.append((t.position.x,t.position.y))
            else:
                for t in teleporters:
                    if not self._position_equals(t.position, self.goal_position):
                        current_action_avoid_tuples.append((t.position.x,t.position.y))

                if tp_entry_for_target_obj and self.goal_position and
self._position_equals(self.goal_position, tp_entry_for_target_obj.position):
                    if (self.goal_position.x, self.goal_position.y) in current_action_avoid_tuples:
                        current_action_avoid_tuples.remove((self.goal_position.x, self.goal_position.y))

            if self.goal_position:
                return self._get_direction_advanced(current_pos, self.goal_position,
current_action_avoid_tuples, board.width, board.height)

        if red_buttons and (not diamonds or len(diamonds) <
LOW_DIAMOND_THRESHOLD_FOR_RED_BUTTON):
```

```
        self.goal_position = red_buttons[0].position
        avoid_for_red = [(b.position.x, b.position.y) for b in other_bots]
        for t in teleporters: avoid_for_red.append((t.position.x,t.position.y))
        if self.goal_position:
            return self._get_direction_advanced(current_pos, self.goal_position, avoid_for_red,
board.width, board.height)

    return 0,0
```

2. Penjelasan Alur Program

Secara umum, pada setiap giliran, bot akan melakukan serangkaian evaluasi dan keputusan untuk menentukan langkah terbaik berikutnya:

- Inisialisasi dan Pengumpulan Informasi Awal:
  - Bot mendapatkan status dirinya saat ini (posisi, jumlah berlian di inventaris, sisa waktu, posisi markas) dari objek **board_bot**.
  - Bot mengumpulkan informasi semua objek relevan dari papan permainan **(board):** daftar berlian, daftar teleporter, daftar bot lain, dan daftar tombol merah.

- Evaluasi Target Berlian **(_evaluate_diamonds_by_distance_priority):**
  - Untuk setiap berlian yang ada di papan:
  - Hitung jarak efektif dari posisi bot saat ini ke berlian tersebut. Perhitungan ini menggunakan fungsi **_calculate_path** yang mempertimbangkan jalur langsung dan semua kemungkinan jalur melalui pasangan teleporter, lalu memilih jalur terpendek.
  - Berikan "skor" pada berlian. Skor ini didesain agar jarak terpendek menjadi prioritas utama. Skor dihitung sebagai: **score = -jarak_efektif + (poin_berlian * 0.01).**
  - Komponen **-jarak_efektif** memastikan jarak yang lebih kecil menghasilkan skor yang lebih tinggi.
  - Komponen **(poin_berlian * 0.01)** (dimana **poin_berlian** bisa 1 untuk biru dan 2 untuk merah) berfungsi sebagai *tie-breaker*. Jika dua berlian memiliki jarak yang sama persis, berlian merah akan memiliki skor sedikit lebih tinggi dan akan lebih diprioritaskan.
  - Hasilnya adalah daftar berlian (**diamond_targets**) yang telah diberi skor dan informasi jalur, diurutkan berdasarkan skor tertinggi (jarak terdekat, dengan prioritas merah jika jarak sama).

- Evaluasi Target *Tackle* **(_evaluate_tackle_targets_deprioritized):**
  - Bot mengevaluasi semua bot lawan sebagai target potensial untuk di-*tackle*.

- ➢ Sebuah bot lawan dipertimbangkan jika membawa sejumlah berlian di atas batas minimal (**MIN_DIAMONDS_TO_TACKLE**) dan berada dalam jarak maksimal (**MAX_DISTANCE_TO_CONSIDER_TACKLE**).
- ➢ Skor untuk *tackle* sengaja dibuat sangat rendah dengan menambahkan penalti besar (**TACKLE_SCORE_PENALTY**). Hal ini memastikan bahwa dalam strategi saat ini, *tackle* hampir tidak akan pernah menjadi prioritas di atas pengambilan berlian.
- ➢ Hasilnya adalah daftar target *tackle* (**tackle_targets**) yang diurutkan berdasarkan skornya (yang sangat rendah).

❖ Penggabungan dan Pemilihan Target Keseluruhan **(next_move):**
- ➢ Daftar **diamond_targets** dan **tackle_targets** digabungkan menjadi satu daftar **all_targets**.
- ➢ Daftar **all_targets** diurutkan kembali berdasarkan **score_value** secara descending (skor tertinggi dulu).
- ➢ Target dengan skor tertinggi dari daftar gabungan ini dipilih sebagai **best_overall_target**. Karena skor *tackle* sangat rendah, **best_overall_target** hampir selalu akan berupa berlian terdekat.

❖ Keputusan Kembali ke Markas **(_should_return_to_base):**
- ➢ ni adalah logika dengan prioritas tinggi dan dievaluasi setelah **best_overall_target** diketahui (karena salah satu kondisi kembali ke markas bergantung pada target berikutnya).

❖ Eksekusi Target Terbaik (Jika Tidak Kembali ke Markas):
- ➢ Jika **best_overall_target** ada dan skornya "cukup baik" (tidak terlalu negatif, yang mungkin terjadi jika hanya ada target *tackle* dengan skor sangat rendah):
- ➢ Jika jalur terbaik ke target tersebut adalah melalui teleporter (informasi ada di **best_overall_target.get("teleporter_entry"))**, dan bot belum berada di posisi teleporter masuk tersebut, maka **self.goal_position** diubah menjadi posisi teleporter masuk.
- ➢ **self.goal_position** diatur ke posisi objek target **(best_overall_target["obj"].position).**
- ➢ Gerakan **(delta_x, delta_y)** dihitung menggunakan **_get_direction_advanced** dan dikembalikan. Alur berhenti di sini untuk giliran ini.

4.2 Struktur Data yang Digunakan

❖ **List[Dict[str, Any]]** digunakan secara ekstensif untuk menyimpan daftar target yang telah dievaluasi (misalnya, dalam **diamond_targets, tackle_targets, dan all_targets).** Setiap elemen dalam list ini adalah sebuah *dictionary* Python. Penggunaan *dictionary* memungkinkan penyimpanan berbagai atribut untuk setiap target secara terstruktur dan mudah diakses menggunakan kunci (key). Atribut yang umum disimpan meliputi:

- ➢ **"obj"**: Referensi ke objek **GameObject** dari target tersebut (misalnya, objek berlian atau objek bot musuh).
- ➢ **"distance"**: Jarak efektif terhitung dari posisi bot saat ini ke target.
- ➢ "points": Jumlah poin yang dimiliki berlian, atau jumlah berlian yang dibawa bot musuh (untuk target tackle).
- ➢ **"score_value":** Skor numerik yang dihitung untuk target tersebut, yang digunakan untuk menentukan prioritas. Semakin tinggi skor, semakin menarik target tersebut.
- ➢ **"type":** Sebuah string yang mengidentifikasi jenis target, misalnya "**diamond"** atau **"tackle".**
- ➢ **"teleporter_entry":** Jika jalur terbaik ke target adalah melalui teleporter, ini menyimpan referensi ke objek **GameObject** dari teleporter masuk yang harus dituju. Jika jalur langsung lebih baik, nilainya adalah **None.**

- ❖ **Optional[Position]** digunakan untuk atribut **self.goal_position**. Ini adalah objek Position (dari **game.models**) yang merepresentasikan koordinat **(x,y)** dari tujuan akhir pergerakan bot pada langkah saat itu. Bisa berupa posisi berlian, markas, tombol merah, atau teleporter masuk. Tipe **Optional** menandakan bahwa atribut ini bisa saja bernilai **None**, misalnya jika bot tidak memiliki tujuan yang jelas atau sedang diam.

- ❖ **List[Tuple[int,int]]** digunakan untuk **avoid_positions_tuples** dan **current_action_avoid_tuples**. Struktur ini menyimpan daftar koordinat **(x, y)** dari posisi-posisi di papan yang harus dihindari oleh bot saat bergerak. Penggunaan tuple **(x,y)** untuk koordinat adalah cara standar dan efisien untuk merepresentasikan posisi dalam grid.

- ❖ Objek dari **game.models**: Bot juga secara langsung menggunakan kelas-kelas yang didefinisikan dalam **game.models.py** yang disediakan oleh *starter pack*:
  - ➢ **GameObject**: Kelas dasar untuk semua entitas interaktif di papan permainan.
  - ➢ **Board:** Merepresentasikan seluruh papan permainan dan berisi daftar semua **GameObject** yang ada.
  - ➢ **Position:** Struktur data sederhana untuk menyimpan koordinat x dan y.
  - ➢ **Properties:** Menyimpan atribut spesifik dari **GameObject,** seperti **points** pada berlian atau **diamonds** dan **base** pada objek bot.

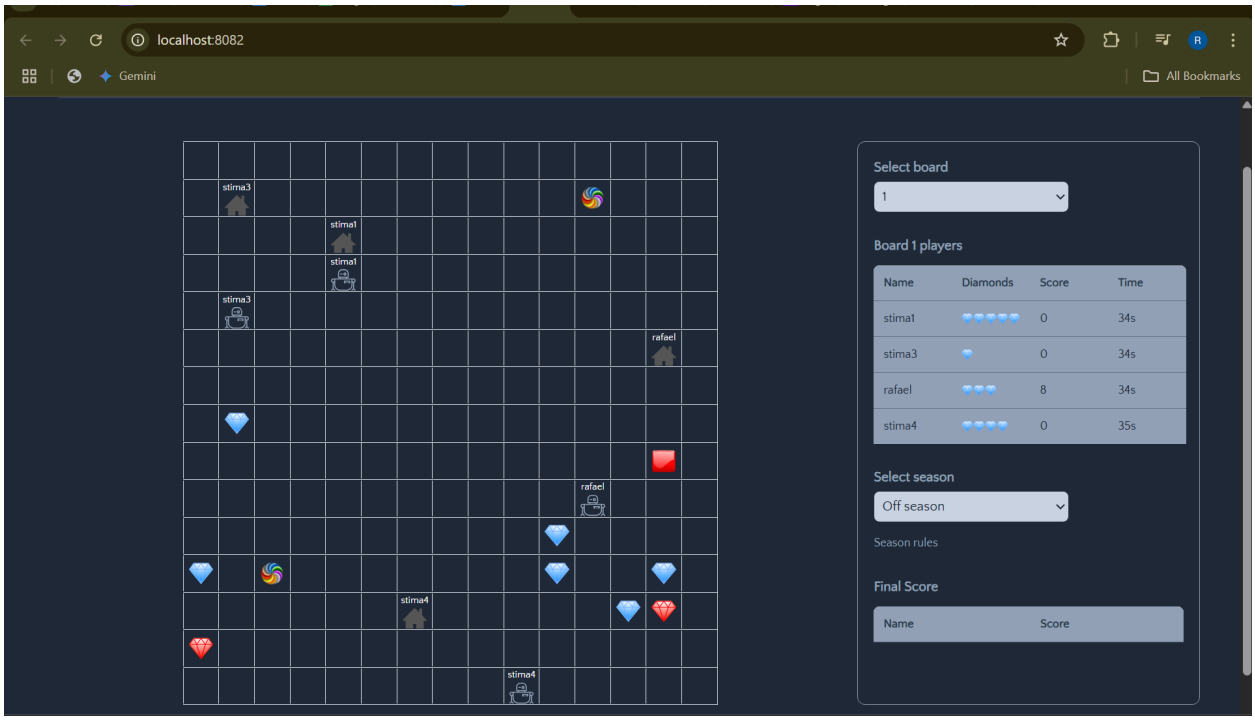
Penggunaan kombinasi list dan dictionary ini memungkinkan pengelolaan dan pengurutan target secara fleksibel, sementara objek dari **game.models** menyediakan representasi standar untuk elemen-elemen permainan.

4.3 Pengujian Program

1. Skenario Pengujian

analisis dan Pengujian Kami melakukan analisis dan pengujian terhadap SpuerBot (bot yang dibuat oleh kelompok kami) dengan bot referensi dari logic yang lainnya. Hasil dari

pengujian dapat berubah dengan dinamis karena peta pada permainan bisa berubah-ubah setiap permainan dimulai kembali. Kami melakukan pengujian pada beberapa aksi yang dapat dilakukan oleh bot



2. Hasil Pengujian dan Analisis

Kami juga menaruh bot kami pada posisi pertama dan ketiga, dan dari hasil pengujian ternyata urutan juga berpengaruh pada poin yang didapatkan. Poin bot kami pada urutan ketiga (Rafael) memiliki total skor yang lebih besar daripada bot kami pada urutan pertama. Hal ini bisa terjadi karena waktu yang dimiliki oleh bot terakhir lebih banyak sehingga waktu untuk mentackle atau menghindari lawan juga lebih banyak dan berlian yang didapat juga bisa lebih banyak daripada saat bot berada di posisi pertama. Berikut hasil rekap poin pertandingan yang didapat

Select board

1

Board 1 players

Name	Diamonds	Score	Time

Select season

Off season

Season rules

Final Score

Name	Score
rafael	18
stima1	10
stima3	1
stima4	0

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Dalam pengembangan bot untuk permainan *Diamonds*, algoritma **Greedy** terbukti menjadi pendekatan yang efektif untuk pengambilan keputusan secara cepat dan efisien. Strategi yang digunakan, yaitu gabungan antara *Greedy by Distance*, *Greedy by Points*, dan *Greedy by Tackle*, memungkinkan bot untuk mengoptimalkan perolehan poin dalam waktu terbatas dengan tetap mempertimbangkan risiko seperti tackle dan inventory penuh.

Implementasi bot menunjukkan bahwa strategi gabungan ini memberikan keseimbangan antara kecepatan dalam mengumpulkan diamond, efisiensi dalam pemilihan target, serta respons terhadap dinamika permainan seperti kemunculan lawan dan regenerasi diamond. Hasil pengujian menunjukkan bahwa posisi awal bot, waktu yang dimiliki, serta taktik tackling memiliki dampak signifikan terhadap performa bot dalam permainan.

5.2 Saran

- Bot sebaiknya dikembangkan lebih lanjut agar mampu beradaptasi secara dinamis terhadap kondisi real-time di papan permainan, seperti perubahan distribusi diamond atau posisi lawan.
- Perlu ditambahkan sistem prioritas adaptif, agar bot bisa menyesuaikan strateginya (misal, lebih defensif saat inventory penuh atau lebih agresif saat waktu hampir habis).
- Untuk meningkatkan efisiensi, penggunaan machine learning atau reinforcement learning bisa dipertimbangkan ke depannya agar bot dapat belajar dari permainan sebelumnya dan mengoptimalkan keputusannya secara otomatis.

- Visualisasi debug (seperti heatmap pergerakan bot) bisa ditambahkan untuk mempermudah analisis strategi dan pengambilan keputusan bot di setiap tick permainan.

LAMPIRAN

A. **Repository Github**

   https://github.com/14-128-RagilBayuSaputra/Tubes1_3J.git

B. **Video Penjelasan**

   https://drive.google.com/drive/folders/1kr4F72AVB__JeBVutf2FojW7jLAtJtP1?usp=sharing

DAFTAR PUSTAKA

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, dan C. Stein, *Introduction to Algorithms*, edisi ke-3, Cambridge, MA: MIT Press, 2009, Bab 16.2, hlm. 423.

[2] J. Kleinberg dan É. Tardos, *Algorithm Design*, Boston: Pearson/Addison Wesley, 2006, Bab 4, hlm. 123–155.

[3] S. Russell dan P. Norvig, *Artificial Intelligence: A Modern Approach*, edisi ke-3, Upper Saddle River, NJ: Prentice Hall, 2010, Bab 3, hlm. 91–105.

[4] I. Parberry, *Introduction to Game Physics with Box2D*, Boca Raton: CRC Press, 2013, hlm. 67–89.