LAPORAN TUGAS BESAR
IF2211 STRATEGI ALGORITMA

Pemanfaatan Algoritma Greedy dalam Aplikasi Permainan “Galaxio”

Disusun oleh

Moh. Aghna Maysan Abyan 13521076

Noel Christoffel Simbolon 13521096

Jericho Russel Sebastian 13521107

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKOLOGI BANDUNG

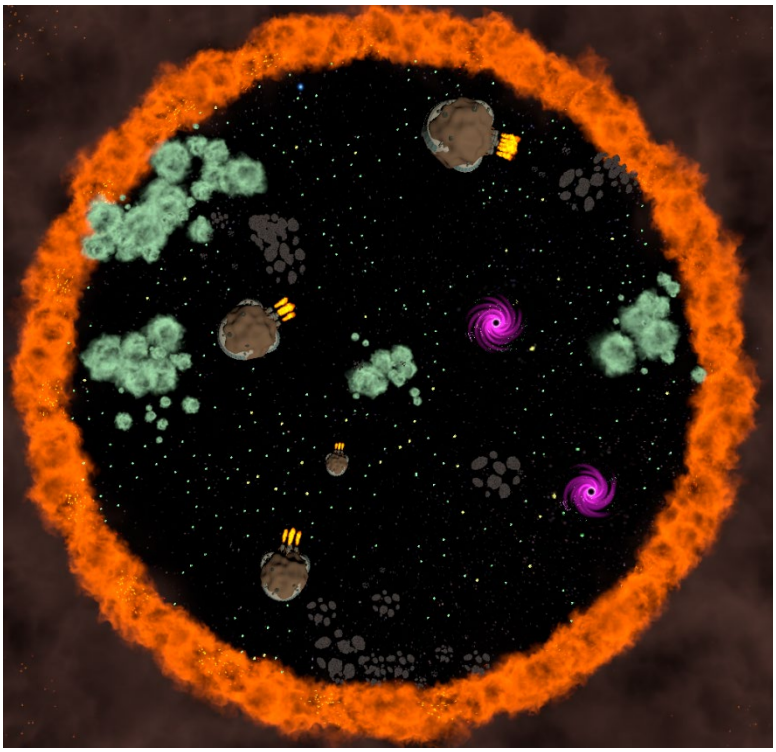
2023

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

1.1 Deskripsi Tugas

Galaxio adalah sebuah game *battle royale* yang mempertandingkan bot kapal anda dengan beberapa bot kapal yang lain. Setiap pemain akan memiliki sebuah bot kapal dan tujuan dari permainan adalah agar bot kapal anda yang tetap hidup hingga akhir permainan. Penjelasan lebih lanjut mengenai aturan permainan akan dijelaskan di bawah. Agar dapat memenangkan pertandingan, setiap bot harus mengimplementasikan strategi tertentu untuk dapat memenangkan permainan (Contributors, 2021).



Gambar 1 Ilustrasi Permainan Galaxio

Pada tugas besar pertama Strategi Algoritma ini, gunakanlah sebuah game engine yang mengimplementasikan permainan Galaxio. *Game engine* dapat diperoleh pada tautan https://github.com/EntelectChallenge/2021-Galaxio.

Tugas mahasiswa adalah mengimplementasikan bot kapal dalam permainan Galaxio dengan menggunakan strategi *greedy* untuk memenangkan permainan. Untuk mengimplementasikan bot tersebut, mahasiswa disarankan melanjutkan program yang terdapat pada starter-bots di dalam starter-pack pada tautan https://github.com/EntelectChallenge/2021-Galaxio/releases/tag/2021.3.2.

Spesifikasi permainan yang digunakan pada tugas besar ini disesuaikan dengan spesifikasi yang disediakan oleh game engine Galaxio pada tautan di atas. Beberapa aturan umum adalah sebagai berikut.

1. Peta permainan berbentuk kartesius yang memiliki arah positif dan negatif. Peta hanya menangani angka bulat. Kapal hanya bisa berada di integer x,y yang ada di peta. Pusat peta adalah 0,0 dan ujung dari peta merupakan radius. Jumlah ronde maximum pada game sama dengan ukuran radius. Pada peta, akan terdapat 5 objek, yaitu Players, Food, Wormholes, Gas Clouds, Asteroid Fields. Ukuran peta akan mengecil seiring batasan peta mengecil.
2. Kecepatan kapal dilambangkan dengan x. Kecepatan kapal akan dimulai dengan kecepatan 20 dan berkurang setiap ukuran kapal bertambah. Ukuran (radius) kapal akan dimulai dengan ukuran 10. Heading dari kapal dapat bergerak antar 0 hingga 359 derajat. Efek afterburner akan meningkatkan kecepatan kapal dengan faktor 2, tetapi mengecilkan ukuran kapal sebanyak 1 setiap tick. Kemudian kapal akan menerima 1 salvo charge setiap 10 tick. Setiap kapal hanya dapat menampung 5 salvo charge. Penembakan slavo torpedo (ukuran 10) mengurangkan ukuran kapal sebanyak 5.
3. Setiap objek pada lintasan punya koordinat x,y dan radius yang mendefinisikan ukuran dan bentuknya. Food akan disebarkan pada peta dengan ukuran 3 dan dapat dikonsumsikan oleh kapal player. Apabila player mengkonsumsi Food, maka Player akan bertambah ukuran yang sama dengan Food. Food memiliki peluang untuk berubah menjadi Super Food. Apabila Super Food dikonsumsi maka setiap makan Food, efeknya akan 2 kali dari Food yang dikonsumsi. Efek dari Super Food bertahan selama 5 tick.
4. Wormhole ada secara berpasangan dan memperbolehkan kapal dari player untuk memasukinya dan keluar di pasangan satu lagi. Wormhole akan bertambah besar setiap tick game hingga ukuran maximum. Ketika Wormhole dilewati, maka wormhole akan mengecil sebanyak setengah dari ukuran kapal yang melewatinya dengan syarat wormhole lebih besar dari kapal player.
5. Gas Clouds akan tersebar pada peta. Kapal dapat melewati gas cloud. Setiap kapal bertabrakan dengan gas cloud, ukuran dari kapal akan mengecil 1 setiap tick game. Saat kapal tidak lagi bertabrakan dengan gas cloud, maka efek pengurangan akan hilang.
6. Torpedo Salvo akan muncul pada peta yang berasal dari kapal lain. Torpedo Salvo berjalan dalam lintasan lurus dan dapat menghancurkan semua objek yang berada pada lintasannya. Torpedo Salvo dapat mengurangi ukuran kapal yang ditabraknya. Torpedo Salvo akan mengecil apabila bertabrakan dengan objek lain sebanyak ukuran yang dimiliki dari objek yang ditabraknya.
7. Supernova merupakan senjata yang hanya muncul satu kali pada permainan di antara quarter pertama dan quarter terakhir. Senjata ini tidak akan bertabrakan dengan objek lain pada lintasannya. Player yang menembakannya dapat meledakannya dan memberi damage ke player yang berada dalam zona. Area ledakan akan berubah menjadi gas cloud.
8. Player dapat meluncurkan teleporter pada suatu arah di peta. Teleporter tersebut bergerak dalam direksi dengan kecepatan 20 dan tidak bertabrakan dengan objek apapun. Player tersebut dapat berpindah ke tempat teleporter tersebut. Harga setiap peluncuran teleporter adalah 20. Setiap 100 tick player akan mendapatkan 1 teleporter dengan jumlah maximum adalah 10.

9. Ketika kapal player bertabrakan dengan kapal lain, maka kapal yang lebih besar akan mengonsumsi oleh kapal yang lebih kecil sebanyak 50% dari ukuran kapal yang lebih besar hingga ukuran maximum dari ukuran kapal yang lebih kecil. Hasil dari tabrakan akan mengarahkan kedua dari kapal tersebut lawan arah.
10. Terdapat beberapa command yang dapat dilakukan oleh player. Setiap tick, player hanya dapat memberikan satu command. Untuk daftar commands yang tersedia, bisa merujuk ke tautan panduan di spesifikasi tugas
11. Setiap player akan memiliki score yang hanya dapat dilihat jika permainan berakhir. Score ini digunakan saat kasus tie breaking (semua kapal mati). Jika mengonsumsi kapal player lain, maka score bertambah 10, jika mengonsumsi food atau melewati wormhole, maka score bertambah 1. Pemenang permainan adalah kapal yang bertahan paling terakhir dan apabila tie breaker maka pemenang adalah kapal dengan score tertinggi.

Adapun peraturan yang lebih lengkap dari permainan Galaxio, dapat dilihat pada tautan https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/game-engine/game-rules.md.

1.2 Spesifikasi Tugas

Pada tugas besar kali ini, anda diminta untuk membuat sebuah bot untuk bermain permainan Galaxio yang telah dijelaskan sebelumnya. Untuk memulai, anda dapat mengikuti panduan singkat sebagai berikut.

1. *Download latest release* starter pack.zip dari tautan berikut https://github.com/EntelectChallenge/2021-Galaxio/releases/tag/2021.3.2.
2. Untuk menjalankan permainan, kalian butuh beberapa requirement dasar sebagai berikut.
   a. Java (minimal Java 11): https://www.oracle.com/java/technologies/downloads/#java
   b. IntelIiJ IDEA: https://www.jetbrains.com/idea/
   c. .Net Core 3.1: *link* tersedia pada panduan berikut
3. Panduan mengenai cara menjalankan permainan, membuat bot, build src code, dan melihat visualizer bisa dicek melalui tautan berikut.
4. Silahkan bersenang-senang dengan memodifikasi bot yang disediakan di starter-bot. Ingat bahwa bot kalian harus menggunakan bahasa Java. Dilarang menggunakan kode program yang sudah ada, mahasiswa wajib membuat program/strategi sendiri. Tetapi, belajar dari program yang sudah ada tidak dilarang.

Strategi greedy yang diimplementasikan tiap kelompok harus dikaitkan dengan fungsi objektif dari permainan itu sendiri, yaitu memenangkan permainan dengan cara mempertahankan kapal pemain paling terakhir untuk hidup. Salah satu contoh pendekatan greedy yang bisa digunakan (pendekatan tak terbatas pada contoh ini saja) adalah menghindari objek yang dapat mengurangi ukuran kapal. Buatlah strategi greedy terbaik, karena setiap bot dari masing-masing kelompok akan diadu dalam suatu kompetisi Tubes 1 (TBD).

Strategi greedy harus dijelaskan dan ditulis secara eksplisit pada laporan, karena akan diperiksa pada saat demo apakah strategi yang dituliskan sesuai dengan yang diimplementasikan. Tiap kelompok dapat menggunakan kreativitas mereka dalam menyusun strategi greedy untuk memenangkan permainan. Implementasi pemain harus dapat dijalankan pada game engine yang telah disebutkan pada spesifikasi tugas besar, serta dapat dikompetisikan dengan pemain dari kelompok lain.

BAB II
LANDASAN TEORI

2.1 Dasar Teori Algoritma Greedy

Algoritma *greedy* merupakan metode yang paling populer dan sederhana untuk memecahkan persoalan optimasi (persoalan mencari solusi optimal). Algoritma *greedy* adalah algoritma yang memecahkan persoalan secara langkah per langkah (*step-by-step*) sedemikian rupa sehingga pada setiap langkah akan 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 (Munir, 2021).

Elemen-elemen algoritma *greedy* adalah sebagai berikut.

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.

Dengan menggunakan elemen-elemen di atas, maka dapat dikatakan bahwa algoritma *greedy* melibatkan pencarian sebuah himpunan bagian, S, dari himpunan kandidat, C; yang dalam hal ini, S harus memenuhi beberapa kriteria yang ditentukan, yaitu S menyatakan suatu solusi dan S dioptimisasi oleh fungsi obyektif.

Pada sebagian persoalan, algoritma *greedy* tidak selalu berhasil memberikan solusi optimal, dikarenakan 2 hal, yaitu:

1. Algoritma *greedy* tidak beroperasi secara menyeluruh terhadap semua kemungkinan solusi yang ada (sebagaimana pada metode *exhaustive search*).
2. Terdapat beberapa fungsi seleksi yang berbeda, sehingga kita harus memilih fungsi yang tepat jika kita ingin algoritma menghasilkan solusi optimal.

Contoh-contoh persoalan yang diselesaikan dengan algoritma *greedy* adalah sebagai berikut.

1. Persoalan penukaran uang (*coin exchange problem*)
2. Persoalan memilih aktivitas (*activity selection problem*)
3. Minimisasi waktu di dalam sistem
4. Persoalan knapsack (*knapsack problem*)
5. Penjadwalan Job dengan tenggat waktu (*job schedulling with deadlines*)
6. Pohon merentang minimum (*minimum spanning tree*)
7. Lintasan terpendek (*shortest path*)

2.2 Cara Kerja Program Secara Umum

Game Galaxio memiliki struktur *turn-based*. Setiap program dari *player* mengirimkan *command* kepada *game engine* yang mengeksekusi *command* tersebut dan meng-*update game state*. Dalam setiap *turn*, atau bisa juga disebut *tick*, *game engine* mengirimkan *snapshot* dari *game state* ke program *player*. *Snapshot*, yang dikirimkan ke semua *player* yang sedang bermain game tersebut, berisi *payload* yang berisi *state* dari *world* dalam permainan, semua objek dalam game, dan semua objek *player* dalam game.

Berdasarkan isi dari *payload* yang dikirimkan *game engine* ke program *player* ini, *program* tiap player akan menentukan *command* apa yang akan dieksekusi berikutnya dan mengirimkan *command* yang ingin dieksekusi tersebut ke *game engine*. Proses penentuan *command* ini bergantung pada algoritma apa yang terimplementasi pada program *player*. Dalam tugas besar ini, kami ditugaskan untuk mendesain serta mengimplementasikan algoritma *greedy* yang bertugas untuk mengevaluasi *command* apa yang akan dijalankan oleh *player*. Untuk dapat mengimplementasikan algoritma *greedy* ini, kami memodifikasi apa yang terjadi pada fungsi `computeNextPlayerAction` yang terdapat pada *file* `BotService.java`. Fungsi ini dipanggil dari *main method* pada *main class* dan merupakan *entry point* dari algoritma *greedy* yang kami implementasikan. Inti dari algoritma *greedy* yang kami implementasikan berada pada *directory* `src/main/java/Rakus`.

Untuk menjalankan permainan, di sistem operasi Windows, perlu dilakukan konfigurasi jumlah bot pada `appsettings.json` yang terdpat pada folder `runner-publish` dan `engine-publish`. Setelah itu, perlu dijalankan *runner*, *game engine*, serta *logger*. Lalu, bot-bot yang ingin dimainkan akan dijalankan. Selanjutnya, untuk dapat melihat game berjalan layaknya *video game* sesungguhnya, perlu dijalankan *visualizer*.

Berikut adalah *batch script* untuk mem-*build* program menjadi JAR *file* menggunakan Maven; menjalankan *runner*, *game engine*, *logger*; menjalankan tiga *reference bot* yang telah disediakan; menjalankan *bot* Rakus yang algoritma *greedy*-nya telah kami implementasikan; serta menjalankan program untuk memvisualisasikan permainan.

```
:: Run this script from the root folder

echo "Building JAR file with Maven..."
start "" mvn clean package

@echo off
:: Game Runner
cd ./starter-pack/runner-publish/
start "" dotnet GameRunner.dll

:: Game Engine
cd ../engine-publish/
timeout /t 1
start "" dotnet Engine.dll

:: Game Logger
cd ../logger-publish/
timeout /t 1
start "" dotnet Logger.dll
```

```
:: Three default bots
cd ../reference-bot-publish/
timeout /t 3
start "ReferenceBot" dotnet ReferenceBot.dll
timeout /t 3
start "ReferenceBot" dotnet ReferenceBot.dll
timeout /t 3
start "ReferenceBot" dotnet ReferenceBot.dll
timeout /t 3

:: Rakus bot
cd ../../target/
start "Rakus" java -jar Rakus.jar

:: Launch visualiser
cd ../starter-pack/visualiser
Galaxio.exe

pause
```

BAB III
APLIKASI STRATEGI GREEDY

3.1 Pemetaan Persoalan ke Elemen-Elemen Algoritma Greedy

Berikut adalah tabel yang berisi pemetaan persoalan permainan Galaxio ke elemen-elemen algoritma *greedy*.

Elemen Algoritma Greedy	Analog Elemen pada Permainan Galaxio
Himpunan Kandidat (C)	`BotState-BotState` yang mungkin terpilih pada tiap *tick*, yaitu `IDLE`, `FEED`, `FLEE_LOW`, `CHASE_LOW`, `CHASE_HIGH`, dan `PICK_SUPERNOVA`
Himpunan Solusi (S)	`BotState` yang terpilih berdasarkan evaluasi algoritma *greedy* dalam setiap *tick*.
Fungsi Solusi	Validasi apakah `BotState` yang terpilih sudah sesuai dengan tujuan yang diinginkan berdasarkan *state* dari permainan, seperti untuk *firing torpedoes*, *prioritizing feeding*, *aggressive attacking*, ataupun tujuan-tujuan lain.
Fungsi Seleksi	Proses pemilihan `BotState` dengan *evaluation score* tertinggi. Proses ini terjadi dalam *method* `getNextState` di *file* `BotState.java`
Fungsi Kelayakan	Proses yang melakukan pengecekan apakah `BotState` pantas untuk dipilih sebagai `BotState` yang dieksekusi oleh bot.
Fungsi Objektif	Memenangkan permainan Galaxio dengan menjadi satu-satunya bot yang *survive* dalam permainan atau menjadi bot yang memiliki skor tertinggi jika jumlah maksimum *tick* tercapai.

3.2 Eksplorasi Alternatif Solusi

Kami mendesain beberapa alternatif solusi yang dalam algoritma *greedy* yang kami implementasikan, yaitu:

1. *Teleporter heavy*
   Dalam alternatif solusi ini, penggunaan *power-up teleporter* akan diutamakan dengan cara memodifikasi konstanta yang berhubungan langsung dalam penentuan keputusan penggunaan *teleporter*, seperti `CHASE_HIGH_TELEPORT_RANGE` dalam file `Vars.java`. Modifikasi ini akan meningkatkan kemungkinan penembakan *teleporter* dalam `BotState-BotState` tertentu. Dalam kasus ini, penggunaan *teleporter* lebih cenderung untuk tujuan menyerang (agresif) sehingga *teleporter* bukan dipakai untuk melarikan diri dari lawan dalam situasi yang genting (defensif).
2. *Supernova is our saviour*
   Dalam alternatif solusi ini, pengambilan *supernova* akan diutamakan menjadi prioritas utama jika *supernova* sudah terdeteksi *spawn*-nya dalam *map*. Hal ini dapat dilakukan dengan cara memodifikasi konstanta yang berhubungan langsung dalam penentuan

keputusan penggunaan *teleporter* untuk pengambilan *supernova*, seperti `PICK_SUPERNOVA_SCOREMULT` dan `PICK_SUPERNOVA_WEIGHT_BIAS` dalam file `Vars.java`. Dengan memodifikasi konstanta-konstanta tersebut, *evaluation score* dari `BotState` `PICK_SUPERNOVA` dapat ditingkatkan dan menyebabkan peningkatan kemungkinan `BotState` tersebut terpilih dalam *tick* yang sedang berjalan.

3. *Get more obese*
   Dalam alternatif solusi ini, bot akan difokuskan untuk *feeding* sejak awal mulai permainan. Bot akan mengutamakan penggendutan diri demi mencapai *size* yang cukup besar untuk dapat melahap bot-bot musuh. Pemrioritasan *feeding* dapat dilakukan dengan cara memodifikasi konstanta yang berhubungan langsung dalam penentuan keputusan pemrioritasan *feeding*, seperti `FEED_SCOREMULT` dalam file `Vars.java`. Dengan memodifikasi konstanta-konstanta tersebut, *evaluation score* dari `BotState` `FEED` dapat ditingkatkan dan menyebabkan peningkatan kemungkinan `BotState` tersebut terpilih dalam *tick* yang sedang berjalan.
4. *Torpedo mania*
   Dalam alternatif solusi ini, bot akan difokuskan untuk menembakkan torpedo yang terdapat pada `BotState`- `BotState` tertentu. Pemrioritasan *torpedo firing* dapat dilakukan dengan cara memodifikasi konstanta yang berhubungan langsung dalam penentuan keputusan *torpedo firing*, seperti `FLEE_LOW_TORPEDO_RANGE`, `CHASE_LOW_TORPEDO_RANGE`, dan `CHASE_HIGH_TORPEDO_RANGE` dalam file `Vars.java`. Dengan memodifikasi konstanta-konstanta tersebut, akan terjadi peningkatan kemungkinan *torpedo* ditembakkan dalam *tick* yang sedang berjalan.
5. *The final form*
   Alternatif solusi ini merupakan hasil dari *balancing* dan *fine-tuning* dari alternatif-alternatif lain. Dengan kata lain, alternatif ini merupakan gabungan dari alternatif-alternatif lain yang telah disebutkan di atas. Untuk mencapai hal ini, setiap konstanta dalam `Vars.java` ditentukan dari hasil *testing*, estimasi, dan logika heuristik. Berbeda dari alternatif-alternatif yang lain, alternatif ini memiliki pemilihan `BotState` yang jauh lebih bervariasi karena kondisi dari permainan akan berubah-ubah sepanjang permainan dan alternatif ini menyesuaikan dengan perubahan kondisi tersebut.

3.3 Analisis Efisiensi dan Efektivitas Alternatif Solusi

Desain solusi yang kami implementasikan menerapkan *approach* berbasis `BotState`. Dalam setiap *tick*, *method* `getNextState` dieksekusi. Dalam *method* ini, *score* dari setiap `BotState` akan dikalkulasi berdasarkan *circumstances* dan kondisi dari permainan dalam setiap *tick*. Oleh karena itu, *lambda function* yang bertindak sebagai *score calculator* dalam tiap `BotState` akan selalu tereksekusi apapun kondisi dari permainan.

Berdasarkan analisis yang telah kami lakukan, *average* dan *worst case scenario* untuk evaluasi *score* dalam setiap *tick* adalah `O(n log n)` dengan `n` adalah jumlah `GameObject` yang sedang ada dalam permainan. Hal ini didasari dari *time complexity* yang dimiliki *method* pembantu `public static <T extends Comparable<? super T>> List<GameObject>`

`findAll(Function<GameObject, ? extends T> comparator, Predicate<GameObject> pred)`. *Method* pembantu ini banyak dieksekusi dalam *method-method* pembantu lain yang dipanggil dalam proses evaluasi *score* tiap `BotState`. Bagian yang membuat *time complexity* dari *method* ini `O(n log n)` adalah `.sorted(Comparator.comparing(comparator))` yang juga memiliki *time complexity* `O(n log n)`.

Berbeda dari efisiensi algoritma (yang seragam *regardless* dari alternatif solusi), efektivitas dari tiap alternatif solusi berbeda-beda tergantung dari kondisi permainan. Untuk alternatif solusi *teleporter heavy*, melakukan teleportasi agresif tidaklah selalu efektif karena untuk melakukan teleportasi, *size* dari bot akan berkurang 20. Angka ini cukup signifikan dalam menentukan nasib bot selanjutnya. Selain itu, teleportasi agresif terhadap bot musuh juga merupakan sebuah langkah *gambling* karena masih ada kemungkinan untuk bot musuh menghindar. Oleh karena itu, alternatif solusi *teleporter heavy* kurang efektif.

Alternatif solusi yang lain adalah *supernova is our saviour*. Mengingat fakta bahwa *supernova* adalah *weapon* yang hanya muncul sekali dalam tiap permainan, pengandalan *supernova* sebagai amunisi utama untuk memenangkan permain bukanlah strategi yang bijak. Selain itu, mendapatkan *supernova* tidak mudah karena bersaing dengan bot-bot musuh lain. Jika ingin mendapatkan *supernova* dengan cepat, satu-satunya cara adalah menggunakan *teleporter*. Namun, *teleporter* memiliki *cost* yang mahal jika digunakan dengan tidak memperhatikan *size* dari bot. Kalaupun bot telah mendapatkan *supernova*, penentuan kapan untuk menembakkan dan men-*detonate supernova* tidaklah sederhana. Oleh karena itu, alternatif solusi *supernova is our saviour* kurang efektif.

Selain itu, alternatif solusi yang lain adalah *get more obese*. Dalam awal-awal permainan, alternatif solusi ini biasanya efektif karena prioritas utama pada awal permainan adalah untuk bertambah besar. Namun, alternatif solusi ini tidak dapat melawan bot-bot musuh yang bersifat agresif. Selain itu, banyak *hazard* yang menjadi bagian dari *world* dalam permainan, seperti *gas clouds* dan *asteroid field*. Jika bot memprioritaskan makanan di atas segalanya, ia akan mengabaikan *hazard-hazard* tersebut yang dapat menyebabkan *size* bot yang mengecil. Oleh karena itu, alternatif solusi *get more obese* kurang efektif.

Alternatif solusi lainnya adalah *torpedo mania*. Untuk dapat menembakkan satu *torpedo*, 5 unit dari *bot size* harus dikonsumsi. Namun, dari awal permainan, *bot size* tidaklah besar, jadi untuk memprioritaskan penembakan *torpedo* sejak awal permainan bukanlah strategi yang bijak karena *size* dari bot akan sulit bertambah jika selalu dikonsumsi akibat penembakan *torpedo*. Selain itu, batasan lainnya dari alternatif solusi ini adalah adanya amunisi maksimal *torpedo*, yaitu lima. Ditambah lagi, bot hanya mendapatkan satu amunisi *torpedo* dalam 10 *tick*. Belum lagi jika kita mempertimbangkan *collision* dari *torpedo*. Oleh karena itu, alternatif solusi *torpedo mania* kurang efektif.

Alternatif solusi terakhir, *the final form*, merupakan hasil penggabungan keempat alternatif sebelumnya, disertai berbagai tambahan dan pengembangan. Dengan bobot setiap aksi yang berimbang dan memperhatikan konteks keadaan permainan, alternatif ini merupakan yang terbaik dan menggabungkan keunggulan keempat alternatif lainnya, sembari mengisi kekurangan yang ada di setiap alternatif. Bot yang mengikuti solusi *the final form* akan

berusaha mengimbangi *size* dan agresi terhadap lawan, memastikan keamanan daerah di sekitarnya, dan mampu memanfaatkan *supernova* jika diberikan kesempatan.

3.4 Strategi Greedy yang Dipilih

Nature dari permainan Galaxio adalah *map generation* yang acak dan kondisi permainan tidak dapat diprediksi. Strategi algoritma tertentu dapat menjadi efektif dalam kondisi permainan tertentu, tetapi juga dapat menjadi kurang efektif jika kondisinya berubah. Oleh karena itu, agar algoritma yang diimplementasi optimal, diperlukan algoritma yang performanya konsisten dalam tiap permainan.

Dengan mempertimbangkan banyak hal yang dapat terjadi pada permainan dan melakukan berbagai *testing*, kami mengombinasikan bagian-bagian dari alternatif solusi dan menyatukannya untuk membangun sebuah solusi yang cukup optimal, yaitu alternatif kelima: *the final form*.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma Greedy

```
{ Mencerminkan proses utama bot Rakus dalam mengikuti permainan;
  Pseudocode ini merupakan oversimplification dan hanya merupakan analogi
  dari cara kerja bot Rakus; prosedur ini mengabaikan banyak faktor seperti
  self-update, state ANY, dll. dan bukan merupakan representasi one-to-one
  dari kode bot Rakus }
procedure rakusBot(KANDIDAT: array of BotState)
{ KAMUS }
stateDipilih: BotState

{ ALGORITMA }
{ Loop hingga permainan berakhir (SOLUSI) }
while not PERMAINAN_BERAKHIR do
    { Pilih state terbaik untuk tick ini (SELEKSI) }
    stateDipilih <- getNextState(KANDIDAT)

    { Dalam kode bot Rakus sebenarnya, KELAYAKAN diperiksa bersamaan dengan
      SELEKSI: state yang tidak LAYAK tidak akan dipilih oleh SELEKSI }
    if LAYAK(stateDipilih) then
        { Mainkan state ini }
        stateDipilih.AKSI()
```

```
{ Mengembalikan state selanjutnya dari bot }
function getNextState(KANDIDAT: array of BotState): BotState
{ KAMUS }
current, newState: BotState
currentEval, newEval: integer

{ ALGORITMA }
{ Pilih state pertama sebagai asumsi maksimum sekarang }
current <- KANDIDAT[1]
currentEval <- current.EVALUASI()

{ Iterasi semua state yang ada, hitung skor evaluasi }
i traversal [2..N_STATE] do
    newState <- SEMUA_STATE[i]
    newEval <- newState.EVALUASI()
    if newEval > currentEval then
        { State ini lebih baik: pilih state ini }
        current <- newState
        currentEval <- newEval

{ State terbaik sudah ditemukan }
-> current
```

4.2 Struktur Data

Permainan Galaxio, secara *default*, memiliki struktur data yang cukup kompleks, terdiri dari sebuah *main class* dan tiga *package*: `Enums`, `Models`, dan `Services`. Namun, kami menambahkan satu *package* tambahan untuk mengimplementasikan algoritma *greedy*, yaitu `Rakus`. Alasan penambahan *package* terpisah ini agar algoritma *greedy* kami lebih modular, lebih terbaca, dan lebih mudah diidentifikasi. Berikut merupakan penjelasan struktur data yang terdapat pada permainan Galaxio serta struktur data tambahan yang kami implementasikan.

1. *Package* `Enums`
    a. *Enum* `ObjectTypes`
       Berisi *constant objects* yang merepresentasikan tipe-tipe objek dalam permainan.
    b. *Enum* `PlayerActions`
       Berisi *constant objects* yang merepresentasikan tipe-tipe aksi *player* dalam permainan.
2. *Package* `Models`
    a. *Class* `GameObject`
       Berisi atribut-atribut yang dimiliki setiap objek dalam permainan serta *method-method constructor*, *setter*, dan *getter* untuk atribut-atribut tersebut.
    b. *Class* `GameState`
       Berisi atribut yang merepresentasikan dunia permainan, objek permainan (*nonplayer*), dan objek permainan (*player*) serta *method-method constructor*, *setter*, dan *getter* untuk atribut-atribut tersebut.
    c. *Class* `GameStateDto`
       Berisi atribut-atribut yang berguna sebagai ”perantara” *payload* dari *runner* untuk dikonversi ke atribut-atribut *class* GameState, serta berisi *method-method setter* dan *getter* untuk atribut-atribut “perantara” tersebut.
    d. *Class* `PlayerAction`
       Berisi atribut-atribut yang merepresentasikan aksi-aksi yang dapat dilakukan *player* dalam permainan serta *method-method setter* dan *getter* untuk atribut-atribut tersebut.
    e. *Class* `Position`
       Berisi atribut-atribut yang merepresentasikan lokasi objek-objek permainan dalam *map* serta *method-method constructor*, *setter*, dan *getter* untuk atribut-atribut tersebut.
    f. *Class* `World`
       Berisi atribut-atribut yang merepresentasikan dunia permainan serta *method-method setter* dan *getter* untuk atribut-atribut tersebut.
3. *Package* `Rakus`
    a. *Interface* `ActionFunc`
       Merupakan *functional interface* yang bertindak sebagai “*template*” dari *lambda expression* yang menspesifikasikan aksi bot dalam struktur data `BotState`.
    b. *Interface* `GameEvaluator`

Merupakan *functional interface* yang bertindak sebagai "*template*" dari *lambda expression* yang mengevaluasi *score* dalam struktur data `BotState`.

c. *Enum* `BotState`
Berisi struktur data `BotState-BotState` dalam bentuk *enum* yang tiap `BotState`-nya berisi dua *lambda expression*: untuk evaluasi *score* dan spesifikasi aksi bot, serta berisi *method* untuk menyeleksi `BotState` yang dijalankan sebagai aksi bot dalam tiap *tick*-nya.

d. *Enum* `Effects`
Berisi representasi *active effects* dari bot.

e. *Class* `Objects`
Berisi *method-method* pembantu dalam mengimplementasikan algoritma *greedy*.

f. *Class* `Vars`
Berisi konstanta-konstanta yang bersifat variabel untuk melakukan *fine-tuning* dari *behaviour* bot.

4. *Package* `Services`
   a. *Class* `BotService`
   Berisi semua hal yang dibutuhkan untuk mengontrol *behaviour* dari bot. *Class* ini memuat *method* yang memanggil evaluasi aksi bot yang kami implementasikan dalam algoritma *greedy* pada *package* `Rakus`.
5. *Class* `Main`
   Berisi *main method* yang berfungsi sebagai *entry point* dari permainan dan pengatur *flow* dari permainan.

4.3 Analisis Desain Solusi

Meskipun pemilihan *bot state* merupakan proses yang deterministik (tanpa menggunakan variabel acak), perhitungan prioritas suatu *bot state* sangat bergantung kepada keadaan permainan saat ini (*game state*). Ketidakmampuan bot melihat jauh ke masa depan akibat sifat bot yang *greedy* menjadikan progresi *game state* sebagai proses yang tidak dapat diprediksi (*chaotic*). Sehingga, efektivitas solusi yang diterapkan oleh bot, seberapapun baiknya, akan selalu dipengaruhi secara acak oleh *game state*. Banyak kasus *game state* yang dapat ditangani oleh bot dengan sangat baik, dan banyak pula kasus lainnya di mana bot merespon dengan buruk dan merugikan posisinya sendiri dalam permainan.

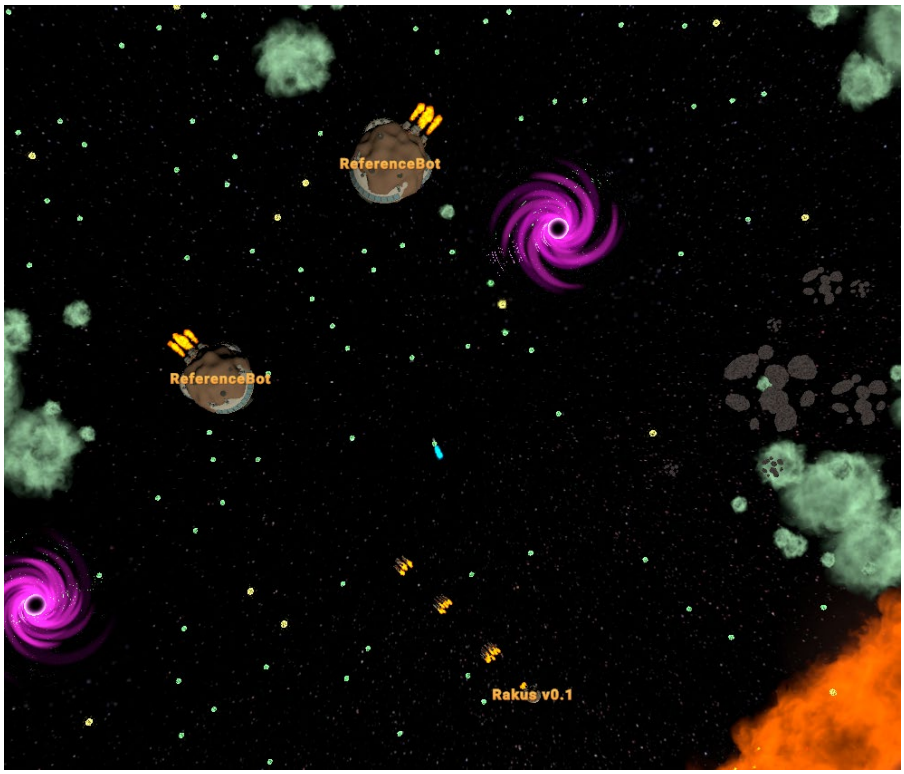
Gambar 2 Kasus Pertarungan Antara Dua Bot

Pada kasus ini, terdapat dua *instance* dari bot yang menggunakan implementasi algoritma yang sama. Bot A (kanan) mengevaluasi *game state* ini, di mana ukuran A lebih besar dari ukuran lawan-lawannya, sebagai kesempatan yang baik untuk bertindak secara agresif, dan memberikan respon berupa tembakan torpedo. Bot B (kiri) mengenali klaster torpedo dari A sebagai bahaya dan menanggapinya dengan mengaktifkan *shield*. Hal ini menunjukkan bahwa bot memiliki kinerja yang cukup baik dalam mengenali dan mengikuti taktik dasar seperti *offense* dan *defense*.

Gambar 3 Kasus Penggunaan Teleport Secara Ofensif

Kondisi di atas merupakan keadaan *endgame* di mana hanya tersisa dua bot yang masih aktif dalam permainan. Setelah mengaktifkan *shield* dan mengevaluasi *game state*, bot I (kiri) menyadari bahwa ukurannya lebih dari ukuran bot J (kanan) ditambah biaya menembakkan *teleporter*. Berdasarkan fakta ini, bot I memutuskan untuk menggunakan *teleporter* untuk

segera menangkap bot J. Taktik ini sangat efektif untuk menangkap lawan karena *teleporter* bergerak lebih cepat dari bot (di atas ukuran tertentu). Tujuan dari taktik ini adalah untuk mencegah lawan mendapatkan kesempatan bertumbuh dan mengakhiri permainan dengan cepat.



Gambar 4 Kasus Penggunaan Teleport untuk Meraih Supernova

Pada kasus ini, bot tersudutkan di bagian bawah *world*. Terdapat sebuah *supernova pickup* di pusat *world* (atas), namun jalur antara bot dan *supernova* dihalangi oleh dua bot lawan. Dapat diamati pada gambar bahwa bot memutuskan untuk menembakkan sebuah *teleporter* untuk melewati bot lawan dan mencapai pusat *world*, dengan rencana menembakkan *supernova* untuk memperkecil ukuran lawan dan menyeimbangkan posisi. Ini merupakan taktik yang efektif namun berisiko, karena menembakkan *teleporter* menghabiskan cukup banyak sumber daya dan ada kemungkinan bahwa pusat *world* dikuasai sepenuhnya oleh lawan, mencegah bot untuk mencapai *supernova*.

Gambar 5 Kasus Penggunaan Supernova

Gambar di atas menggambarkan kasus di mana bot baru saja mendapatkan sebuah *supernova*. Bot kemudian langsung menembakkan *supernova* ke arah lawan dengan ukuran terbesar, dengan harapan bahwa ledakan *supernova* dan *gas cloud* yang dihasilkan akan mengurangi ukuran lawan secara signifikan. Meskipun pada kasus ini penggunaan *supernova* merupakan taktik yang efektif, terdapat kasus lain di mana penggunaan *supernova* tidak akan memberikan dampak yang signifikan terhadap keadaan permainan. Dalam kasus-kasus seperti itu, *supernova* lebih baik disimpan untuk digunakan di lain kesempatan secara ofensif atau dalam keadaan darurat.

Dengan mempertimbangkan keempat kasus di atas, tanpa mengabaikan kasus-kasus lain yang tidak diilustrasikan, dapat disimpulkan bahwa strategi algoritma yang digunakan sudah memberikan hasil yang hampir optimal dalam mayoritas kasus dalam permainan. Menggunakan algoritma yang diimplementasikan, bot mampu meraih keunggulan dalam hampir seluruh permainan melawan *reference bot*. Namun, mengingat bahwa ruang *state* permainan Galaxio sangat besar, dan *reference bot* tidak menggunakan algoritma yang sangat mendalam untuk membentuk sifatnya, lumrah bahwa hasil yang berbeda dapat dihasilkan dalam beberapa kasus yang mungkin terjadi dalam permainan melawan bot lain yang lebih kuat taktik dan strategi permainannya dibandingkan dengan *reference bot*.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Kami berhasil mengimplementasikan algoritma *greedy* untuk meng-*improve* algoritma *template* yang disediakan oleh Entelect Challenge. Algoritma *greedy* yang kami implementasikan memiliki *approach* untuk memilih `BotState` dengan *evaluation score* tertinggi dan mengeksekusi aksi yang terspesifikasi dalam tiap `BotState`-nya. Berdasarkan analisis alternatif strategi yang telah dilakukan, kami memilih alternatif *the final form* karena kami menemukan strategi tersebut merupakan strategi terefektif yang kami temukan untuk mencapai objektif permainan Galaxio.

5.2 Saran

Algoritma *greedy* yang kami desain dan implementasikan masih memiliki *room for improvement*. Untuk dapat mengembangkannya lebih lanjut, perlu dilakukan pengujian dengan mempertarungkan bot Rakus dengan bot-bot yang memiliki algoritma lebih bervariasi. Dengan melakukan hal tersebut, dapat lebih dipahami strategi-strategi algoritma lain sehingga algoritma bot Rakus dapat dikembangkan untuk mengantisipasi bot-bot dengan strategi yang lebih *robust* dan unik.

DAFTAR PUSTAKA

Contributors, E. C. (2021, August 17). *Entelect Challenge 2021 - Galaxio*. Retrieved from GitHub: https://github.com/EntelectChallenge/2021-Galaxio/

Munir, R. (2021). *IF2211 Strategi Algoritma - Semester II Tahun 2022/2023.* Retrieved from Homepage Rinaldi Munir: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/stima22-23.htm

LAMPIRAN

Tautan *remote repository*

Berikut adalah tautan *remote repository* yang berisi *source code* untuk tugas ini.

https://github.com/noelsimbolon/Tubes1_Rakus/

Tautan video

Berikut adalah tautan video untuk tugas ini.

https://bit.ly/TheDawnOfRakus