

Tugas Besar I IF2211 Strategi Algoritma

Semester II Tahun 2022/2023

**Pemanfaatan Algoritma Greedy dalam Aplikasi Permainan  
“Galaxio”**



Disusun oleh:

Athif Nirwasito 13521053

Louis Caesa Kesuma 13521069

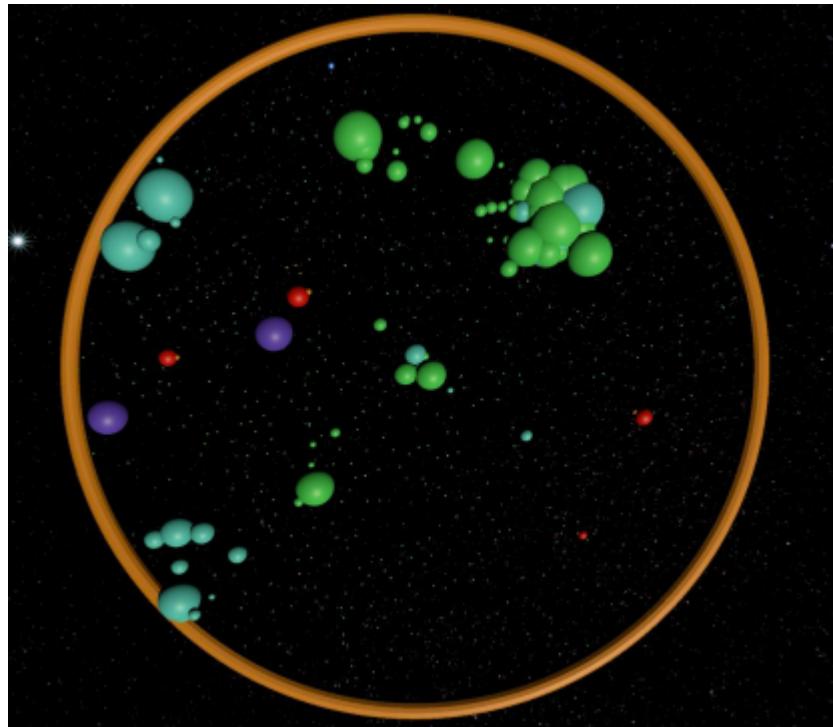
Yanuar Sano Nur Rasyid 13521110

**TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2023**

## BAB I

### 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.



Pada tugas besar pertama Strategi Algoritma ini, gunakanlah sebuah game engine yang mengimplementasikan permainan Galaxio. Game engine dapat diperoleh pada laman berikut: <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 laman berikut ini: <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 disebarluaskan pada peta dengan ukuran 3 dan dapat dikonsumsi 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 IF2211 Strategi Algoritma - Tugas Besar 1 3 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 menembakkannya 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 dikonsumsi 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. Berikut jenis-jenis dari command yang ada dalam permainan:
  - a. FORWARD
  - b. STOP
  - c. START\_AFTERRUNNER
  - d. STOP\_AFTERRUNNER
  - e. FIRE\_TORPEDOES
  - f. FIRE\_SUPERNOVA
  - g. DETONATE\_SUPERNOVA
  - h. FIRE\_TELEPORTER
  - i. TELEPORT
  - j. USE\_SHIELD
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 laman :  
<https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/game-engine/game-rules.md>

## BAB II

### LANDASAN TEORI

#### 2.1.Greedy Algorithm

Algoritma greedy adalah sebuah algoritma yang umum digunakan untuk menyelesaikan persoalan optimasi. Optimasi ini bisa dilakukan dengan maksimasi atau minimasi. Algoritma greedy menyelesaikan persoalan dengan langkah per langkah. Pada setiap langkahnya, algoritma greedy mengambil pilihan yang terbaik dengan harapan pilihan optimum lokal tersebut akan berakhir dengan optimum global. Perlu diperhatikan bahwa algoritma greedy tidak selalu menghasilkan solusi paling optimal, tetapi mendekati solusi yang optimal.

Algoritma greedy memiliki elemen-elemen sebagai berikut:

- a. Himpunan kandidat, C: kandidat yang akan dipilih pada setiap langkah
- b. Himpunan solusi, S: kandidat yang sudah dipilih
- c. Fungsi solusi: memeriksa apakah solusi sudah tercapai dari himpunan kandidat
- d. Fungsi seleksi: memilih kandidat berdasarkan strategi greedy
- e. Fungsi kelayakan: memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi
- f. Fungsi obyektif: memaksimumkan atau meminimumkan

#### 2.2.Galaxio

- a. Cara Kerja Bot Galaxio

Ketika bot dijalankan, pertama-tama *bot* akan mengakses data gameState. Data-data tersebut berupa,

- gameObjects: Menyimpan data-data objek dari gim yang berisi data *size*, *currentHeading*, dan *position*. GameObjectType terdiri dari
  - Food
  - Wormhole
  - Gas Cloud
  - Astroid Field
  - Torpedo Salvo
  - Superfood
  - Supernova Pickup
  - Supernova Bomb
  - Teleporter
  - Shield
- playerGameObjects: Menyimpan data player, dengan setiap player memiliki atribut
  - size: Radius kapal player
  - speed: Kecepatan kapal player

- currentHeading: Arah gerak
  - position: Posisi player dalam
  - gameObjectType: Tipe objek(player)
  - effect: Efek yang dialami oleh bot, disimpan secara *bitwise*.
    - 0 = Tidak ada effect
    - 1 = Afterburner nyala
    - 2 = Asteriod Field
    - 4 = Gas cloud
  - torpedoSalvoCount: Banyak torpedo yang tersedia pada player.
  - supernovaAvailable: Memeriksa apakah player mempunyai supernova.
  - teleporterCount: Banyak teleporter yang tersedia pada player.
  - shieldCount: Banyak shield yang tersedia pada player.
  - teleporterAngle: Menyimpan sudut teleporter yang dilempar.
- world: Menyimpan data radius dari zona serta tick permainan.

Data-data tersebut kemudian dapat digunakan untuk mengkalkulasikan aksi yang akan diambil oleh bot. Aksi-aksi yang dapat dilakukan berupa

- FORWARD: Maju dengan arah *heading* tertentu
- STOP
- START\_AFTERBURNER : Menyalakan afterburner yang mengkalikan kecepatan dengan 2. Mereduksi ukuran player setiap *tick* sebanyak 1.
- STOP\_AFTERBURNER: Mematikan afterburner
- FIRE\_TORPEDOES: Mengonsumsi 1 *salvo charge* dari player dan mengurangi ukuran player sebesar 5 untuk menembakkan suatu torpedo pada *heading* tertentu.
- FIRE\_SUPERNOVA: Menembakkan supernova. Hanya bisa dilakukan jika supernova telah diambil oleh player.
- DETONATE\_SUPERNOVA: Meledakkan supernova yang ditembakkan.
- FIRE\_TELEPORTER: Menembakkan teleporter
- TELEPORT: Melakukan lompat ke teleporter yang ditembakkan

Aksi ini kemudian disimpan bersama *heading* aksi di dalam sebuah objek PlayerAction. PlayerAction kemudian disimpan di dalam prosedur computeNextPlayerAction untuk mengeksekusi aksi tersebut.

Setelah bot disusun menggunakan algoritma greedy bot dikompilasi menggunakan maven. Pada folder tempat disusunnya bot, jalankan command di terminal “mvn clean package”. Perintah tersebut akan menghasilkan sebuah jar file.

#### b. Cara Menjalankan Galaxio

Untuk menjalankan Galaxio, unduh starter-pack yang bisa di unduh dari laman github Galaxio. Extract zip starter-pack. Lakukan konfigurasi jumlah bot yang diingan pada file JSON “appsettings.json” dalam folder

“runner-publish” dan “engine-publish”. Buka terminal baru dan jalankan runner dengan perintah “dotnet GameRunner.dll”. Buka terminal baru kedua dan jalankan perintah “dotnet Engine.dll”. Buka terminal ketiga dan jalankan perintah “dotnet logger.dll”. Kemudian, jalankan seluruh bot yang ingin dipertandingkan. Hasil permainan akan disimpan di file JSON “GameStateLog\_{Timestamp}” dalam folder “logger-publish”. Pada windows, perintah-perintah tersebut bisa dilakukan dengan membuat batch script seperti berikut.

```
@echo off
:: Game Runner
cd ./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

:: Bots
cd ../reference-bot-publish/
timeout /t 3
start "" dotnet ReferenceBot.dll
timeout /t 3
start "" dotnet ReferenceBot.dll
timeout /t 3
start "" dotnet ReferenceBot.dll
timeout /t 3
start java -jar java.jar
cd ../

pause
```

Untuk Unix-based OS, hal yang sama dapat dilakukan dengan menjalankan dan memodifikasi “run.sh”. Untuk melihat visualisasi pertandingan, dapat dengan mengekstrak salah satu zip dengan format “Galaxio-{OS}” di folder visualiser. Jalankan aplikasi galaxio di dalam folder yang sudah di ekstrak. Buka menu “Options”, salin path folder “logger-publish”, pada “Log.Files Location”, dan tekan

tombol save. Buka menu load dan pilih file JSON yang ingin di visualisasikan. Tekan tombol start untuk memulai visualisasi pertandingan.

### 2.3.Implementasi Algoritma *Greedy* ke dalam Bot Galaxio

Algoritma *greedy* diterapkan dalam bot Galaxio dengan mengkalkulasikan gerakan terbaik berdasarkan *GameState* pada waktu tersebut. Gerakan terbaik pada waktu tersebut dilakukan dengan harapan keputusan optimum tersebut akan memberikan keputusan paling optimum. Namun, perlu diingat bahwa batasan algoritma greedy dalam mengambil sebesar-besarnya pada setiap langkah dapat menghalangi algoritma untuk mendapatkan optimum global. Jika bot menggunakan algoritma greedy, *bot* tidak bisa memprediksi keputusan musuh atau mengulang ulang keputusan-keputusan musuh. Meskipun itu, pembuatan algoritma greedy lebih sederhana dibandingkan pendekatan-pendekatan tersebut. Selain itu, dengan algoritma yang sederhana, diharapkan keputusan-keputusan optimum pada waktu tertentu dapat membawa *bot* ke hasil yang terbaik, yaitu *bot* berhasil menjadi kapal terakhir yang masih hidup.

## **BAB III**

### **APLIKASI STRATEGI GREEDY**

#### **3.1. *Mapping persoalan Galaxio menjadi elemen-elemen anggota greedy***

- a. Himpunan kandidat, himpunan PlayerAction yang berisikan PlayerActions dan heading dari PlayerActions tersebut. Aksi PlayerActions terdiri dari FORWARD, STOP, STARTAFTERBURNER, STOPAFTERBURNER, FIRETORPEDOES, FIRESUPERNOVA, DETONATESUPERNOVA, FIRETELEPORT, TELEPORT, ACTIVATESHIELD. Heading berupa derajat dari 0 sampai 359.
- b. Himpunan solusi, PlayerAction yang terpilih.
- c. Fungsi solusi, memeriksa apakah solusi yang dipilih adalah solusi yang paling optimal pada saat itu.
- d. Fungsi seleksi, memilih solusi yang paling optimal pada saat itu.
- e. Fungsi kelayakan, memeriksa apakah solusi yang dipilih tidak terlalu berlebihan.
- f. Fungsi obyektif, solusi yang dipilih adalah solusi yang paling optimal pada saat itu.

#### **3.2. *Eksplorasi alternatif solusi greedy***

##### **a. Solusi 1**

Membagi himpunan kandidat kedalam beberapa kategori seperti *defense*, *retreat*, *offense*, dan *farming*. Kemudian menggunakan fungsi seleksi, fungsi kelayakan, dan fungsi obyektif untuk memilih solusi yang paling tepat. Pemilihan solusi yang tepat bisa dilakukan dengan mencari jarak antara *bot* dengan objektifnya seperti *defense*, *retreat*, *offense*, dan *farming* yang akan dimodifikasi dengan skala prioritas tertentu (dicari nilai yang paling optimal dari percobaan-percobaan). Untuk *defense*, *bot* akan mencari jarak tegak lurus antara dirinya sendiri dengan lintasan torpedo. Untuk *retreat*, *bot* akan mencari jarak antara dirinya dengan ancaman-ancaman berbahaya seperti *gas cloud*, pemain yang berukuran lebih besar, lintasan *supernova*, dan *border* yang dapat menyakiti *bot*. Untuk *offense*, *bot* akan membandingkan jarak dengan musuh yang lain, membandingkan *size bot* dengan *size* musuh, memeriksa apakah memiliki *supernova*. Untuk *farm*, *bot* akan memeriksa *food*, *superfood*, atau *supernova* terdekat dan menghitung waktu yang dibutuhkan untuk sampai ke objek tersebut. Kemudian hasil perhitungan tersebut akan di-filter oleh fungsi seleksi, fungsi kelayakan, dan fungsi obyektif agar menghasilkan himpunan solusi yang terbaik pada saat itu. Karena perhitungan dilakukan dengan menghitung jarak *bot* relatif terhadap sesuatu, maka akan dicari solusi dimana hasil perhitungan tersebut adalah yang terkecil.

##### **b. Solusi 2**

Mengaplikasikan solusi *greedy* dalam berbagai *state* seperti *early-game*, *mid-game*, dan *late-game*. Setiap *state* akan berisi kategori-kategori solusinya

sendiri. Untuk *early-game*, *bot* akan fokus pada *farming* makanan dan menghindari musuh agar bisa memperbesar ukurannya. Untuk *mid-game*, *bot* akan berfokus pada menyerang musuh terbesar dengan menggunakan torpedo atau dengan mengejar musuh-musuh kecil dan memakannya. Untuk *late-game*, *bot* akan berfokus pada menyerang musuh-musuh besar menggunakan weapon yang dimilikinya seperti torpedo dan *supernova*. Untuk setiap *state* akan ada pembagian kedalam beberapa kategori seperti solusi 1, namun dengan skala yang berbeda tergantung dengan *state*-nya sekarang. Contohnya, jika sekarang adalah *state early-game* maka skala untuk *farm* akan jauh lebih kecil dibanding skala untuk *offense*. Kemudian sama seperti solusi 1, akan dipilih solusi yang terbaik untuk saat itu.

### 3.3. Analisis efisiensi dan efektivitas

Pada solusi 1, kita harus melakukan banyak sekali perumpamaan dan percobaan agar kita bisa mendapat skala prioritas yang paling optimal. Namun jumlah perumpamaan dan percobaan yang harus kita lakukan untuk solusi 1 tentunya lebih sedikit dibanding untuk solusi 2.

Jika dianalisa dalam segi efektivitas, sebenarnya kita tidak perlu membagi permainan tersebut kedalam berbagai *state* yang memiliki skala prioritasnya sendiri. Umumnya, masih terdapat banyak makanan yang terletak dekat dengan *bot* pada *state early-game*. Setelah banyak pemain yang memakan makanan-makanan yang tersedia, biasanya pemain-pemain tersebut akan fokus untuk menyerang dan memakan pemain lain karena sudah tidak banyak makanan yang berada di sekitar pemain tersebut. Untuk *state late-game* sendiri biasanya hanya tersisa segelintir pemain, sehingga *bot* dapat fokus untuk menyerang musuhnya.

Oleh karena itu, akan lebih efisien dan efektif jika kita menentukan skala prioritas untuk solusi 1 dibanding solusi 2. Karena sebenarnya prioritas objektif yang harus dilakukan *bot* sama dari *state early-game* sampai *late-game*.

### 3.4. Strategi *greedy* yang dipilih

Berdasarkan hasil analisa efisiensi dan efektivitas dari alternatif-alternatif solusi yang tersedia, maka akan lebih optimal jika solusi 1 direalisasikan sebagai algoritma *greedy* yang akan digunakan *bot*.

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1. Implementasi

Implementasi algoritma *greedy* pada program terdapat pada file BotService.java, dalam metode computeNextPlayerAction. Untuk mempermudah proses perealisasian dan *debugging* kami membuat beberapa kelas baru dengan struktur datanya tersendiri.

- a. Implementasi pada computeNextPlayerAction

```
Procedure computeNextPlayerAction (input/output:  
PlayerAction playerAction)

KAMUS LOKAL
decision = Decisionmaker

ALGORITMA FUNGSI
if (!gameState.getGameObjects().isEmpty()) tgen
    decision <- new Decisionmaker()
    playerAction <- decisio.whatShouldBotDo()
    printPlayerAction()
setPlayerAction(playerAction)
```

- b. Implementasi pada *Decisionmaker*

```
Procedure findPriority()

KAMUS LOKAL
double temp_prio;
defense_prio: Defense
retreat_prio: Retreat
farm_prio: Farm
offence_prio: Offence

ALGORITMA FUNGSI
temp_prio = 0;
farm_prio.getFarmPrio(player, gameState)
if (temp_prio > farm_prio.prio) then
    decision_kind = 3
    temp_prio = farm_prio.prio
defense_prio.getDefensePrio()
if (temp_prio > defense_prio.prio) then
    decision_kind = 3;
    temp_prio = defense_prio.prio
retreat_prio.getRetreatPrio()
if (temp_prio > retreat_prio.prio) then
    decision_kind = 2
```

```

        decision_kind_variation = retreat_prio.kind
        temp_prio = retreat_prio.prio
        offence_prio.getPriority()

    if (temp_prio > offence_prio.priority) then
        decision_kind = 4
        temp_prio = offence_prio.priority

Function whatBotShouldDo() -> PlayerAction

KAMUS LOKAL
command: PlayerAction
defense_prio: Defense
retreat_prio: Retreat
farm_prio: Farm
offence_prio: Offence
decision_kind: int
decision_kind_variation: int

ALGORITMA
findPriority()
command <- PlayerAction()
if (decision_kind=1) then
    -> defense_prio.actionDefense()
else if (decision_kind == 2) then
    -> retreat_prio.actionRetreat()
else if (decision_kind == 3) then
    -> farm_prio.normalFarm(command, player,
gameState)
else if (decision_kind == 4) then
    -> offence_prio.doOffence()
else if (decision_kind == 5) then
    -> teleporter_prio.doTeleport(gameState, player)
-> command

```

c. Implementasi pada *General*

```

Function
distanceFromPlayerToProjectileTrajectory(input
GameObject projectile, input GameObject player) ->
Double

{Fungsi yang menghitung jarak player ke suatu
projectile}

KAMUS LOKAL

jari_jari = integer

```

```

projectile_position, player_position, PQ = Position
nx, ny, temp, distance = double
heading, object_speed = integer

ALGORITMA

jari_jari <- player.getSize()/2 + 1 { +1 karena
pembagian di java dibulatkan kebawah}

projectile_position <- projectile.getPosition();

player_position <- player.getPosition()

heading <- projectile.currentHeading % 360

object_speed <- projectile.getSpeed()

if (heading >= 270)

    heading <- heading % 270

    nx <- (object_speed *
Math.sin(Math.toRadians(heading)) )

    ny <- (object_speed *
Math.cos(Math.toRadians(heading)) )

else if (heading >= 180)

    heading <- heading % 180

    nx <- (object_speed *
Math.cos(Math.toRadians(heading)) )

    ny <- (object_speed *
Math.sin(Math.toRadians(heading)) )

else if (heading >= 90) {

    heading <- heading % 90

    nx <- (object_speed *
Math.sin(Math.toRadians(heading)) )

    ny <- (object_speed *

```

```

Math.cos(Math.toRadians(heading)) )

else

    nx <- (object_speed *
Math.cos(Math.toRadians(heading)) )

    ny <- (object_speed *
Math.sin(Math.toRadians(heading)) )

double temp = nx

nx = ny

ny = (-1 * temp)

PQ = new Position()

PQ.setX(player_position.getX() - projectile_position.
getX())

PQ.setY(player_position.getY() - projectile_position.
getY())

distance <- Math.abs((PQ.getX() * nx +
PQ.getY() * ny) / (Math.sqrt(nx * nx + ny * ny)))

distance <- distance - Double.valueOf(jari_jari)

-> distance

Function distanceFromPlayerToObject (input
GameObject object1, input GameObject player) ->
Double

{Fungsi yang menghitung jarak player ke object}

```

#### **KAMUS LOKAL**

triangleX, triangleY = integer

#### **ALGORITMA**

```

triangleX <- Math.abs(object1.getPosition().x -
player.getPosition().x)

triangleY <- Math.abs(object1.getPosition().y -
player.getPosition().y)

```

```
-> Math.sqrt(triangleX * triangleX + triangleY * triangleY)
```

```
Function distanceFromPlayerToPlayer (input GameObject player1,input GameObject player2) -> Double
```

```
{Fungsi yang menghitung jarak player ke player}
```

#### **KAMUS LOKAL**

```
triangleX, triangleY = integer
```

#### **ALGORITMA**

```
triangleX <- Math.abs(player1.getPosition().x - player2.getPosition().x)
```

```
triangleY <- Math.abs(player1.getPosition().y - player2.getPosition().y)
```

```
-> Math.sqrt(triangleX * triangleX + triangleY * triangleY) - player1.getSize() - player2.getSize()
```

```
Function distanceFromPlayerToLocation(input Position x, input GameObject player) -> Double
```

```
{Fungsi yang menghitung jarak player ke Location}
```

#### **KAMUS LOKAL**

```
triangleX, triangleY = integer
```

#### **ALGORITMA**

```
triangleX <- Math.abs(x.x - player.getPosition().x)
```

```
triangleY <- Math.abs(x.y - player.getPosition().y)
```

```
-> Math.sqrt(triangleX * triangleX + triangleY * triangleY)
```

```
Function radToDegree(input double v) -> integer
```

```
{Fungsi yang mengkonversi radian ke derajat}
```

#### **KAMUS LOKAL**

```

PI = double

ALGORITMA

-> (integer) (v * (180 / Math.PI))

Function objectHeading(input GameObject object1,
input GameObject object2)-> integer

{Fungsi yang menghitung sudut object2 untuk menuju
object1}

KAMUS LOKAL

Object2_position = Position

direction = integer

ALGORITMA

object2_position <- object2.getPosition()

direction <- radToDegree(Math.atan2(y -
object2_position.getY(), x -
object2_position.getX()))

-> (direction + 360) % 360

Function objectHeadingToPoint(input int x, input
int y, input GameObject object2) -> integer

{Fungsi yang menghitung derajat untuk player menuju
point tertentu}

KAMUS LOKAL

Object2_position = Position

direction = integer

ALGORITMA

object2_position <- object2.getPosition();

direction <- radToDegree(Math.atan2(y -
object2_position.getY(), x -
object2_position.getX()));

```

```

-> (direction + 360) % 360;

Function distanceFrom(input GameObject object,
input ObjectTypes type, input GameState
gameState) -> List<GameObject>

{ membuat list gameobject tersusun terdekat dari
suatu object }

KAMUS LOKAL

listObject, orderedList = List of GameObject

ALGORITMA

if (type == ObjectTypes.PLAYER)

    listObject <- gameState.getPlayerGameObjects()

else

    listObject <- gameState.getGameObjects()

orderedList <- listObject

.stream().filter(item -> item.getGameObjectType()
== type)

.sorted(Comparator

.comparing(item ->
General.distanceFromPlayerToObject(object, item)))

.collect(Collectors.toList())

-> orderedList

Function getObjectListDistance(input ObjectTypes
type, input GameState gameState, input GameObject
bot) -> List<GameObject>

{ membuat list gameobject tersusun dari yang
terdekat terhadap player}

KAMUS LOKAL

listObject, orderedList = List of GameObject

```

### **ALGORITMA**

```
if (type == ObjectTypes.PLAYER)

    listObject <- gameState.getPlayerGameObjects()

    orderedList <-
getObjectTypeDistance(gameState, bot)

else

    listObject <- gameState.getGameObjects()

    orderedList <- listObject

        .stream().filter(item ->
item.getGameObjectType() == type)

        .sorted(Comparator

            .comparing(item ->
General.distanceFromPlayerToObject(bot, item) ))

        .collect(Collectors.toList())

-> orderedList;

Function getObjectPlayerDistance(input GameState
gameState, input GameObject bot) ->
List<GameObject>

{membuat list gameobject tersusun dari yang
terdekat terhadap player}
```

### **KAMUS LOKAL**

```
listObject, orderedList = List of GameObject
```

### **ALGORITMA**

```
listObject <- gameState.getPlayerGameObjects()

orderedList <- listObject

    .stream().filter(item ->
item.getGameObjectType() == ObjectTypes.PLAYER)
```

```

        .sorted(Comparator
            .comparing(item ->
General.distanceFromPlayerToObject(bot,
item)-bot.getSize()-item.getSize()))

        .collect(Collectors.toList())
-> orderedList

Function getObjectListSize(input ObjectTypes type,
input GameState gameState, input GameObject bot)
-> List<GameObject>

{ membuat list gameobject tersusun dari yang
terkecil}

KAMUS LOKAL

listObject, orderedList = List of GameObject

ALGORITMA

if (type == ObjectTypes.PLAYER)

    listObject <- gameState.getPlayerGameObjects()

else

    listObject <- gameState.getGameObjects()

orderedList = listObject

    .stream().filter(item ->
item.getGameObjectType() == type)

    .sorted(Comparator
        .comparing(item -> item.getSize()))

    .collect(Collectors.toList())
-> orderedList

Function tickFromDistance(input GameObject
projectile, input GameObject player) -> integer

```

```
{ menghitung tick yang dibutuhkan projectile untuk  
menuju player}
```

#### **KAMUS LOKAL**

#### **ALGORITMA**

```
-> (int)  
General.distanceFromPlayerToObject(projectile,  
player) / projectile.getSpeed()
```

```
Function isItHeadingTowards (input GameObject  
object1, input GameObject object2) -> Boolean
```

```
{Fungsi yang menentukan apakah suatu object1 menuju  
object2}
```

#### **KAMUS LOKAL**

```
Heading, x, y, x2, y2 = integer
```

#### **ALGORITMA**

```
heading <- object1.currentHeading % 360;
```

```
x <- object2.getPosition().getX();
```

```
y <- object2.getPosition().getY();
```

```
x2 <- object2.getPosition().getX();
```

```
y2 <- object2.getPosition().getY();
```

```
if (heading >= 270)
```

```
    if (x2 >= x & y2 <= y)
```

```
        -> true
```

```
    else
```

```
        -> false
```

```
else if (heading >= 180)
```

```
    if (x2 <= x & y2 <= y)
```

```

        -> true

    else

        -> false;

else if (heading >= 90)

    if (x2 <= x & y2 >= y)

        -> true

    else

        -> false

else

    if (x2 >= x & y2 >= y)

        -> true

    else

        -> false

```

d. Implementasi pada *Defense*

```

Procedure getPriority()

{Procedure yang menentukan kind dari fungsi
getPriority}

{mendeteksi jika bot dekat dengan lintasan torpedo,
regardless jumlah torpedo yang mendekatinya}

{ jika ada satu saja torpedo yang mendekati bot,
maka bot akan mengeluarkan shield }

KAMUS LOKAL

min = double

torpedoList = List<GameObject>

ALGORITMA

```

```

{ melihat }

torpedoList <- gameState.getGameObjects()

    .stream().filter(item ->
item.getGameObjectType() ==
ObjectTypes.TORPEDOSALVO &
General.isItHeadingTowards(item, bot))

    .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToProjectileTrajectory(it
em, bot)))

    .collect(Collectors.toList());

if (torpedoList.isEmpty()) {

    min <- 3000000{asumsi diurutkan
membesar}

else

    min <-
General.distanceFromPlayerToProjectileTrajectory(to
rpedoList.get(0), bot);

this.prio <- min

Function actionDefense(input GameObject object1,
input GameObject object2) -> PlayerAction

{Fungsi yang mengembalikan aksi yang dilakukan dari
prio}

KAMUS LOKAL

playerAction = PlayerAction

torpedoList = List<GameObject>

ALGORITMA

playerAction <- new PlayerAction();

if (bot.getShieldCount() != 0)

    playerAction.action =

```

```

PlayerActions.ACTIVATESHIELD;

    playerAction.heading = 0;

else
    torpedoList = gameState.getGameObjects()
        .stream().filter(item ->
    item.getGameObjectType() ==
ObjectTypes.TORPEDOSALVO &
General.isItHeadingTowards(item, bot))
        .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToProjectileTrajectory(item, bot)))
        .collect(Collectors.toList())

    playerAction.action <- PlayerActions.FORWARD;

    playerAction.heading
    (General.objectHeading(torpedoList.get(0), bot) +
180) % 360;

-> playerAction

```

e. Implementasi pada *Retreat*

```

Procedure getPrioType()

{ priority untuk kabur dari player lebih penting
daripada menghindari gas cloud }

{untuk gas cloud}

KAMUS LOKAL

gas_prio, run_prio, supernova_prio, border_prio =
double

gasList, playerlist, supernovalist =
List<GameObject>

ALGORITMA

gas_prio <- 1000000;

gasList <- gameState.getGameObjects()

```

```

        .stream().filter(item ->
item.getGameObjectType() == ObjectTypes.GASCLOUD)

        .sorted(Comparator.comparing(item -> General.
distanceFromPlayerToPlayer(item, bot)))

        .collect(Collectors.toList())

if (!gasList.isEmpty())

    gas_prio <-
General.distanceFromPlayerToObject(gasList.get(0),
bot) * 3{ skalanya dapat berubah}

{untuk kabur dari player}

run_prio <- 1000000;

playerlist <- gameState.getGameObjects()

        .stream().filter(item ->
item.getGameObjectType() == ObjectTypes.PLAYER)

        .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToPlayer (item, bot)))

        .collect(Collectors.toList())

if (!playerlist.isEmpty())

    run_prio <-
General.distanceFromPlayerToObject(playerlist.get(1),
bot)

{untuk kabur dari supernova}

supernova_prio <- 1000000

supernovalist <- gameState.getGameObjects()

        .stream().filter(item ->
item.getGameObjectType() ==
ObjectTypes.SUPERNOVABOMB)

        .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToProjectileTrajectory(it

```

```

em, bot) )

.collect(Collectors.toList())

if (!supernovalist.isEmpty()) {

    supernova_prio <-
General.distanceFromPlayerToProjectileTrajectory(su
pernovalist.get(0), bot)

    {untuk kabur dari border}

if (gameState.getWorld().getRadius() != null) {

    border_prio <-
gameState.getWorld().getRadius() *
gameState.getWorld().getRadius()

    border_prio ==
(bot.getPosition().getX()*bot.getPosition().getX()
+
bot.getPosition().getY()*bot.getPosition().getY());

else

    border_prio <- 1000000

if (run_prio <= gas_prio & run_prio <=
supernova_prio & run_prio <= border_prio)

    this.prio = run_prio

    this.kind = 2

else if (gas_prio <= run_prio & gas_prio <=
supernova_prio & gas_prio <= border_prio)

    this.prio = gas_prio

    this.kind = 1

else if (supernova_prio <= run_prio &
supernova_prio <= gas_prio & supernova_prio <=
border_prio)

    this.prio = supernova_prio

    this.kind = 3

```

```

else

    this.prio = border_prio;

    this.kind = 4

Function actionRetreat () -> PlayerAction

{Fungsi yang mengembalikan aksi yang dilakukan dari
prio}

KAMUS LOKAL

playerAction = PlayerAction

gasList, PlayerList, supernovalist=
List<GameObject>

ALGORITMA

playerAction <- new PlayerAction()

if (kind == 1) {lari dari gas}

    gasList <- gameState.getGameObjects()

        .stream().filter(item ->
item.getGameObjectType() == ObjectTypes.GASCLOUD)

        .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToPlayer (item, bot)))

        .collect(Collectors.toList())

    playerAction.action <- PlayerActions.FORWARD;

    playerAction.heading <-
(General.objectHeading(gasList.get(0), bot) + 180)
% 360

else if (kind == 2) { lari dari player lain}

    var playerlist = gameState.getGameObjects()

        .stream().filter(item ->
item.getGameObjectType() == ObjectTypes.PLAYER)

```

```

        .sorted(Comparator.comparing(item -> General.
distanceFromPlayerToPlayer (item, bot)))

        .collect(Collectors.toList())

        playerAction.action <- PlayerActions.FORWARD;

        playerAction.heading <-
(General.objectHeading(playerlist.get(1), bot) +
180) % 360

else if (kind == 3) { lari dari supernova}
{menjauh dari tempat supernova
sekarang}

supernovalist <- gameState.getGameObjects()

        .stream().filter(item ->
item.getGameObjectType() ==
ObjectTypes.SUPERNOVABOMB)

        .sorted(Comparator.comparing(item ->
General.distanceFromPlayerToProjectileTrajectory(it
em, bot)))

        .collect(Collectors.toList())

        playerAction.action <- PlayerActions.FORWARD;

        playerAction.heading <-
(General.objectHeading(supernovalist.get(0), bot) +
180) % 360

else

{menjauh dari border menuju pusat map}

playerAction.action <- PlayerActions.FORWARD;

        playerAction.heading <-
(General.objectHeadingtoPoint(0, 0, bot)) % 360

-> playerAction

```

#### f. Implementasi pada *Offense*

```

Procedure getPrioOffence()

{ Procedure yang menghitung priority offence dan
menyimpannya di prio_gen}

KAMUS LOKAL

playerList = List of GameObject

player, bot = GameObject

distance, prio_gen = double

isSupernovaing = boolean

ALGORITMA

playerList <-
General.getObjectListDistance(ObjectTypes.PLAYER,
gameState, bot)

playerList.remove(bot)

if (playerList.size() == 0)

    prio_gen <- 1000

    ->

    if (isSupernovaing)

        prio_gen <- 0;

        ->

player = playerList[0]

double distance =
General.distanceFromPlayerToObject(bot, player)

prio_gen = distance

Function doOffence() -> PlayerAction

{ Fungsi yang mengembalikan aksi terhadap kind yang

```

```
didapatkan}
```

#### **KAMUS LOKAL**

```
playerList = List of GameObject
```

```
player, bot = GameObject
```

```
distance, prio_gen = double
```

```
isSupernovaing = Boolean
```

```
kind = integer
```

#### **ALGORITMA**

```
enemy <- ObjectTypes.PLAYER
```

```
action <- PlayerActions.FIRETORPEDOES
```

```
if (kind == 1) then
```

```
    action = PlayerActions.FIRETORPEDOES
```

```
else if (kind == 2) then
```

```
    if (!isSupernovaing) then
```

```
        isSupernovaing = true
```

```
->
```

```
basicAttackSize(PlayerActions.FIRESUPERNOVA, enemy,  
true)
```

```
else
```

```
    isSupernovaing = false
```

```
-> detonateSupernova()
```

```
else if (kind == 3) then
```

```
    action = PlayerActions.FIRETELEPORT
```

```
-> basicAttackSize(action, enemy, false)
```

```

Procedure getPrioType()

{Procedure yang menentukan kind dari fungsi
getPrio}

KAMUS LOKAL

prio_tor, prio_sup, prio_eat, kind = double

ALGORITMA

prio_tor <- getPrioTorpedoes()

prio_sup <- getPrioSupernova()

prio_eat <- getPrioEat()

if (prio_tor > prio_sup and prio_tor > prio_eat)
then

    kind = 1

else if (prio_sup > prio_tor and prio_sup >
prio_eat) then

    kind = 2

else if (prio_eat > prio_tor and prio_eat >
prio_sup) then

    kind = 3

else

    kind = 0

Function getPrioTorpedoes () -> double

{Fungsi yang mengembalikan nilai prio dari
penyerangan torpedo}

KAMUS LOKAL

playerListD , playerLists = List of GameObject

max_distance, min_size_attac = intenger

```

```

prio = double

player, bot = GameObject

ALGORITMA

prio <- 0;

max_distance <- 50;

min_size_attack <- 25;

playerListD <-
General.getObjectListDistance(ObjectTypes.PLAYER,
gameState, bot)

playerLists <-
General.getObjectListSize(ObjectTypes.PLAYER,
gameState, bot)

playerListD.remove(bot)

playerListS.remove(bot)

if (playerListD.size() == 0)

    -> 0

if (bot.getSize() < min_size)

    return 0

{apabila player tidak jauh}

if (General.distanceFromPlayerToObject(bot,
playerListD.get(0)) <= max_distance)

    prio += 5

{apabila player lebih besar dari size bot}

if (playerLists.get(0).getSize() > min_size_attack)

    prio += 5

-> prio

```

```

Function getPrioSupernova () -> double
{mendapatkan nilai prioritas untuk melakukan aksi
supernova}

KAMUS LOKAL

mult = intenger

prio = double

ALGORITMA

prio <- 0

mult <- 2

if (bot.supernovaAvailable == 0)

    -> 0

prio += 10;

if (checkNearSupernova())

    prio += 100

-> prio * mult;

Function getPrioEat () -> double

KAMUS LOKAL

playerLists = List of GameObject

min_size, min_distance, mult = intenger

prio = double

player, bot = GameObject

ALGORITMA

var playerLists <-
General.getObjectListSize(ObjectTypes.PLAYER,
gameState, bot);

```

```

playerListS.remove(bot)

if (playerListS.size() == 0)

    -> 0

{apabila player lebih kecil dari size bot}

if (playerLists.get(0).getSize() < min_size)

    prio += 5

if (General.distanceFromPlayerToObject(bot,
playerListS.get(0)) <= min_distance)

    prio += 5

reverse(playerListS);

{apabila player terbesar lebih kecil dari size bot}

if (playerListS.get(0).getSize() < min_size)

    prio += 5

-> prio * mult

Function defaultAction () -> PlayerAction

```

{Mengembalikan playeraction default yaitu STOP ke head makanan atau 0}

#### **KAMUS LOKAL**

```

foodList = List of GameObject

bot = GameObject

gameState = GameState

head = integer

playerAction = PlayerAction

command = PlayerActions

```

#### **ALGORITMA**

```

playerAction <- new PlayerAction()

command <- PlayerActions.STOP

head <- 0

foodList <-
General.getObjectListDistance(ObjectTypes.FOOD,
gameState, bot)

if (!foodList.isEmpty())

    head <-
General.objectHeading(foodList.get(0), bot)

    command <- PlayerActions.FORWARD

playerAction.action <- command

playerAction.heading <- head

println("default offence")

-> playerAction

Function basicAttackDistance (input PlayerActions
command, input ObjectTypes object, input boolean
desc) -> PlayerAction

{melakukan command dengan heading objek dengan
jarak tertentu dari player. Jika true jarak
terdekat false jarak terjauh}

```

#### **KAMUS LOKAL**

```

objectList = List of GameObject

bot = GameObject

gameState = GameState

playerAction = PlayerAction

```

#### **ALGORITMA**

```

playerAction <- new PlayerAction()

```

```

playerAction.setAction(command)

objectList <-
General.getObjectListDistance(object, gameState,
bot)

if(object == ObjectTypes.PLAYER)

    objectList.remove(bot)

if (desc)

    reverse(objectList)

if (!objectList.isEmpty())

playerAction.setHeading(General.objectHeading(objec
tList.get(0), bot))

else

-> defaultAction()

-> playerAction

Function basicAttackSize (input PlayerActions
command, input ObjectTypes object, input boolean
desc) -> PlayerAction

{melakukan command dengan heading objek dengan size
tertentu dari player. Jika true size terkecil false
size terbesar }

```

#### **KAMUS LOKAL**

```

objectList = List of GameObject

bot = GameObject

gameState = GameState

playerAction = PlayerAction

```

#### **ALGORITMA**

```

playerAction <- new PlayerAction()

```

```

playerAction.setAction(command)

objectList <- General.getObjectListSize(object,
gameState, bot)

if (object == ObjectTypes.PLAYER)

    objectList.remove(bot)

if (desc)

    reverse(objectList)

if (!objectList.isEmpty())
playerAction.setHeading(General.objectHeading(objectList.get(0), bot))

else

    -> defaultAction()

-> playerAction

Function checkNearSupernova () -> Boolean

{Fungsi mengembalikan true apabila ada player yang
dekat dengan supernova}

```

#### **KAMUS LOKAL**

```

playerList, supernovaBombList = List of GameObject

bot = GameObject

gameState = GameState

```

#### **ALGORITMA**

```

playerList <-
General.getObjectListDistance(ObjectTypes.PLAYER,
gameState, bot)

playerList.remove(bot)

supernovaBombList <-
General.getObjectListDistance(ObjectTypes.SUPERNOVA
BOMB, gameState, bot)

```

```

if (!playerList.isEmpty() and
    !supernovaBombList.isEmpty())

    ->(General.distanceFromPlayerToObject(playerList.get(0), supernovaBombList.get(0)) <= supernovaRadius)

else

    -> false

Function detonateSupernova () -> PlayerAction

{Mengembalikan PlayerAction yang meledakan
supernova}

KAMUS LOKAL

playerAction = PlayerAction

command = PlayerActions

head = integer

ALGORITMA

playerAction <- new PlayerAction()

command <- PlayerActions.DETONATESUPERNOVA

head <- 0

playerAction.setAction(command)

playerAction.setHeading(head)

-> playerAction

```

g. Implementasi pada *Farm*

```

Procedure getFarmPrio(input GameObject player,
input GameState gamestate)
KAMUS LOKAL
prio: double
ALGORITMA

```

```

prior<-200+gameState.getWorld().getCurrentTick()*1.1

Function normalFarm(PlayerAction command1,
GameObject player, GameState gameState)->
PlayerAction

KAMUS LOKAL
foodList: array of GameObject
command: PlayerAction

ALGORITMA
PlayerAction command = new PlayerAction()

foodList <- gameState.getGameObjects()
.stream()
.filter(item -> (item.getGameObjectType() ==
ObjectTypes.FOOD ||
item.getGameObjectType() == ObjectTypes.SUPERFOOD ||
item.getGameObjectType() == ObjectTypes.SUPERNOVAPICK
UP))
.sorted(Comparator
.comparing(item ->
General.distanceFromPlayerToObject(item, player)))
.collect(Collectors.toList())

command.heading
=General.objectHeading(foodList.get(0), player)

command.setAction(PlayerActions.FORWARD)
-> command

```

## 4.2. Penjelasan struktur data

Struktur data pada implementasi algoritma *greedy* berbentuk *class* dalam bahasa pemrograman Java. Struktur program yang kami buat memiliki empat kategori yaitu *Enum* yang berisi *constant* mengenai aksi dan objek dalam permainan, *Models* yang mengatur kelas yang berisi data-data saat permainan berjalan seperti radius dari *world* dan posisi dari sebuah objek, *Decision* yang berisi kelas yang mengatur aksi apa yang akan bot lakukan pada keadaan tertentu dengan algoritma *greedy*. Berikut adalah penjelasan lebih lanjut dari struktur data pada kelas yang kami gunakan.

### 4.2.1. Kategori Enums

Kelas disini berisi konstanta dari aksi dan objek-objek yang akan digunakan dalam permainan sehingga kelas tersebut hanya berisi konstanta dengan sedikit atau tidak ada *method* dalam kelas.

- i. *ObjectTypes.java*

Kelas yang berisi nilai dari objek dalam *game*.

Attributes	Penjelasan
PLAYER(1)	objek PLAYER memiliki value 1
FOOD(2)	objek FOOD atau makanan memiliki value 2
WORMHOLE(3)	objek WORMHOLE memiliki value 3
GASCLOUD(4)	objek GASCLOUD memiliki value 4
TORPEDOSALVO(5)	objek TORPEDOSALVO memiliki value 5
SUPERFOOD(6)	objek SUPERFOOD memiliki value 6
SUPERNOVAPICKUP(7)	objek SUPERNOVAPICKUP atau senjata supernova, memiliki value 7
SUPERNOVABOMB(8)	objek SUPERNOVABOMB atau proyektil dari senjata supernova memiliki value 8
TELEPORTER(9)	objek TELEPORTER memiliki value 9
SHIELD(10)	objek SHIELD memiliki value 10
public final integer value	nilai integer

Selain itu, terdapat satu method dalam kelas ini yaitu public static ObjectTypes valueOf(Integer value) yang berfungsi untuk mengembalikan objek dari value yang dimasukkan.

#### ii. *PlayerActions.java*

Kelas yang berisi perintah-perintah yang dapat dilakukan dalam game.

Attributes	Penjelasan
FORWARD(1)	perintah FORWARD memiliki value 1
STOP(2)	perintah STOP memiliki value 2

STARTAFTERBURNER(3)	perintah STARTAFTERBURNER memiliki value 3
STOPAFTERBURNER(4)	perintah STOPAFTERBURNER memiliki value 4
FIRETORPEDOES(5)	perintah FIRETORPEDOES memiliki value 5
FIRESUPERNOVA(6)	perintah FIRESUPERNOVA memiliki value 6
DETONATESUPERNOVA(7)	objek DETONATESUPERNOVA atau senjata supernova, memiliki value 7
FIRETELEPORT(8)	perintah FIRETELEPORT atau proyektil dari senjata supernova memiliki value 8
TELEPORT(9)	perintah TELEPORT memiliki value 9
ACTIVATESHIELD(10)	perintah ACTIVATESHIELD memiliki value 10
public final integer value	nilai integer

#### 4.2.2. Kategori Models

Kategori ini berisi

i. *GameObject.java*

Kelas ini berisi mengenai atribut-atribut yang digunakan pada saat permainan berlangsung dan selalu di-update ketika bermain.

Attributes	Penjelasan
public UUID id	id unik dari setiap GameObject dalam permainan
public Integer size	besar dari sebuah objek yang dalam integer
public Integer speed	kecepatan bergerak dari sebuah objek dalam integer
public Integer currentHeading	sudut arah bergerak dari sebuah objek dalam integer
public Position position	posisi dari sebuah objek bertipe Position

public ObjectTypes gameObjectType	tipe sebuah objek dari enums ObjectTypes
public Integer effect	efek yang dapat dimiliki sebuah player yaitu afterburner, asteroid, dan gas cloud
public Integer torpedoSalvoCount	Jumlah torpedosalvo yang dimiliki oleh player dalam integer
public Integer supernovaAvailable	Jumlah supernova yang dimiliki oleh player dalam integer
public Integer teleporterCount	Jumlah teleporter yang dimiliki oleh player dalam integer
public Integer shieldCount	Jumlah shield yang dimiliki oleh player dalam integer
public Integer teleporterAngle	Sudut dari teleporter dalam integer

*Method* yang terdapat pada kelas ini merupakan *method getter* atau pengambilan nilai dan *setter* atau penyimpanan nilai dari setiap atribut-atribut yang sudah dijelaskan di atas.

#### ii. GameState.java

Attributes	Penjelasan
public World world	objek dari Kelas World
public List<GameObject> gameObjects	List dari objek-objek yang terdapat dalam permainan
public List<GameObject> playerGameObjects	List dari player yang berada dalam permainan

*Method* yang terdapat pada kelas ini, sama seperti sebelumnya, terdiri dari *getter* dan *setter*. Selain itu, terdapat *method class creator* yaitu :

Method	Penjelasan
public GameState()	Membuat GameState baru yang terdiri dari atribut yang sudah dijelaskan di atas
public GameState(World world, List<GameObject> gameObjects, List<GameObject>	Membuat GameState dengan keadaan atribut sesuai dengan parameter yang dimasukkan

playerGameObjects)	
--------------------	--

iii. *GameStateDto.java*

Attributes	Penjelasan
private World world	objek dari Kelas World
private Map<String, List<Integer>>	Map String aksi dengan nilai integer heading dari objek pada game
private Map<String, List<Integer>>	Map String aksi dengan nilai integer heading dari player pada game

*Method* yang terdapat pada kelas ini merupakan *method getter* atau pengambilan nilai dan *setter* atau penyimpanan nilai dari setiap atribut-atribut yang sudah dijelaskan di atas.

iv. *PlayerAction.java*

Attributes	Penjelasan
public UUID playerId	id unik dari sebuah pemain
public PlayerActions action	Enum PlayerActions yang melambangkan aksi sebuah bot
public int heading	Besaran sudut dari aksi yang akan dilakukan

*Method* yang terdapat pada kelas ini merupakan *method getter* atau pengambilan nilai dan *setter* atau penyimpanan nilai dari setiap atribut-atribut yang sudah dijelaskan di atas.

v. *Position.java*

Attributes	Penjelasan
public int x	Koordinat pada sumbu x
public int y	Koordinat pada sumbu y

*Method* yang terdapat pada kelas ini hanya terdiri dari *getter* atau *setter*.

vi. *World.java*

Attributes	Penjelasan
Position centerPoint	Titik tengah dari world
Integer radius	Radius batas world
Integer currentTick	Banyaknya gerakan/tick yang telah terjadi

Method yang terdapat pada kelas ini hanya terdiri dari *getter* atau *setter*.

#### 4.2.3. Kategori Decision

Terdapat enam file dari kategori *Class Decision* yaitu *Decisionmaker* yang berisi susunan prioritas algoritma, *General* yang berisi fungsi-fungsi umum yang digunakan untuk *class* lain, *Defense* yang berisi fungsi dan prosedur yang mengatur aksi-aksi defensif untuk *bot*, *Retreat* yang mengatur aksi-aksi yang akan dilakukan untuk situasi mundur, *Offense* berisi aksi-aksi untuk menyerang lawan, dan *Farm* yang berisi perintah kepada *bot* untuk memakan *food* yang tersedia pada *world*. Penjelasan lebih lanjut dari struktur data dari *Class* yang kami gunakan adalah sebagai berikut.

##### i. *Decisionmaker.java*

Attributes	Penjelasan
public int decision_kind	Kategori keputusan yang akan dilakukan oleh <i>bot</i> .
public int decision_kind_variation	Variasi dari kategori keputusan yang akan dilakukan oleh <i>bot</i> .
public Defense defense_prio	Kelas Defense yang digunakan untuk menghitung prioritas defensif
public Retreat retreat_prio	Kelas Retreat yang digunakan untuk menghitung prioritas menghindari musuh atau objek
public Farm farm_prio	Kelas Farm yang digunakan untuk menghitung prioritas mencari makan
public Offence offence_prio	Kelas Offence yang digunakan untuk menentukan prioritas untuk menyerang musuh
private GameObject player	GameObject Player yang melambangkan bot yang sedang

	digunakan.
private GameState gameState	Status-status dari game yang sedang berjalan.

Methods	Penjelasan
public PlayerAction whatBotShouldDo()	Menghasilkan aksi yang tepat dari keadaan bot
public void findPriority()	Membandingkan prioritas antar keputusan, kemudian memilih keputusan dengan prioritas paling kecil.

ii. *General.java*

*Class Defense* berisi *method* dan *attributes* umum yang berfungsi untuk pemakaian oleh kelas lain

Methods	Penjelasan
double distanceFromPlayerToProjectileTrajectory(GameObject projectile, GameObject player)	Menghitung jarak seorang player ke projectile dalam tipe double
double distanceFromPlayerToObject(GameObject object1, GameObject player)	Menghitung jarak player dengan objek.
double distanceFromPlayerToPlayer(GameObject player1, GameObject player2)	Menghitung jarak dari player1 dan player2.
double distanceFromPlayerToLocation(Position x, GameObject player)	Menghitung jarak player dari koordinat x.
int radToDegree(double v)	Mengkonversi radian menjadi derajat.
int objectHeading(GameObject object1, GameObject object2)	Mencari derajat heading object2 agar bisa menuju object1.

int objectHeadingToPoint(int x, int y, GameObject object2)	Mencari derajat heading object2 agar bisa menuju koordinat x.
List<GameObject> distanceFrom(GameObject object, ObjectTypes type, GameState gameState)	Membuat list gameobject tersusun terdekat dari suatu object.
List<GameObject> getObjectListDistance(ObjectTypes type, GameState gameState, GameObject bot)	Membuat list gameobject dengan jenis gameobject tertentu dan tersusun dari yang terdekat terhadap player.
List<GameObject> getObjectPlayerDistance(GameState gameState, GameObject bot)	Membuat list gameobject tersusun dari yang terdekat terhadap player.
List<GameObject> getObjectListSize(ObjectTypes type, GameState gameState, GameObject bot)	Membuat list gameobject tersusun dari ukuran yang terkecil terhadap player.
int tickFromDistance(GameObject projectile, GameObject player)	Menghitung tick yang dibutuhkan projectile untuk menuju player.
boolean isItHeadingTowards(GameObject object1, GameObject object2)	Mengembalikan true jika objek1 bergerak ke objek2.

### iii. *Defense.java*

*Class Defense* berisi *method* dan *attributes* yang berfungsi untuk memberikan *PlayerAction* yang berkaitan dengan penyerangan terhadap lawan.

Attributes	Penjelasan
double prio	Nilai prioritas keputusan Defense.
GameObject bot	GameObject Player yang melambangkan bot yang sedang digunakan.
GameState gameState	Status-status dari game yang sedang berjalan.

Methods	Penjelasan
---------	------------

void getDefensePrio()	Prosedur yang mendapatkan nilai prioritas defensif dan menyimpannya di atribut prio
PlayerAction actionDefense()	Fungsi yang mengembalikan aksi defensif yang tepat dari suatu keadaan

iv. *Retreat.java*

*Class Retreat* berisi *method* dan *attributes* yang berfungsi untuk memberikan *PlayerAction* yang berkaitan dengan menghindari lawan.

Attributes	Penjelasan
public double prio	Nilai prioritas untuk aksi mundur
public int kind	Nilai jenis dari aksi mundur
private GameObject bot	GameObject Player dari bot yang sedang diatur
private GameState gameState	<i>State</i> dari game yang sedang berjalan

Methods	Penjelasan
void getRetreatPrio()	Menghitung prioritas untuk melarikan diri dari bahaya atau musuh.
PlayerAction actionRetreat()	Fungsi mengembalikan PlayerAction yang dapat digunakan untuk melarikan diri dari bahaya atau musuh.

v. *Offence.java*

*Class Offence* berisi *method* dan *attributes* yang berfungsi untuk memberikan *PlayerAction* yang berkaitan dengan penyerangan terhadap lawan.

Attributes	Penjelasan
private int supernovaSize	Ukuran supernova.
private double supernovaRadius	Radius dari ledakan supernova

private int min_size	Minimal size bot untuk melakukan aksi menembak torpedo
public double prio_gen	Nilai prioritas umum dari aksi menyerang
private double kind	Nilai tipe dari aksi menyerang
private static boolean isSupernovaing	Status penembakan supernova
private GameObject bot	GameObject Player yang melambangkan bot yang sedang digunakan.
private GameState gameState;	Status-status dari game yang sedang berjalan.

Methods	Penjelasan
public void getPrioOffence()	Prosedur yang menghitung nilai prioritas umum untuk menyerang
public PlayerAction doOffence()	Fungsi yang memberikan aksi menyerang yang tepat dari prioritas yang ada
public void getPrioType()	Mendapatkan tipe aksi offence yang akan dilakukan
public double getPrioTorpedoes()	Mendapatkan nilai prioritas aksi torpedo
public double getPrioSupernova()	Mendapatkan nilai prioritas aksi supernova
public double getPrioEat()	Mendapatkan nilai prioritas aksi memakan
public PlayerAction defaultAction()	Fungsi yang melakukan aksi default yaitu STOP atau menuju makanan terdekat
public PlayerAction basicAttackDistance(PlayerActions command, ObjectTypes object, boolean desc)	Melakukan sebuah aksi terhadap objek yang terdekat dari bot atau terjauh
public PlayerAction basicAttackSize(PlayerActions	Melakukan sebuah aksi terhadap objek yang terkecil dari bot atau

command, ObjectTypes object, boolean desc)	terbesar
public PlayerAction fireTeleport(GameObject player)	Menembakan teleport ke arah player
public PlayerAction playerTeleport()	Mengembalikan PlayerAction untuk melakukan teleport.
public boolean isItTheCorrectTeleporter(GameObject teleporter)	Memeriksa apakah teleporter sudah tepat dengan bot
public boolean isItInTeleporterRadius(GameObject teleporter, GameObject player)	Memeriksa apakah player berada di radius teleporter
public boolean checkNearSupernova()	Memeriksa apakah ada player yang dekat supernova
public PlayerAction detonateSupernova()	Mengembalikan PlayerAction untuk meledakkan supernova jika sebuah player memasuki radius supernova.
public PlayerAction chasePlayer(GameState gameState)	Mengembalikan PlayerAction untuk mengejar musuh yang lebih kecil.

#### vi. *Farm.java*

*Class Farm* berisi *method* dan *attributes* yang berfungsi untuk memberikan *PlayerAction* yang berkaitan dengan mencari makanan.

Attributes	Penjelasan
double prio	Nilai prioritas keputusan Farm.

Methods	Penjelasan
void getFarmPrio(GameObject player, GameState gameState)	Menghitung prioritas farming
PlayerAction normalFarm(PlayerAction command1, GameObject player, GameState gameState)	Mengembalikan playeraction untuk mencari <i>food</i> , <i>superfood</i> , atau <i>supernova</i> .

#### 4.2.4. Kategori Services

i. *BotService.java*

Sebuah kelas yang berfungsi sebagai penggerak bot.

Attributes	Penjelasan
private GameObject bot	GameObject dari bot yang kita gunakan
private PlayerAction playerAction	Perintah aksi dan heading dalam kelas PlayerAction
private GameState gameState	Status-status dari game yang sedang berjalan

Methods	Penjelasan
public void computeNextPlayerAction(PlayerAction playerAction)	Metode yang akan menentukan aksi yang dilakukan oleh bot. Aksi tersebut berasal dari perhitungan Kelas Decisionmaker
private void printPlayerAction()	Metode yang akan menampilkan aksi dari player ke std output

Selain kelas-kelas di atas, terdapat Main.java sebagai pemulai dan penjalan semua kelas-kelas yang sudah dijelaskan.

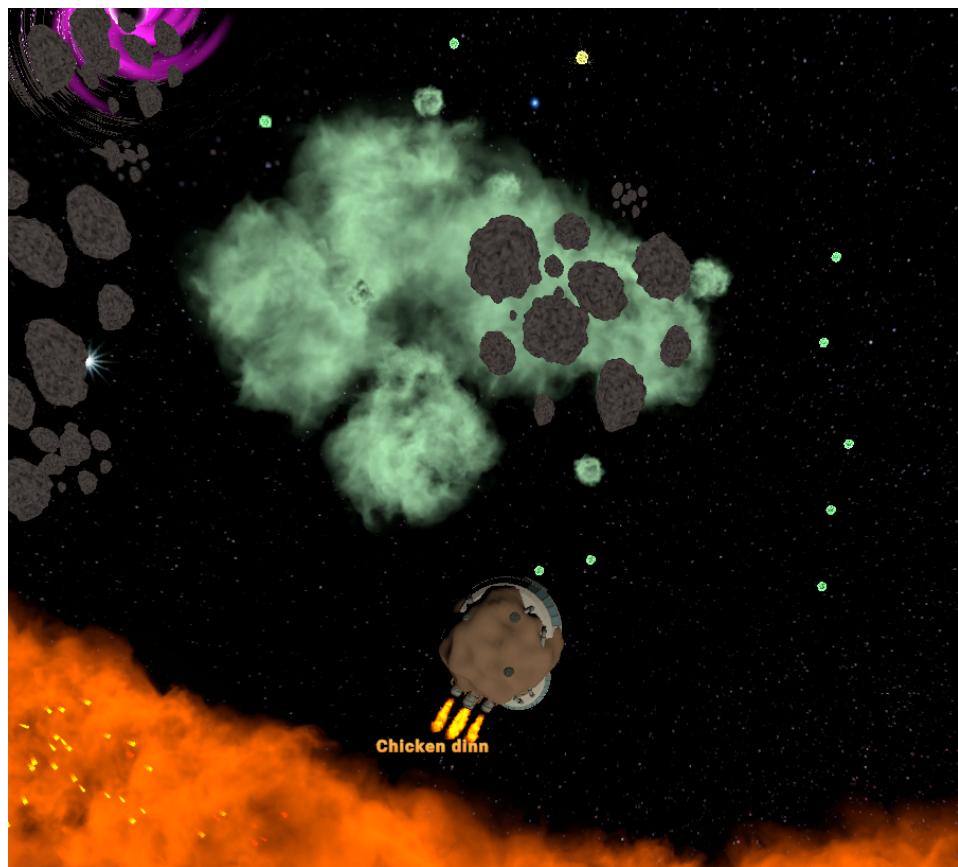
#### 4.3. Analisis kasus

a. Analisis kasus *Defense*



Pada kasus tersebut bisa dilihat bahwa *bot* langsung mengaktifkan *shield* ketika dia berada dekat dengan lintasan torpedo tersebut. Sehingga pada kasus tersebut, *bot* berpikir bahwa menggunakan *shield* lebih diprioritaskan dari prioritas lainnya.

b. Analisis kasus *Retreat*



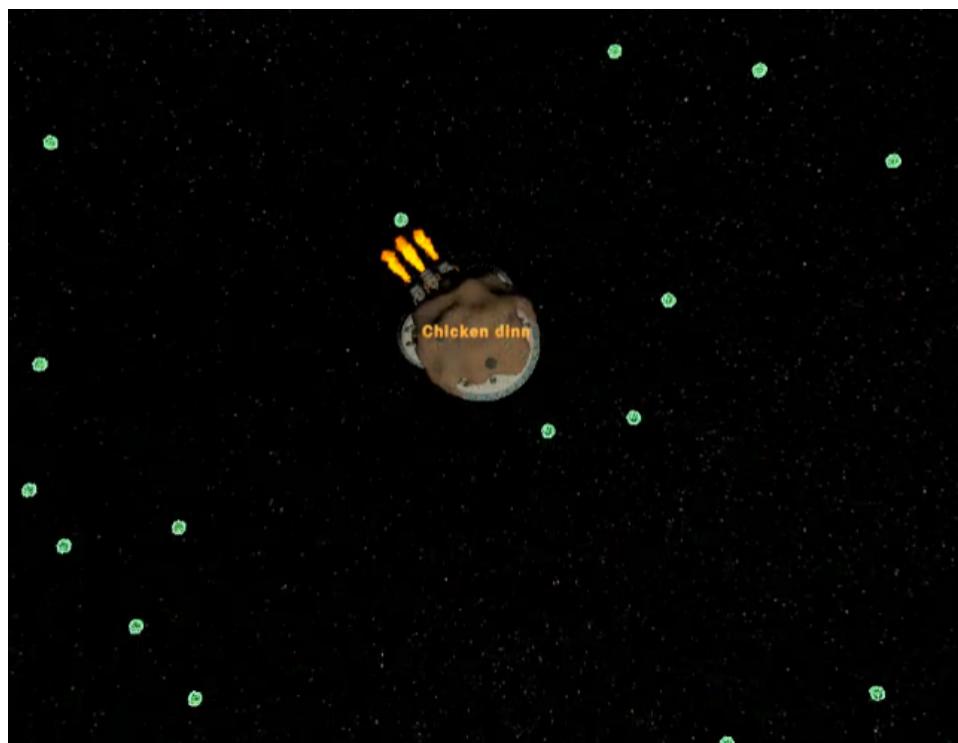
Pada kasus tersebut, *bot* terjebak dikarenakan implementasi aksi yang harus dilakukan jika ia bertemu dengan *gas cloud* serta *border* dari game.

c. Analisis kasus *Offense*



Pada kasus tersebut, *bot* terdesak diantara kabur dari border dan menyerang *player* lain. Berdasarkan prioritasnya, *bot* akan mengutamakan menyerang *player* lain terlebih dahulu baru kabur.

d. Analisis kasus *Farm*



Pada kasus tersebut, *bot* berada diantara 2 buah *food* yang berjarak sama sehingga *bot* akan kebingungan saat memilih aksi selanjutnya.

## **BAB V**

### **KESIMPULAN, SARAN, KOMENTAR, DAN REFLEKSI**

#### **5.1. Kesimpulan**

Algoritma *greedy* bisa kita aplikasikan ke dalam sebuah permainan. Walaupun algoritma *greedy* tidak mengkalkulasikan efek dari tindakannya sekarang, algoritma *greedy* dapat memberikan solusi yang secara objektif paling tepat pada situasi tersebut.

#### **5.2. Saran**

Jika ingin menggabungkan elemen-elemen seperti fungsi solusi, seleksi, kelayakan dan kandidat, akan lebih baik jika memikirkan cara penggabungan fungsi-fungsi tersebut sehingga dapat mempermudah proses perealisasian dan proses *debugging*.

#### **5.3. Refleksi**

Akan lebih baik jika memikirkan elemen-elemen pada algoritma *greedy* saat mengimplementasikan algoritma *greedy* pada *bot*. Merealisasikan algoritma yang sesuai dengan elemen-elemen tersebut dapat mempermudah waktu *debugging*.

## **DAFTAR PUSTAKA**

edunex.itb.ac.id (Diakses pada tanggal 12 Februari 2023)

Munir, Rinaldi. 2021. “Algoritma Greedy (Bagian 1)”.

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf), diakses 12 Februari 2023.

Munir, Rinaldi. 2021. “Algoritma Greedy (Bagian 2)”.

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf), diakses 12 Februari 2023.

Munir, Rinaldi. 2021. “Algoritma Greedy (Bagian 3)”.

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag3.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag3.pdf), diakses 12 Februari 2023.

<https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/game-engine/game-rules.md#teleport>, diakses 12 Februari 2023.

## **LAMPIRAN**

Link github : [https://github.com/yansans/Tubes1\\_Chicken-dinner](https://github.com/yansans/Tubes1_Chicken-dinner)  
Link video demo : <https://youtu.be/hMxsDvE5U1Y>