

Pemanfaatan Algoritma *Greedy* dalam Aplikasi Permainan “Galaxio”

Tugas Besar 1 IF2211 Strategi Algoritma



Disusun Oleh Kelompok 38:

Akbar Maulana Ridho – 13521093

Juan Christopher Santoso – 13521116

Rinaldy Adin – 13521134

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

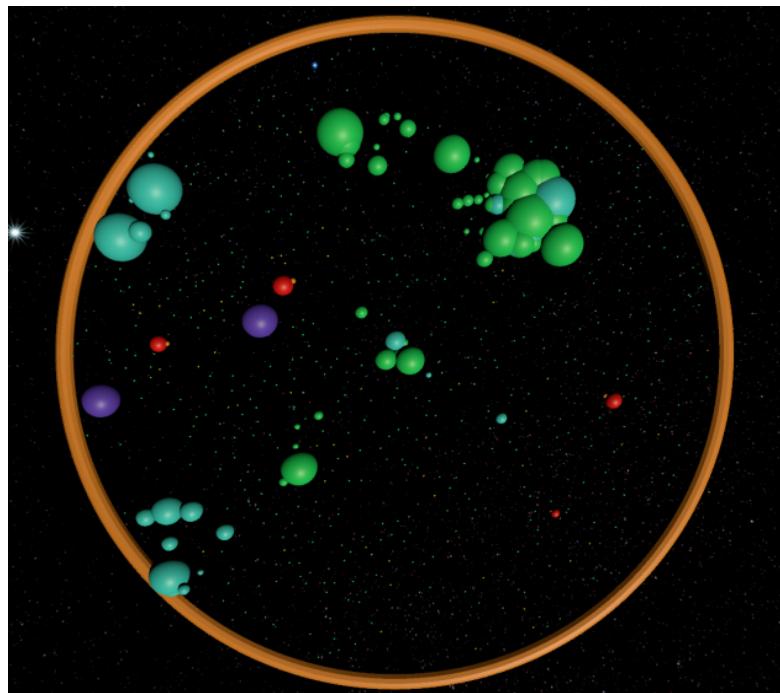
Bab I - Deskripsi Tugas	1
Bab II - Landasan Teori	5
A. Algoritma Greedy	5
B. Game Engine	6
C. Struktur Program	6
D. Alur Jalan Program	7
E. Alur Pengembangan Strategi	8
Bab III - Aplikasi Strategi Greedy	12
A. Attacker	12
B. Defender	13
C. Escape Gas	14
D. Evade Collision	16
E. Explorer	17
F. Shield	18
G. Sniper	19
H. To Center	20
Bab IV - Implementasi dan Pengujian	22
A. Implementasi dalam Pseudocode	22
1. Implementasi Aksi yang Dapat Dilakukan oleh Bot	22
2. Implementasi Strategi Algoritma Greedy pada Bot	27
3. Implementasi Perhitungan	36
B. Struktur Data Program	38
1. Actions	39
2. Agents	41

3. Enums	43
4. Models	44
5. Services	44
6. Utils	44
7. Main	44
C. Analisis dan Pengujian	44
1. Attacker	45
2. Defender	46
3. Escape Gas	47
4. Evade Collision	48
5. Explorer	49
6. Shield	50
7. Sniper	51
8. To Center	52
Bab V - Kesimpulan dan Saran	53
A. Kesimpulan	53
B. Saran	53
Daftar Pustaka	54
Lampiran	55
A. Repository Github	55
B. Link Video YouTube	55

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. Agar dapat memenangkan pertandingan, setiap bot harus mengimplementasikan strategi tertentu untuk dapat memenangkan permainan. Dalam tugas ini, digunakan sebuah game engine yang mengimplementasikan permainan Galaxio. Bot kapal dalam permainan Galaxio akan mengimplementasikan strategi algoritma Greedy untuk memenangkan permainan. Strategi greedy yang diimplementasikan harus dikaitkan dengan fungsi objektif dari permainan itu sendiri, yaitu memenangkan permainan dengan cara mempertahankan kapal pemain paling terakhir untuk hidup.



Gambar 1.1 Ilustrasi permainan Galaxio

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 salvo 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. 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 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
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_AFTERTURBURNER
 - d. STOP_AFTERTURBURNER
 - e. FIRE_TORPEDOES
 - f. FIRE_SUPERNOVA
 - g. DETONATE_SUPERNOVA

- h. FIRE_TELEPORTER
 - i. TELEPORTUSE_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.

Bab II

Landasan Teori

A. Algoritma Greedy

Algoritma Greedy merupakan metode yang digunakan dalam suatu penyelesaian masalah optimasi yang diselesaikan secara bertahap (step by step), dengan harapan hasil yang didapatkan merupakan hasil terbaik yang dapat diperoleh. Algoritma Greedy melibatkan pencarian sebuah himpunan bagian kandidat yang harus memenuhi beberapa kriteria yang ditentukan yaitu menyatakan suatu solusi dan optimasi dari suatu objek. Tujuan dari Greedy Algorithm adalah mencari solusi yang mendekati nilai optimal dengan memaksimumkan atau meminimumkan hasil penyelesaian yang didapat.

Algoritma Greedy memiliki sebuah prinsip yaitu “take what you can get now!” yang artinya ambil apa yang bisa kamu dapatkan sekarang. Algoritma Greedy memiliki beberapa komponen dalam proses penyelesaian masalah yaitu

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

Pemilihan Algoritma Greedy dikarenakan metode ini dapat digunakan untuk mengambil keputusan berdasarkan nilai-nilai yang bervariasi. Hal ini karena terdapat banyak kondisi permainan yang dapat digunakan untuk menentukan langkah terbaik sehingga perlu diimplementasi solusi greedy untuk memilih kondisi permainan paling optimal dalam penentuan langkah.

B. Game Engine

Game Engine memiliki pengertian sebagai sistem perangkat lunak yang dapat digunakan untuk mengembangkan dan menciptakan sebuah game. *Game engine* itu sendiri pada dasarnya adalah sebuah library yang dapat digunakan untuk membuat game. Game engine akan memberikan kemudahan bagi pengembang permainan karena menyediakan fungsi-fungsi inti dari sebuah permainan. Game Engine Dapat digunakan untuk membuat lebih dari satu permainan, dan pengembang permainan dapat mengoptimalkan proses pengembangan dengan cara menggunakan atau mengadaptasi game engine yang telah ada sebelumnya.

C. Struktur Program

Dalam program Galaxio yang menjadi subjek Tugas Besar kali ini, terdapat beberapa komponen yang disertai dalam starter-pack yang telah disediakan. Komponen yang diperlukan agar permainan dapat berjalan adalah sebagai berikut

1. Game engine: Game engine bertanggung jawab dalam menjalankan permainan sesuai aturan-aturan yang sudah ditetapkan. Game engine bekerja dengan menjalankan command bot yang diterima.
2. Game runner: Game runner akan menjalankan permainan diantara semua bot yang terhubung. Runner akan menerima dan mengirim command dari bot untuk dieksekusi oleh Game Engine.
3. Game logger: Game logger akan mencatat semua perubahan dan error yang terjadi dalam permainan serta menghasilkan file log pada akhir permainan.
4. Ec-compose: Ec-compose berisi pengaturan docker untuk menjalankan game engine, runner, dan logger di dalam container docker.
5. Starter-bots: Starter bots berisi program bot yang dapat digunakan sebagai *boilerplate* dalam mengembangkan bot pemain.
6. Visualizer: Visualizer berisi program executable yang dapat memvisualisasikan permainan yang sudah dijalankan. Visualizer akan melihat file log

matchState.json dan matchState_gameComplete.json untuk ditampilkan permainannya.

Pada folder starter-bots, terdapat contoh-contoh bot yang dapat digunakan untuk mengembangkan bot dalam berbagai bahasa pemrograman. Dalam folder bot berbahasa Java, terdapat folder source code, folder target, Dockerfile, dan file pom.xml. Dockerfile berguna untuk menjalankan bot dalam container docker. Pom.xml berguna untuk menyimpan konfigurasi dan dependency build yang digunakan saat melakukan build menggunakan maven. Folder target diantaranya berisi file .jar hasil kompilasi. Di dalam folder source code, disediakan

- File Main.java: berisi logika utama program bot, yang berfungsi untuk menerima data state permainan dari runner, serta memanggil dan mengeksekusi logika tingkah laku bot.
- Folder Enums: berisi enumerator yang menyatakan jenis-jenis objek dan jenis-jenis aksi yang dapat dilakukan pemain.
- Folder Models: berisi kelas berupa model/struktur data yang ada dalam permainan, seperti GameObject, GameState, dan lain-lain.
- Folder Services: berisi logika yang memperhitungkan gerakan/aksi bot berdasarkan data yang diterima oleh program bot.

D. Alur Jalan Program

Untuk menjalankan program permainan serta bot yang ingin dijalankan, diperlukan beberapa tools yang perlu diinstall, yaitu Java Development Kit untuk menjalankan bot, .Net Core untuk menjalankan program runner, engine, dan logger, dan Maven untuk build program bot. Jika semua tools sudah diinstall, program dapat dijalankan dengan menjalankan game-runner, game-engine, dan game-logger dengan menjalankan command dotnet <namaprogram>.dll atau dengan menggabungkan semua command tersebut ke dalam batch file agar dapat dijalankan sekaligus.

Ketika semua program permainan sudah jalan, permainan akan menunggu bot untuk terhubung ke dalam program. Bot dapat dijalankan dengan menjalankan command java -jar <path> dengan path berupa alamat file .jar program yang sudah di-build dan compile

menggunakan maven, dengan command mvn clean package atau dengan menggunakan IDE.

Setelah semua bot terhubung, permainan akan mulai dan akan berjalan hingga hanya tersisa satu bot. Setelah program berakhir, semua komponen program akan berhenti dan file log matchState.json dan matchState_gameComplete.json akan dihasilkan oleh program. Kedua file log tersebut dapat digunakan untuk melihat visualisasi dari game yang telah berjalan. Visualizer yang terdapat di dalam zip perlu di extract terlebih dahulu. Jika sudah di extract, program visualizer dapat dijalankan dan menampilkan permainan.

E. Alur Pengembangan Strategi

Starter pack menyediakan starter bot sebagai titik awal dari pengembangan bot. Starter bot sudah dibekali kode yang dibutuhkan oleh bot untuk melakukan koneksi dengan runner sehingga kita dapat fokus kepada strategi permainan. Untuk memulai, silahkan salin folder JavaBot (atau tidak usah, terserah kalian) dan beri nama bot kalian dengan cara mengubah string “Coffee Bot” pada file Main.java.

Karena *boilerplate* program bot berbahasa Java tidak *update*, maka perlu ditambahkan beberapa enum object dan player actions. Oleh karena itu, file enums ObjectTypes.java dan PlayerActions.java adalah sebagai berikut,

ObjectTypes.java

```
public enum ObjectTypes {
    PLAYER(1),
    FOOD(2),
    WORMHOLE(3),
    GAS_CLOUD(4),
    ASTEROID_FIELD(5),
    TORPEDO_SLAVO(6),
    SUPER_FOOD(7),
    SUPERNOVA_PICKUP(8),
    SUPERNOVA_BOMB(9),
    TELEPORTER(10),
    SHIELD(11);

    public final Integer value;
```

```

ObjectTypes(Integer value) {
    this.value = value;
}

public static ObjectTypes valueOf(Integer value) {
    for (ObjectTypes objectType : ObjectTypes.values()) {
        if (Objects.equals(objectType.value, value)) return
objectType;
    }

    throw new IllegalArgumentException("Value not found");
}
}

```

PlayerActions.java

```

public enum PlayerActions {
    FORWARD(1),
    STOP(2),
    STARTAFTERBURNER(3),
    STOPAFTERBURNER(4),
    FIRETORPEDOES(5),
    FIRE_SUPERNOVA(6),
    DETONATE_SUPERNOVA(7),
    FILE_TELEPORT(8),
    TELEPORT(9),
    ACTIVATESHIELD(10);

    public final Integer value;

    PlayerActions(Integer value) {
        this.value = value;
    }
}

```

Selain itu, file GameObjects.java perlu juga ditambah agar dapat menerima informasi tambahan. Oleh karena itu, konstruktor dan fungsi FromStateList pada kelas tersebut adalah sebagai berikut,

GameObject.java

```

public GameObject(
    UUID id,

```

```

        Integer size,
        Integer speed,
        Integer currentHeading,
        Position position,
        ObjectTypes gameObjectType,
        Integer torpedoSalvoCount,
        Integer effects,
        Integer supernovaAvailable,
        Integer shieldCount,
        Integer teleporterCount) {
    this.id = id;
    this.size = size;
    this.speed = speed;
    this.currentHeading = currentHeading;
    this.position = position;
    this.gameObjectType = gameObjectType;
    this.torpedoSalvoCount = torpedoSalvoCount;
    this.effects = effects;
    this.supernovaAvailable = supernovaAvailable;
    this.shieldCount = shieldCount;
    this.teleporterCount = teleporterCount;
}

public static GameObject FromStateList(UUID id,
List<Integer> stateList) {
    Position position = new Position(stateList.get(4),
stateList.get(5));

    if (stateList.size() == 7) {
        return new GameObject(
            id,
            stateList.get(0),
            stateList.get(1),
            stateList.get(2),
            position,
            ObjectTypes.valueOf(stateList.get(3)),
            0,
            stateList.get(6),
            0,
            0,
            0);
    }

    return new GameObject(
        id,
        stateList.get(0),

```

```
        stateList.get(1),  
        stateList.get(2),  
        position,  
        ObjectTypes.valueOf(stateList.get(3)),  
        stateList.get(7),  
        stateList.get(6),  
        stateList.get(8),  
        stateList.get(10),  
        stateList.get(9));  
    }  
}
```

Dengan mempertimbangkan struktur data diatas, maka bot sudah dapat dikembangkan dengan menambahkan fungsi-fungsi dan juga logika pada ServiceBot.java.

Bab III

Aplikasi Strategi *Greedy*

Berdasarkan tujuan dari strategi bot yang dianggap sesuai dengan berbagai situasi yang dapat muncul dalam suatu ronde permainan, kami telah membuat strategi bot dengan tujuan menjelajah, menyerang, dan menghindar/pertahanan. Untuk lebih khususnya, strategi bot yang kami kembangkan adalah sebagai berikut,

A. Attacker

Attacker adalah strategi dengan pendekatan greedy yang memprioritaskan untuk menembakkan torpedo dan bergerak mengikuti target. Syarat agar pendekatan ini diaktifkan adalah terdapat lawan pada radius tertentu atau apabila hanya terdapat satu lawan. Selanjutnya, terdapat tiga kemungkinan aksi dari strategi ini. Bila salvo torpedo telah penuh dan ukuran kapal mencukupi, kapal akan menembak torpedo menuju kapal terdekat. Bila syarat sebelumnya tidak terpenuhi, kapal akan mencoba melakukan penyesuaian arah kapal berdasarkan situasi di sekitar kapal. Bila perlu penyesuaian, kapal akan mengoreksi arah pergerakan kapal. Bila arah pergerakan sudah sesuai, kapal akan menembakkan torpedo kepada lawan.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command forward dan fire torpedo dengan arah gerak bebas
- 2) Himpunan solusi: Command forward dengan arah hasil penyesuaian arah gerak kapal, Command fire torpedo dengan arah yang akan mengenai target
- 3) Fungsi solusi: Memilih arah gerak yang optimal saat menyerang dan torpedo mengenai lawan
- 4) Fungsi seleksi: Memilih arah gerak yang mengikuti lawan dan memiliki makanan yang optimal, tetapi tidak menabrak pada kapal lawan. Menembak kapal target dengan jarak paling dekat
- 5) Fungsi kelayakan: Bila arah gerak kapal saat ini bukanlah arah gerak optimal, pergerakan kapal akan dikoreksi. Sudut tembakan disesuaikan dengan jarak serta kecepatan sudut target relatif terhadap pemain.
- 6) Fungsi objektif: menghancurkan kapal lawan secepat-cepatnya

b. Analisis Efisiensi Solusi

Untuk mencari target kapal terdekat, diperlukan kompleksitas sorting $O(n \log n)$ dengan n adalah banyaknya kapal lawan. Perhitungan sudut tembakan torpedo memerlukan kompleksitas $O(1)$. Terakhir, pencarian arah gerak optimal, yang meliputi pencarian makanan, menggunakan algoritma sorting dengan kompleksitas $O(n \log n)$ dengan n adalah banyaknya kandidat arah gerak objek. Sehingga, kompleksitas algoritma strategi ini adalah $O(n \log n)$.

c. Analisis Efektivitas Solusi

Solusi ini sangat efektif untuk menghancurkan lawan. Dengan menembakkan torpedo, ukuran kapal lawan bisa berkurang secara drastis. Selain itu, dengan menyesuaikan arah gerak kapal dengan kapal lawan, kapal bisa menjaga jarak dan menembak dengan efektif. Meskipun begitu, Torpedo yang ditembakkan seringkali bisa dengan mudah dihindari oleh lawan, sehingga diperlukan uji coba untuk menentukan jarak optimal agar kapal bisa memberikan kerusakan yang optimal.

Meskipun begitu, pendekatan ini tidak begitu efektif untuk melawan kapal berukuran lebih besar yang memakan kapal lain dengan teleporter. Salah satu solusi pencegahannya adalah dengan membuat kapal lebih agresif sehingga mencegah kapal lawan untuk tumbuh jauh lebih besar.

B. Defender

Defender adalah strategi dengan pendekatan greedy yang akan memprioritaskan menghindari torpedo dan supernova yang mengarah ke pemain. Ketika terdeteksi bahwa terdapat torpedo, supernova, atau teleporter yang berpotensi untuk mengenai player, bot akan memilih strategi Defender untuk mengubah arah jalannya agar dapat menghindari serangan tersebut.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command forward dengan arah heading bebas
- 2) Himpunan solusi: Command forward dengan arah heading yang aman dari lintasan proyektil
- 3) Fungsi solusi: Memilih heading yang aman dari lintasan proyektil.
- 4) Fungsi seleksi: Memilih heading tegak lurus dengan lintasan proyektil.

- 5) Fungsi kelayakan: Memeriksa apakah proyektil tersebut berada dalam jarak dekat dan sedang mengarah ke pemain.
- 6) Fungsi objektif: Meminimalisir damage yang dikenai player dari proyektil.

b. Analisis Efisiensi Solusi

Untuk menghitung apakah proyektil akan mengenai player maka diperiksa range sudut dari kedua sisi pemain terhadap posisi proyektil. Jika heading proyektil berada dalam range tersebut serta berada dalam jarak tertentu sehingga proyektil dapat dianggap berbahaya, bot akan bergerak maju dengan arah yang tegak lurus dengan arah proyektil tersebut. Karena solusi Defender perlu memeriksa semua proyektil yang ada, kompleksitas solusi ini adalah sebesar **O(n)** dengan n berupa jumlah proyektil yang ada pada suatu saat.

c. Analisis Efektivitas Solusi

Defender memeriksa heading dan jarak dari proyektil yang berpotensi berbahaya bagi player. Hanya satu proyektil yang terpilih untuk dihindar, yaitu proyektil dengan bahaya terbesar, serta dengan jarak terdekat.

Strategi efektif apabila:

- Proyektil ditembak dari jarak yang jauh sehingga player memiliki waktu untuk menghindar
- Player memiliki ukuran yang cukup kecil agar dapat menghindar dalam waktu cepat

Strategi tidak efektif apabila:

- Proyektil ditembakkan dari jarak dekat sehingga langsung mengenai player.
- Terdapat banyak proyektil yang dari arah berbeda yang dapat mengenai player karena proyektil selain proyektil paling dekat tidak akan diperhitungkan.
- Player memiliki ukuran yang terlalu besar sehingga kecepatan player terlalu pelan untuk menghindar.

C. Escape Gas

Escape adalah strategi dengan pendekatan greedy yang akan memprioritaskan keluar dari dalam gas cloud. Ketika terdeteksi player berada di dalam gas cloud, bot akan memilih strategi Escape Gas untuk mengubah arah jalannya agar dapat keluar dari gas cloud secepat mungkin.

- a. Mapping Elemen Greedy
 - 1) Himpunan kandidat: Command forward dengan arah heading bebas
 - 2) Himpunan solusi: Command forward dengan arah heading yang mengeluarkan pemain dari dalam gas cloud.
 - 3) Fungsi solusi: Memilih heading yang mengeluarkan pemain dari gas cloud.
 - 4) Fungsi seleksi: Memilih heading yang sama dengan heading antara gas cloud dengan player.
 - 5) Fungsi kelayakan: Memeriksa apakah player masih di dalam gas cloud.
 - 6) Fungsi objektif: Meminimalisir damage yang dikenai player dari gas cloud.
- b. Analisis Efisiensi Solusi

Untuk menghitung apakah player sedang berada di dalam gas cloud, program hanya perlu memeriksa atribut effects yang dikirim oleh program permainan. Setelah diketahui bahwa pemain berada di dalam gas cloud, program akan menyortir semua gas cloud yang ada di permainan berdasarkan jarak gas cloud dengan player. Dari gas cloud paling dekat, bot akan mendapatkan heading paling optimal dengan mencari heading antara gas cloud terdekat dengan player. Karena algoritma sortir dalam java menggunakan TimSort, maka dapat disimpulkan bahwa kompleksitas algoritma strategi ini adalah **O(n log n)** dengan n merupakan jumlah gas cloud yang ada.

- c. Analisis Efektivitas Solusi

Escape gas memilih gas cloud yang paling dekat dengan player sehingga hanya satu gas cloud yang dipilih sebagai patokan. Selain itu, program bot hanya akan jalan dengan kecepatan normal karena tidak menggunakan afterburner.

Strategi efektif apabila:

- Player masih memiliki ukuran yang cukup untuk jalan keluar dari gas cloud.
- Gas cloud yang menutup player tidak merangkap dengan gas cloud lain.

Strategi tidak efektif apabila:

- Player terlalu kecil sehingga player tereliminasi sebelum berhasil keluar
- Heading yang dihasilkan strategi malah mengarahkan player ke dalam gas cloud lain.

D. Evade Collision

Evade Collision adalah strategi dengan pendekatan greedy yang akan memprioritaskan menjauh dari lawan yang sedang arah geraknya akan menabrak. Ketika terdeteksi lawan yang lebih besar akan menabrak player, bot akan memilih strategi Evade Collision untuk mengubah arah jalannya agar dapat menghindari lawan tersebut secepat mungkin.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command forward dengan arah heading bebas
- 2) Himpunan solusi: Command forward dengan arah heading yang menghindari player dari lintasan gerak lawan.
- 3) Fungsi solusi: Memilih heading yang menjauhkan pemain dari lintasan lawan.
- 4) Fungsi seleksi: Memilih lawan terdekat dan lebih besar dari player, serta memilih heading tegak lurus yang paling cepat dalam menghindari lawan.
- 5) Fungsi kelayakan: Memeriksa apakah lawan akan mengenai player dalam 2.5 tick ke depan.
- 6) Fungsi objektif: Memaksimalkan jarak antara player dengan lawan yang ingin menabrak player.

b. Analisis Efisiensi Solusi

Untuk menghitung apakah lawan terdekat akan bertabrakan dengan player, program bot akan memastikan apakah jarak real (jarak antara sisi terluar kedua objek) cukup dekat, jika cukup dekat perlu diinterpolasi jarak minimal lawan akan mengenai player dalam lintasan pada saat itu, interpolasi diberi batas agar perhitungan tidak terlalu kompleks/banyak. Setelah diketahui bahwa lawan akan mengenai pemain dalam 2.5 tick, program akan mencari arah tegak lurus dengan heading lawan yang paling optimal. Karena solusi Evade Collision hanya memerlukan perhitungan interpolasi dan perhitungan sederhana lainnya dan hanya memerlukan TimSort untuk menemukan lawan paling dekat, maka dapat disimpulkan bahwa kompleksitas algoritma strategi ini adalah **O(n log n)** dengan n merupakan jumlah lawan yang ada.

c. Analisis Efektivitas Solusi

Evade collision memilih pemain yang paling dekat dengan player sehingga hanya satu lawan yang dihindari.

Strategi efektif apabila:

- Hanya satu lawan yang dihindari dan tidak ada bahaya lain di arah berlawanan.
- Player cukup kecil sehingga dapat menghindar dalam waktu yang cukup.

Strategi tidak efektif apabila:

- Heading yang dihasilkan strategi malah mengarahkan player ke ancaman lain, seperti proyektil, gas cloud, atau lawan lain.

E. Explorer

Explorer adalah strategi dengan pendekatan greedy yang akan memprioritaskan mencari makanan selagi mencari rute makanan yang aman dari berbagai ancaman. Strategi ini aktif tergantung dari ukuran kapal. Bila ukuran kapal kritis (sangat kecil), strategi ini akan memiliki prioritas tinggi. Selain itu, bila kapal belum mencapai ukuran ideal, prioritas akan normal. Bila sudah ideal, prioritas untuk mencari makanan akan menjadi rendah.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command forward dengan arah bebas
- 2) Himpunan solusi: Command forward yang mengarah pada arah makanan yang paling optimal
- 3) Fungsi solusi: Memilih arah makanan yang paling dekat dan aman dari ancaman
- 4) Fungsi seleksi: Membagi area objek relatif terhadap kapal menjadi enam bagian. Pilih bagian dengan ancaman terkecil dan makanan terbanyak. Setelah bagian terpilih, incar makanan paling dekat pada area tersebut.
- 5) Fungsi kelayakan: Memeriksa apakah arah makanan yang diincar optimal
- 6) Fungsi objektif: Memaksimalkan ukuran kapal

b. Analisis Efisiensi Solusi

Strategi ini pada dasarnya mengurutkan semua makanan pada peta berdasarkan jarak dan banyaknya ancaman. Pengurutan makanan memiliki kompleksitas $O(n \log n)$ dengan n adalah banyaknya makanan. Perhitungan ancaman memiliki kompleksitas $O(m)$ dengan m adalah banyaknya potensi ancaman.

c. Analisis Efektivitas Solusi

Strategi ini merupakan salah satu strategi dasar selain strategi Attacking. Solusi ini esensial karena strategi ini memiliki prioritas untuk meningkatkan ukuran kapal. Ukuran kapal sangat diperlukan untuk melakukan berbagai aksi, seperti mengaktifkan pelindung dan menembakkan torpedo. Dengan membagi kapan harus mengaktifkan explorer, kita bisa membagi prioritas untuk mencari makanan sehingga kapal bisa mendahulukan aksi lain bila ada yang memiliki prioritas lebih tinggi. Selain itu, strategi ini juga efektif untuk menjaga agar ukuran kapal tidak kritis dan mati.

F. Shield

Shield adalah strategi dengan pendekatan greedy yang akan memprioritaskan menyalaikan shield ketika terdapat banyak torpedo yang berisiko mengenai pemain. Ketika terdeteksi dua torpedo atau lebih yang akan mengenai pemain, bot akan memilih strategi Shield untuk menyalaikan shield agar pemain aman jika terkena torpedo.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command shield
- 2) Himpunan solusi: Command shield
- 3) Fungsi solusi: Menyalakan command shield.
- 4) Fungsi seleksi: Memeriksa apakah terdapat dua torpedo atau lebih yang lintasannya akan mengenai player.
- 5) Fungsi kelayakan: Memeriksa apakah player memiliki ukuran yang cukup untuk menyalaikan shield dan memeriksa apakah player memiliki shield.
- 6) Fungsi objektif: Meminimalkan damage akibat serangan torpedo.

b. Analisis Efisiensi Solusi

Untuk menghitung apakah torpedo akan mengenai player maka diperiksa range sudut dari kedua sisi pemain terhadap posisi torpedo. Jika heading torpedo berada dalam range tersebut serta berada dalam jarak tertentu dan terdapat dua atau lebih torpedo berbahaya, bot akan menyalaikan shieldnya apabila pemain memiliki ukuran yang cukup. Karena solusi Defender perlu memeriksa semua torpedo yang ada, kompleksitas solusi ini adalah sebesar **O(n)** dengan n berupa jumlah torpedo yang ada pada suatu saat.

c. Analisis Efektivitas Solusi

Bot akan memeriksa keberadaan torpedo berbahaya dan akan memanggil strategi ini jika bot memiliki ukuran yang cukup besar atau terdapat lawan yang sangat dekat dan baru menembakkan torpedo yang banyak.

Strategi efektif apabila:

- Terdapat sangat banyak torpedo yang akan mengenai player sehingga membuat strategi ini *worthwhile*.

Strategi tidak efektif apabila:

- Hanya terdapat sedikit torpedo yang mengenai player sehingga strategi ini membuang buang ukuran player.

G. Sniper

Sniper adalah strategi dengan pendekatan greedy yang akan memprioritaskan menembak lawan dengan ukuran yang besar. Ketika ukuran player sudah melebihi 50 dan terdapat lawan dengan selisih ukuran minimal 60, bot akan memilih strategi Sniper untuk menembak lawan tersebut dengan torpedo.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command menembak torpedo dengan heading bebas
- 2) Himpunan solusi: Command menembak torpedo dengan heading mengarah lawan terbesar.
- 3) Fungsi solusi: Menghitung heading optimal untuk menembak torpedo
- 4) Fungsi seleksi: Memilih heading yang menuju tengah arena.
- 5) Fungsi kelayakan: Memeriksa apakah player memiliki torpedo yang cukup untuk menembak lawan.
- 6) Fungsi objektif: Memaksimalkan kemungkinan torpedo yang ditembak mengenai target.

b. Analisis Efisiensi Solusi

Program bot akan memperhatikan ukuran player dan ukuran lawan untuk menentukan apakah strategi ini cocok untuk dilakukan. Jika cocok, untuk menentukan strategi membidik yang optimal, program akan melihat jarak player dengan target. Apabila target dekat dengan

pemain, strategi membidik yang menggunakan akan memperhatikan posisi dan heading dari target. Apabila target jauh, strategi membidik akan memperhatikan kecepatan sudut target dari perspektif player. Karena solusi Sniper hanya melakukan kondisional dan perhitungan matematis, kompleksitas solusi ini adalah sebesar **O(1)**.

c. Analisis Efektivitas Solusi

Bot akan memeriksa keberadaan lawan dengan ukuran yang selisihnya 60 unit lebih besar dari player yang minimal ukurannya 50 unit. Oleh karena itu, dalam solusi ini bot akan menembakkan torpedo ke lawan dengan ukuran terbesar dari jarak manapun, baik jauh maupun dekat.

Strategi efektif apabila:

- Lawan target sangat besar sehingga kemungkinan torpedo mengenai target lebih besar.

Strategi tidak efektif apabila:

- Lawan dekat dengan player sehingga berisiko untuk collision dengan player.
- Lawan masih relatif lincah dan pergerakannya sering berubah sehingga torpedo tidak kena karena tidak sesuai dengan prediksi bidikan.

H. To Center

To center adalah strategi dengan pendekatan greedy yang akan memprioritaskan player kembali ke area bermain. Ketika player mendekati batas arena, bot akan jalan menuju tengah arena hingga sudah jauh dari batas permainan.

a. Mapping Elemen Greedy

- 1) Himpunan kandidat: Command FORWARD dengan heading bebas
- 2) Himpunan solusi: Command FORWARD dengan heading menjauhi batas arena
- 3) Fungsi solusi: Menghitung heading optimal untuk menjauhi batas arena
- 4) Fungsi seleksi: Memilih heading yang menuju tengah .
- 5) Fungsi kelayakan: Memeriksa apakah player tetap berada dekat dengan batas arena.
- 6) Fungsi objektif: Meminimalisir damage akibat batas permainan.

b. Analisis Efisiensi Solusi

Program bot akan memperhatikan jarak player dengan batas arena. Jika bot berjarak 10 unit dari batas arena, bot akan mencari heading yang mengarah ke tengah arena. Karena solusi To Center hanya melakukan kondisional dan perhitungan matematis sederhana, kompleksitas solusi ini adalah sebesar **O(1)**.

c. Analisis Efektivitas Solusi

Program bot akan memeriksa jarak player dengan batasan arena, lalu akan mengarahkan player ke tengah arena.

Strategi efektif apabila:

- Arena masih besar sehingga penyusutan ukurannya tidak akan mengejar player.
- Tidak ada ancaman, seperti torpedo, gas cloud, dan lainnya, diantara player dengan arah tujuannya

Strategi tidak efektif apabila:

- Ada ancaman jika player menuju tengah arena
- Arena sudah kecil sehingga penyusutannya lebih cepat dan tetap memberi damage terhadap player.

Bab IV

Implementasi dan Pengujian

A. Implementasi dalam *Pseudocode*

1. Implementasi Aksi yang Dapat Dilakukan oleh Bot

Berikut adalah implementasi program di dalam *folder Actions* yang digunakan dalam pemilihan aksi untuk bot yang digunakan.

Class Armory

```
function calculateTorpedoHeading(player, target) -> int

Algoritma
if (Math.getDistanceBetween(player, target) > 300) then
    angularVelocity <- Math.calculateAngularVelocity(player, target)
    initialHeading <- Math.getHeadingBetween(player, target)
    distance <- Math.getDistanceBetween(player, target)

    adjustment <- 0

    if (angularVelocity ≥ 2) then
        adjustment <- adjustment + 5
    else if (angularVelocity ≤ -2) then
        adjustment <- adjustment - 5

    if (distance > 500) then
        adjustment <- adjustment * 2

    if (adjustment ≠ 0) then
        output(System.out.println("Torpedo firing adjustment"))

    -> initialHeading + adjustment;
else
    -> Math.getInterceptHeading(TORPEDO_SPEED,
        target.currentHeading, target.getSpeed(),
        target.position.x - player.position.x,
        target.position.y - player.position.y)
```

```
function fireTorpedo (target) -> PlayerAction
```

Algoritma

```

heading <- calculateTorpedoHeading(GameWatcherManager.
getWatcher().player, target)
-> fireTorpedoWithHeading(heading)

function fireTorpedoWithHeading(heading) -> PlayerAction

Algoritma
action <- new PlayerAction()
action.setHeading(heading)
action.setAction(PlayerActions.FIRETORPEDOES)
-> action

```

Class SearchEnemy

```

function closestEnemy() -> GameObject

Algoritma
watcher <- GameWatcherManager.getWatcher()
player <- watcher.player

if (watcher.enemies.size() = 0)then
    ->null

sortedEnemies <- watcher.enemies.stream().
    sorted(Comparator.comparing(enemy -> Math.
        getTrueDistanceBetween(player, enemy))).
    collect(Collectors.toList())

->sortedEnemies.get(0)

function largestEnemy() -> GameObject

Algoritma
if (watcher.enemies.size() = 0) then
    ->null

sortedEnemies <- watcher.enemies.stream().
    sorted(Comparator.comparing(enemy -> enemy.size * -1)).
    collect(Collectors.toList())

->sortedEnemies.get(0)

function enemyAboveRange(radius) -> List of GameObject

```

```

Algoritma
watcher <- GameWatcherManager.getWatcher()
player <- watcher.player

if (watcher.enemies.size() = 0)
    ->null

->watcher.enemies.stream().
    filter(enemy -> Math.getDistanceBetween(player, enemy) >= radius).
    collect(Collectors.toList())

```

Class SearchFood

```

function closestFood() -> GameObject

Algoritma
watcher <- GameWatcherManager.getWatcher()
-> closestFoodRelativeTo(watcher.player)

function closestFoodRelativeTo(target) -> GameObject

Algoritma
watcher <- GameWatcherManager.getWatcher()

if (watcher.foods.size() == 0) then
    -> null

sortedFood <- watcher.foods.stream().
    sorted(Comparator.comparing(food -> Math.
        getDistanceBetween(target, food))).
    collect(Collectors.toList())

-> sortedFood.get(0)

```

function closestSafestFood() -> GameObject

```

Algoritma
watcher <- GameWatcherManager.getWatcher()

potentialArea <- watcher.radar.getMostAdvantageousArea()

result <- null
int i <- 0

while (result = null and i < potentialArea.size()) do
    result <- potentialArea.get(i++).getClosestFood()

```

```
-> result
```

Class SearchSupernova

```
function getDangerousSupernova(player, gameObjects) -> GameObject

Algoritma
-> gameObjects.stream().filter(supernova ->
    supernova.gameObjectType = ObjectTypes.SUPERNOVA_BOMB
    and (Utils.Math.projectileWillHit(supernova, player,
        SUPERNOVA_ADDED_BEARINGS, SUPERNOVA_DANGER_DIST) or
        Utils.Math.getDistanceBetween(supernova, player)
        ≤ SUPERNOVA_DANGER_DIST)).findFirst().orElse(null)

function safestHeading() -> int

Algoritma
watcher <- GameWatcherManager.getWatcher()
player <- watcher.player
supernova <- getDangerousSupernova(player, watcher.others)

if (supernova = null) then
    -> player.currentHeading
else if (Utils.Math.getDistanceBetween(player, supernova) ≤
        watcher.world.radius / 4 and Utils.Math.headingDiff(
            Utils.Math.getHeadingBetween(supernova, player),
            supernova.currentHeading) ≥ 90) then
    -> Utils.Math.getHeadingBetween(supernova, player)
else
    -> supernova.currentHeading > Utils.Math.
        getHeadingBetween(supernova, player)
    ? (supernova.currentHeading - 90) % 360
    : (supernova.currentHeading + 90) % 360
```

Class SearchTeleporter

```
function filterDangerousTeleporters(player, gameObjects, enemies) ->
List of GameObject

Algoritma
enemies.stream().max(Comparator.comparing(enemy -> enemy.size)) .
    ifPresentOrElse((enemy) -> largestEnemySize = enemy.size,
    () -> largestEnemySize = 0)

-> gameObjects.stream().filter(teleporter -> teleporter.gameObjectType =
    ObjectTypes.TELEPORTER and (Utils.Math.projectileWillHit(
        teleporter, player, TELEPORTER_ADDED_BEARINGS,
```

```

TELEPORTER_DANGER_DIST) or Utils.Math.getDistanceBetween
(teleporter, player) <= largestEnemySize)).
collect(Collectors.toList())

```

```

function safestHeading() -> int

Algoritma
watcher <- GameWatcherManager.getWatcher()
player <- watcher.player
List<GameObject> teleporters <- filterDangerousTeleporters(player,
    watcher.others, watcher.enemies)

closestTeleporter <- teleporters.stream().
    min(Comparator.comparing(teleporter ->
        teleporter.currentHeading)).orElse(null)

if (closestTeleporter == null) then
    -> player.currentHeading

if (Utils.Math.getDistanceBetween(player, closestTeleporter)
    ≤ largestEnemySize and Utils.Math.headingDiff
    (Utils.Math.getHeadingBetween(closestTeleporter, player),
    closestTeleporter.currentHeading) > 60) then
    -> Utils.Math.getHeadingBetween(closestTeleporter, player)

-> (closestTeleporter.currentHeading + 90) % 360

```

Class SearchTorpedoes

```

function filterDangerousTorpedoes(player, gameObjects)-> List of Game
Object

Algoritma
-> gameObjects.stream().filter(gameObject -> gameObject.gameObjectType
    = ObjectTypes.TORPEDO_SLAVO).filter(torpedo -> Utils.Math.
        projectileWillHit(torpedo, player, TORPEDO_ADDED_BEARINGS,
        TORPEDO_DANGER_DIST)).collect(Collectors.toList())

```

```

function safestHeading() -> int

Algoritma
watcher <- GameWatcherManager.getWatcher()
player <- watcher.player
List<GameObject> torpedoes <- filterDangerousTorpedoes
    (player, watcher.torpedoes)

closestTorpedo <- torpedoes.stream().min(Comparator.
    comparing(torpedo -> torpedo.currentHeading)).orElse(null)

```

```

if (closestTorpedo = null) then
    -> player.currentHeading

if (Utils.Math.getDistanceBetween(player, closestTorpedo) ≤
    TORPEDO_EXTREMELY_CLOSE and Utils.Math.headingDiff(Utils.
    Math.getHeadingBetween(closestTorpedo, player),
    closestTorpedo.currentHeading) > 45)
    -> Utils.Math.getHeadingBetween(closestTorpedo, player)

-> (closestTorpedo.currentHeading + 90) % 360

```

2. Implementasi Strategi Algoritma *Greedy* pada Bot

Berikut adalah implementasi program di dalam *folder Strategy* yang digunakan sebagai strategi dalam penerapan algoritma *greedy*.

Class Attacker

```

function computeNextAction() -> PlayerAction

Algoritma
if (this.target = null) then
    -> null

{ cek jika memerlukan heading adjustment }
adjustment <- headingAdjustment(false)

int SHIP_SAFE_SIZE = 50
if (this.watcher.player.torpedoSalvoCount ≥ 5 and
    (this.watcher.player.size > SHIP_SAFE_SIZE or
     this.watcher.enemies.size() = 1 and
     this.watcher.player.size ≥ 15)) then
    -> Armory.fireTorpedo(this.target)

if (adjustment ≠ null) then
    -> clearSuddenTurn(adjustment)

torpedoTreshold <- 0

if (this.watcher.player.torpedoSalvoCount > torpedoTreshold) then
    if ((this.watcher.radar.clearToShoot(this.target) and Math.
        getTrueDistanceBetween(this.watcher.player, this.target)
        < 700) or (this.watcher.enemies.size() = 1 and
        this.watcher.player.size ≥ 15)) then
            -> Armory.fireTorpedo(this.target)

newAdjustment <- headingAdjustment(true)

if (this.watcher.foods.size() ≤ 3 or newAdjustment = null)

```

```

-> oppositeWay()

-> clearSuddenTurn(newAdjustment)

function clearSuddenTurn(action) -> PlayerAction

Algoritma
headingDiff <- Math.getModulus(this.watcher.player.
                                currentHeading - action.heading, 360)

if (headingDiff ≥ 160 and headingDiff ≤ 200) then
    output("Adjustment called")
    if (headingDiff ≤ 180) then
        action.setHeading(Math.getModulus
                          (action.getHeading() - 30, 360))
    else
        action.setHeading(Math.getModulus
                          (action.getHeading() + 30, 360))
-> action

```

```

function oppositeWay() ->PlayerAction

Algoritma
act = new PlayerAction()
act.setAction(PlayerActions.FORWARD)
act.setHeading(Math.getModulus(this.target.currentHeading + 180, 360))
-> act

```

```

function headingAdjustment(forceNew) ->PlayerAction

Algoritma
if (this.target = null)then
    -> null

targetSection <- watcher.radar.determineSection(this.target)

if (forceNew) then
    currentHeading <- null
else
    currentHeading <- watcher.radar.heading

radarSectionCount <- watcher.radar.sectionCount
enemyLarger <- watcher.player.size - 10 < this.target.size

{fokus mengejar tetapi tetap jaga jarak }
angularVelocity <- Math.calculateAngularVelocity
                    (this.watcher.player, this.target)

headingDiff <- Math.getModulus(target.currentHeading -
                                this.watcher.player.currentHeading, 360)

```

```

ANGULAR_VELOCITY_OFFSET <- 2

if (enemyLarger) then
    if (angularVelocity < ANGULAR_VELOCITY_OFFSET * (-1)) then
        safeSection <- new Integer[]
            {(targetSection - 3) % radarSectionCount,
             (targetSection - 2) % radarSectionCount}
    else if (angularVelocity > ANGULAR_VELOCITY_OFFSET) then
        safeSection <- new Integer[]
            {(targetSection + 3) % radarSectionCount,
             (targetSection + 2) % radarSectionCount}
    else
        if (headingDiff > 150 && headingDiff < 210) then
            safeSection <- new Integer[] {
                targetSection + radarSectionCount / 2,
                (targetSection - 2) % radarSectionCount,
                (targetSection - 3) % radarSectionCount,
                (targetSection + 2) % radarSectionCount,
                (targetSection + 3) % radarSectionCount}

        else if (headingDiff < 30) then
            safeSection <- new Integer[] {targetSection,
                (targetSection - 1) % radarSectionCount,
                (targetSection + 1) % radarSectionCount,
                (targetSection - 2) % radarSectionCount,
                (targetSection + 2) % radarSectionCount}

        else
            safeSection <- new Integer[] {
                (targetSection - 1) % radarSectionCount,
                (targetSection + 1) % radarSectionCount,
                (targetSection - 2) % radarSectionCount,
                (targetSection + 2) % radarSectionCount}

else
    if (angularVelocity < ANGULAR_VELOCITY_OFFSET * (-1)) then
        safeSection <- new Integer[] {
            targetSection,
            (targetSection - 1) % radarSectionCount,
            (targetSection - 2) % radarSectionCount}
    else if (angularVelocity > ANGULAR_VELOCITY_OFFSET) then
        safeSection <- new Integer[] {
            targetSection,
            (targetSection + 1) % radarSectionCount,
            (targetSection + 2) % radarSectionCount}
    else
        if (headingDiff > 150 && headingDiff < 210) then
            safeSection <- new Integer[] {
                targetSection + radarSectionCount / 2,
                (targetSection - 1) % radarSectionCount,
                (targetSection + 1) % radarSectionCount}

        else if (headingDiff < 30) then
            safeSection <- new Integer[] {targetSection,

```

```

        (targetSection - 1) % radarSectionCount,
        (targetSection + 1) % radarSectionCount}

    else
        safeSection <- new Integer[] {
            (targetSection - 1) % radarSectionCount,
            (targetSection + 1) % radarSectionCount}

    if (currentHeading != null) then
        currentHeadingSection <- this.watcher.radar.
            determineSection(currentHeading)

        if (List.of(safeSection).contains(currentHeadingSection)) then
            -> null

potentialAreas <- this.watcher.radar.getMostAdvantageousAreaIn
                (safeSection)
-> foodTargetFromAreaList(potentialAreas)

```

```
function foodTargetFromAreaList(scannedArea) -> PlayerAction
```

Algoritma

```

foodTarget <- null
i <- 0

while (foodTarget = null and i < scannedArea.size()) do
    foodTarget <- scannedArea.get(i++).getClosestFood()

if (foodTarget = null) then
    -> null

action = new PlayerAction()
action.setAction(PlayerActions.FORWARD)
action.setHeading(Math.getHeadingBetween(this.watcher.player,
    foodTarget))

-> action

```

```
function getPriorityLevel() -> int
```

Algoritma

```

target <- SearchEnemy.closestEnemy()
{ target direset setiap tick, sehingga tidak akan berisi target pada
state sebelumnya }

if (target = null) then
    this.target <- null
    -> Priority.NONE
else if (Math.getTrueDistanceBetween(this.watcher.player, target) < 600
or this.watcher.enemies.size() = 1) then
    this.target <- target

```

```

-> Priority.NORMAL

this.target <- null
-> Priority.NONE

```

Class Defender

```

function computeNextAction() -> PlayerAction

Algoritma
act <-new PlayerAction()
act.setAction(PlayerActions.FORWARD)

if (supernovaBomb # null) then
    act.setHeading(SearchSupernova.safestHeading())
else if (not dangerousTeleporters.isEmpty()) then
    act.setHeading(SearchTeleporter.safestHeading())
else
    act.setHeading(SearchTorpedoes.safestHeading())

-> act

function getPriorityLevel() -> int

Algoritma
this.dangerousTorpedoes <- SearchTorpedoes.filterDangerousTorpedoes
(watcher.player, watcher.torpedoes)
this.supernovaBomb <- SearchSupernova.getDangerousSupernova
(watcher.player, watcher.others)
this.dangerousTeleporters <-
SearchTeleporter.filterDangerousTeleporters
(watcher.player, watcher.others, watcher.enemies)

if (supernovaBomb # null or not dangerousTeleporters.isEmpty()) then
    -> Priority.EMERGENCY

if (not dangerousTorpedoes.isEmpty()) then
    -> Priority.HIGH

-> Priority.NONE

```

Class EscapeGas

```

function computeNextAction() -> PlayerAction

Algoritma

```

```

gases <- this.watcher.gasClouds.stream().sorted(Comparator.
    comparing(gas -> -1 * Math.getDistanceBetween
        (this.watcher.player, gas))).collect(Collectors.toList())

if (gases.size() = 0) then
    -> null

action = new PlayerAction()
action.setAction(PlayerActions.FORWARD)
action.setHeading(Math.getHeadingBetween(gases.get(0), this.
    watcher.player))
-> action

function getPriorityLevel() -> int

Algoritma
if (this.watcher.player.isInsideGasCloud()) then
    -> Priority.EMERGENCY

-> Priority.NONE

```

Class EvadeCollision

```

function computeNextAction() -> PlayerAction

Algoritma
if (this.target = null) then
    -> null

action <- new PlayerAction()
action.setAction(PlayerActions.FORWARD)

targetHeading <- this.target.currentHeading
headingBetween <- Math.getHeadingBetween(this.target, this.
    watcher.player)
headingDiff <- targetHeading - headingBetween

if (headingDiff ≥ -180 && headingDiff ≤ 180)
    { starboard side }
    action.setHeading(Math.getModulus(watcher.player.
        currentHeading - 90, 360))
else
    { port side }
    action.setHeading(Math.getModulus(watcher.player.
        currentHeading + 90, 360))

-> action

```

```
function getPriorityLevel() -> int
```

```

Algoritma
target <- SearchEnemy.closestEnemy()

if (target = null) then
    this.target <- null
    -> Priority.NONE

if (Math.getTrueDistanceBetween(target, this.watcher.player) > 200) then
    -> Priority.NONE

tick <- Math.tickTillCollision(target, this.watcher.player)

if (tick ≤ 2.5 and target.size + 15 > watcher.player.size) then
    this.target <- target
    -> Priority.EMERGENCY

-> Priority.NONE

```

Class Explorer

```

function computeNextAction() -> PlayerAction

Algoritma
act <- new PlayerAction()

food <- SearchFood.closestSafestFood()
if (food = null) then
    food <- SearchFood.closestFood()
    if (food = null) then
        act.setHeading(Math.getCenterHeading(this.watcher.player))
        act.setAction(PlayerActions.FORWARD)
        -> act

    act.setHeading(Math.getHeadingBetween(watcher.player, food))
    act.setAction(PlayerActions.FORWARD)

    currentHeading <- this.watcher.radar.heading

    if (currentHeading ≠ null) then
        headingDiff <- Math.getModulus(watcher.player.
            currentHeading - act.getHeading(), 360)
        { unstuck player }
        if (headingDiff > 150 && headingDiff < 210) then
            act.setHeading(watcher.player.currentHeading)
        else
            if (this.watcher.player.torpedoSalvoCount ≥ 5 and
                this.watcher.player.getSize() ≥ SHIP_SIZE_CRITICAL * 2) then
                { Kalo ada gas cloud, tembak gas cloud }

```

```

gas <- this.watcher.radar.closestGasCloud()
if (gas ≠ null)
    output("Firing gas clouds")
    -> Armory.fireTorpedo(gas)
else
    closestEnemy = SearchEnemy.closestEnemy()
    -> Armory.fireTorpedo(closestEnemy)

-> act

function getPriorityLevel() -> int

Algoritma
if (this.watcher.foods.size() = 0) then
    -> Priority.NONE

averageEnemySize <- this.watcher.enemies.stream().mapToDouble
    (enemy -> enemy.size).average()

if (averageEnemySize.isEmpty()) then
    idealShipSize <- SHIP_SIZE_IDEAL
else
    idealShipSize <- java.lang.Math.max(averageEnemySize.
        getAsDouble(), SHIP_SIZE_IDEAL)

if (watcher.player.size ≤ SHIP_SIZE_CRITICAL) then
    -> Priority.HIGH
else if (watcher.player.size ≤ idealShipSize) then
    -> Priority.NORMAL
else
    -> Priority.LOW

```

Class Shield

```

function computeNextAction() -> PlayerAction

Algoritma
act <- new PlayerAction()
act.setAction(PlayerActions.ACTIVATESHIELD)

-> act

```

```
function getPriorityLevel() -> int
```

```

Algoritma
closestEnemy <- SearchEnemy.closestEnemy()
dangerousEnemy <- false

```

```

if (closestEnemy != null) then
    dangerousEnemy <- Math.getTrueDistanceBetween
        (this.watcher.player, closestEnemy) < 60

if (this.watcher.player.shieldCount > 0 and SearchTorpedoes.
    filterDangerousTorpedoes(watcher.player, watcher.torpedoes) .
    size() ≥ 2 and (this.watcher.player.getSize() ≥ 80 or
    (dangerousEnemy and watcher.player.size > 30))) then
    -> Priority.EMERGENCY

-> Priority.NONE

```

Class Sniper

```

function computeNextAction() -> PlayerAction

Algoritma
if (this.target = null) then
    -> null

-> Armory.fireTorpedo(this.target)

function getPriorityLevel() -> int

Algoritma
largestEnemy <- SearchEnemy.largestEnemy()

if (largestEnemy = null) then
    this.target <- null
    -> Priority.NONE

if (watcher.player.torpedoSalvoCount > 0 and watcher.player.size > 30
    and (largestEnemy.size - watcher.player.size) > 30) then
    this.target <- largestEnemy
    -> Priority.HIGH

this.target <- null
-> Priority.NONE

```

Class Supernova

```

function computeNextAction() -> PlayerAction

Algoritma

```

```

-> null

function getPriorityLevel() -> int

Algoritma
potentialTarget <- SearchEnemy.enemyAboveRange(500)
if (this.watcher.player.supernovaAvailable = 1 and potentialTarget ≠
    null and potentialTarget.size() ≠ 0) then
    -> Priority.EMERGENCY

-> Priority.NONE

```

Class ToCenter

```

function computeNextAction() -> PlayerAction

Algoritma
act = new PlayerAction()
act.setHeading(Math.getCenterHeading(this.watcher.player))
act.setAction(PlayerActions.FORWARD)
-> act

function getPriorityLevel() -> int

Algoritma
if (Math.isOutOfBounds(this.watcher.player,
    this.watcher.world.radius - 10)) then
    -> Priority.EMERGENCY

-> Priority.NONE

```

3. Implementasi Perhitungan

Berikut adalah implementasi dari *class Math* yang digunakan untuk menunjang segala perhitungan yang dilakukan oleh *class* lainnya.

Class Math

```

function getDistanceBetween(object1, object2) -> float

Algoritma
-> sqrt(deltax^2 + deltay^2)

function getTrueDistanceBetween(object1, object2) -> float

Algoritma

```

```

-> getDistancebetween(object1, object2) - object1.size - object2.size

function getHeadingBetween(object1, object2) -> int

Algoritma
-> arctan(deltaY/deltaX)

function getCenterHeading(object) -> int

Algoritma
-> -1 * arctan(object.y/object.x)

function headingDiff(theta1, theta2) -> int

Algoritma
-> abs(theta1 - theta2) % 360

function getInterceptHeading(projectileSpeed, targetHeading,
targetSpeed, xDist, yDist) -> int

Algoritma
tgtDeg <- toRadians(targetHeading)
adjAngle <- java.lang.Math.atan(yDis / xDis)
tgtAspect <- targetSpeed * (java.lang.Math.sin(tgtDeg) -
(yDis / xDis * java.lang.Math.cos(tgtDeg)))
gamma <- java.lang.Math.asin(-1 * java.lang.Math.cos(adjAngle) * 
tgtAspect / projectileSpeed)
res <- toDegrees(adjAngle - gamma)

{Check if it is possible to intercept with the current angle}
verificator <- yDis / (projectileSpeed * java.lang.Math.
sin(toRadians(res)) - targetSpeed * java.lang.Math.
sin(toRadians(targetHeading)))

output("Intercepting Torpedos Deployed")
if (verificator < 0) then
-> toDegrees(adjAngle + gamma - java.lang.Math.PI)
else
-> res

function potentialIntercept(target, projectile) -> boolean

Algoritma
targetHeading <- headingBetween(projectile, target)
Tolerance <- atan(target.size/distanceBetween(target, projectile))
-> abs(targetHeading - projectile.currentHeading) ≤ tolerance

function potentialInterceptFood(player, food) -> boolean

Algoritma
distance <- getDistanceBetween(player, food)
angleTolerance <- asin(player.size/distance)
-> abs(player.currentHeading - getHeadingbetween(player, food) <
angletolerance

```

```

function calculateAngularVelocity(pivot, target) -> int

Algoritma
relativeHeading <- getHeadingBetween(pivot,target) -
                    target.currentHeading
speedPerpendicular <- cos(relativeHeading * target.speed)
-> speedPerpendicular / getDistancebetween(pivot, target)

function isOutOfBound(player, radius) -> boolean

Algoritma
-> sqrt(player.x**2 + player.y**2) + player.size - radius

function trueDistanceRaw(deltax, deltay,r1, r2) -> int

Algoritma
-> sqrt(deltax**2 + deltay**2) - r1 - r2

function tickTillCollision(player1, player2) -> float

Algoritma
Tick <- 0

while (trueDistanceRaw(player1, player2) >= 0 do
    append player position based on speed per unit tick

-> tick

function projectileWillHit(projectile, object, addedDegrees,
minDistance) -> bool

Algoritma
dangerHeading <- toDegrees( java.lang.Math.abs
                            (java.lang.Math.asin(object.size/ getDistanceBetween
                            (projectile, object)))) + addedDegrees
-> getDistanceBetween(projectile, object) < minDistance) and
headingDiff(getHeadingBetween(projectile,object),
projectile.currentHeading) < dangerHeading

```

B. Struktur Data Program

Bot yang digunakan dalam permainan *Galaxio* ini dikembangkan menggunakan bahasa pemrograman Java. Paradigma yang digunakan dalam pengembangan bot tersebut adalah paradigma berorientasi objek. Dengan begitu, setiap data yang digunakan untuk keberlangsungan bot disimpan dalam bagian yang disebut kelas (*class*). Setiap *class* yang

telah dibuat memiliki fungsinya masing-masing sesuai dengan bagaimana *class* tersebut dibutuhkan. Secara umum, program yang telah dibuat dapat dibagi menjadi tujuh bagian besar, yaitu enam *folders* antara lain *Actions*, *Agents*, *Enums*, *Models*, *Services*, dan *Utils*, serta sebuah *main file* yang digunakan untuk mengelola keenam bagian lainnya.

```

|- src
  |- main
    |- java
      |- Actions
        |- Armory.java
        |- SearchEnemy.java
        |- SearchFood.java
        |- SearchSupernova.java
        |- SearchTorpedoes.java
      |- Agents
        |- Ling                                # Used Bot
          |- Strategy
            |- Attacker.java
            |- Defender.java
            |- Explorer.java
            |- EscapeGas.java
            |- EvadeCollision.java
            |- Shield.java
            |- Sniper.java
            |- Supernova.java
            |- ToCenter.java
          |- Ling.java
          |- Priority.java
          |- StrategyInterface.java
          |- Agent.java
          |- AgentManager.java
        |- Enums
          |- ObjectTypes.java
          |- PlayerActions.java
        |- Models
          |- GameObject.java
          |- Gamestate.java
          |- GamestateDto.java
          |- PlayerAction.java
          |- Position.java
          |- World.java
        |- services
          |- DebugWriter.java
          |- GameWatcher.java
          |- GameWatcherManager.java
          |- Radar.java
          |- RadarSection.java
          |- RadarUnitArea.java
        |- utils
          |- Math.java
      |- Main.java

```

Gambar 4.1 Struktur Data Program

1. Actions

Folder Actions terdiri dari semua *class* yang digunakan untuk menunjang aksi yang dapat dilakukan oleh bot. Bagian ini mencakup enam *classes* antara lain:

a. Armory

Class Armory digunakan untuk menunjang bot dalam melakukan aksi serangan. *Class* ini mencakup fungsi untuk menghitung arah torpedo (*calculateTorpedoHeading*), menembakkan torpedo (*fireTorpedo*), dan menembakkan torpedo dengan arah tertentu (*fireTorpedoWithHeading*).

b. SearchEnemy

Class SearchEnemy digunakan oleh bot saat bot berada pada fase akan menyerang. Dalam hal ini, *class* mencakup semua fungsi yang digunakan untuk mencari target serangan, yaitu fungsi untuk mencari musuh terdekat (*closestEnemy*), mencari musuh terbesar (*largestEnemy*), dan mencari musuh di luar radius tertentu (*enemyAboveRange*).

c. SearchFood

Class SearchFood digunakan oleh bot saat bot berada pada fase *exploring* atau sedang mencari makan. *Class* ini terdiri dari fungsi untuk mencari makanan terdekat (*closestFood*) dan makanan terdekat yang aman (*closestSafestFood*).

d. SearchSupernova

Class SearchSupernova digunakan oleh bot untuk menghindari *supernova* yang ditembakkan oleh lawan. *Class* ini terdiri dari fungsi untuk mendeteksi adanya *supernova* yang ditembakkan oleh lawan (*getDangerousSupernova*) dan mencari arah gerak paling aman (*safestHeading*).

e. SearchTeleporter

Class SearchTeleporter digunakan oleh bot untuk mendeteksi kedatangan *teleporter* lawan. *Class* ini mencakup

fungsi untuk menyaring teleporter yang berbahaya (*filterDangerousTeleporter*) dan mencari sudut arah gerak yang aman (*safestHeading*).

f. SearchTorpedoes

Class SearchTorpedoes digunakan oleh bot saat bot berada pada fase bertahan. *Class* ini terdiri atas fungsi-fungsi yang digunakan bot untuk mendeteksi kedatangan torpedo lawan. Fungsi-fungsi tersebut antara lain, fungsi untuk menyaring torpedo yang berbahaya (*filterDangerousTorpedoes*) dan mencari sudut arah gerak yang aman (*safestHeading*).

2. Agents

Folder Agents merupakan *folder* yang menampung program utama dari bot yang telah dibuat. Dengan kata lain, ‘sejatinya’ bot yang telah dikembangkan disimpan dalam *folder Agents*. *Folder* ini mengandung *file Agent.java* dan *AgentManager.java* yang digunakan untuk pengaturan bot. Di sisi lain, di dalam *folder Agents* juga terdapat *folder Ling*. Dalam hal ini, *folder Ling* terdiri atas sebuah *folder Strategy*, *file Ling.java*, *Priority.java*, dan *StrategyInterface.java*. Secara singkat, *Ling.java* adalah file yang mengandung *class Ling* yang mengatur seluruh strategi yang ada di *folder Strategy*. Di sisi lain, *Priority.java* mengandung *settingan* tingkat prioritas yang digunakan dalam setiap *class* lainnya dan *StrategyInterface.java* mengandung pengambilan tingkat prioritas yang perlu diutamakan oleh bot. Terakhir, *folder Strategy* mengandung sembilan *classes* antara lain, *Attacker*, *Defender*, *EscapeGas*, *EvadeCollision*, *Explorer*, *Shield*, *Sniper*, *Supernova*, dan *ToCenter*.

a. Attacker

Class Attacker berisi implementasi dari strategi yang digunakan bot dalam menembak, mengejar, dan memakan target.

Dengan begitu, tentunya *class* ini memanfaatkan implementasi program dari *class* *Armory* dan *SearchEnemy*.

b. Defender

Class *Defender* berisi implementasi dari strategi yang digunakan untuk bertahan dan mengelak dari serangan lawan. *Class* ini memanfaatkan implementasi program dari *class* *SearchTorpedoes*, *SearchTeleporter* dan *SearchSupernova*.

c. EscapeGas

Class *EscapeGas* berisi implementasi dari strategi yang digunakan untuk menghindari objek *gas cloud*.

d. EvadeCollision

Class *EvadeCollision* berisi implementasi dari strategi yang digunakan untuk menghindari terjadinya tabrakan (*collision*) dengan bot lain. Implementasi dari *class* ini digunakan bila terjadinya tabrakan dengan bot yang memiliki ukuran lebih besar. Dalam penggunaannya, *class* ini menggunakan implementasi dari *class* *SearchEnemy*.

e. Explorer

Class *Explorer* berisi implementasi dari strategi yang digunakan untuk saat bot menjelajahi *map*. Dalam hal ini, menjelajahi *map* mencakup pencarian makanan yang dianggap aman, menembak objek *gas cloud*, dan menembak lawan yang berada pada posisi dekat dengan bot. Dalam penggunaannya, *class* ini memanfaatkan implementasi dari *class* *Armory*, *SearchFood*, dan *SearchEnemy*.

f. Shield

Class Shield berisi implementasi dari strategi yang digunakan bot untuk mengaktifkan *shield* ketika dibutuhkan. *Class* ini memanfaatkan implementasi dari *class SearchEnemy* dan *SearchTorpedoes*.

g. Sniper

Class Sniper berisi implementasi dari strategi yang digunakan bot untuk menembakkan torpedo dari jarak jauh. Target utama dalam implementasi *class* ini adalah bot musuh yang memiliki ukuran yang tergolong besar. *Class* ini memanfaatkan implementasi dari *class Armory* dan *SearchEnemy*.

h. Supernova

Class Supernova berisi implementasi dari strategi yang digunakan bot untuk menembakkan supernova ke arah lawan. *Class* ini memanfaatkan implementasi dari *class SearchEnemy*. Dalam hal ini, dengan berbagai pertimbangan, penggunaan *class Supernova* tidak jadi diimplementasikan.

i. ToCenter

Class ToCenter berisi implementasi dari strategi yang digunakan bot untuk mencegah bot bergerak keluar dari batas *map*. Pada dasarnya, arah gerak bot akan secara manual dipindahkan ke pusat *map* bila bot mendekat ke pembatas *map*.

3. Enums

Folder Enums berisi objek-objek dalam game yang enumerasikan. Terdapat dua *files* yang berada di dalam *folder* ini, yaitu *ObjectTypes.java* yang berisikan segala objek yang ada di dalam game *Galaxio* dan *PlayerActions.java* yang mengandung semua aksi valid yang dapat dilakukan oleh bot.

4. Models

Folder Models merupakan *folder* dimana keterangan dan atribut dari objek-objek di dalam game *Galaxio* disimpan. *Folder* ini berisikan enam *files* yaitu *GameObject.java*, *GameState.java*, *GameStateDto.java*, *PlayerAction.java*, *Position.java*, dan *World.java*.

5. Services

Folder Services menampung semua implementasi program yang digunakan untuk menunjang kelangsungan bot. Pada *folder* ini terdapat *file* yang digunakan untuk membantu pelaksanaan debug yaitu *DebugWriter.java*, *files* yang digunakan untuk menampung pengambilan *Game State* yaitu *GameWatcher.java* dan *GameWatcherManager.java*, dan *files* yang digunakan sebagai radar untuk mendeteksi objek di sekitar bot pada radius tertentu yaitu *Radar.java*, *RadarSection.java*, dan *RadarUnitArea.java*.

6. Utils

Folder Utils berisi implementasi program yang digunakan sebagai ‘perlengkapan’ bagi implementasi program *class* lainnya. *Folder* ini berisikan *class* *Math* pada *file* *Math.java* yang mengandung semua fungsi dan prosedur perhitungan yang digunakan untuk menunjang seluruh *class* lainnya.

7. Main

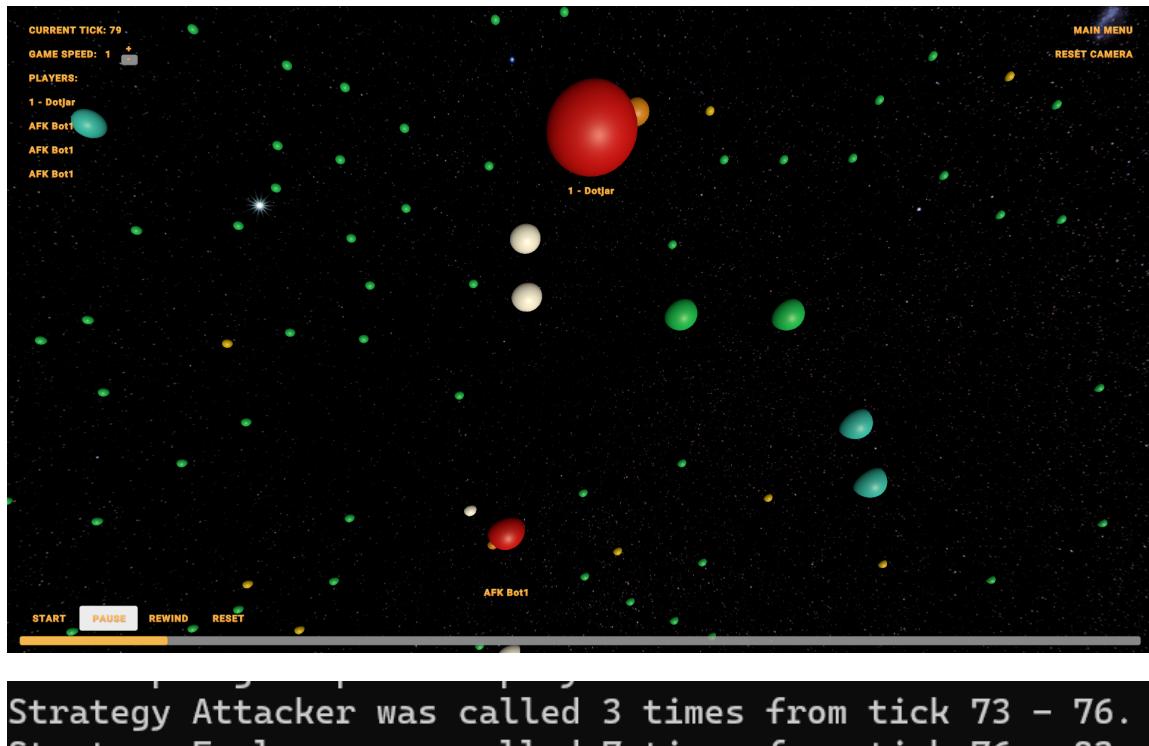
File Main.java mengandung segala implementasi dari pengaturan bot selama permainan. Hal ini mencakup penamaan dari bot dan penampilan tanda bahwa bot telah dimakan oleh bot lainnya dan/atau permainan telah selesai.

C. Analisis dan Pengujian

Kami melakukan analisis dan pengujian dengan mempertarungkan bot yang kami buat dengan bot lain sesuai dengan solusi greedy yang ingin kami uji. Kami mengamati tingkah laku

dari bot kami menggunakan visualizer yang sudah disediakan, serta dengan melihat output program kami yang menunjukkan strategi apa yang sedang dijalankan pada tick tertentu. Konfigurasi pengaturan permainan yang kami gunakan adalah konfigurasi default sehingga world permainan akan dihasilkan secara random. Menurut kami, penggunaan bot lawan yang cocok cukup untuk memvisualisasi strategi greedy yang digunakan.

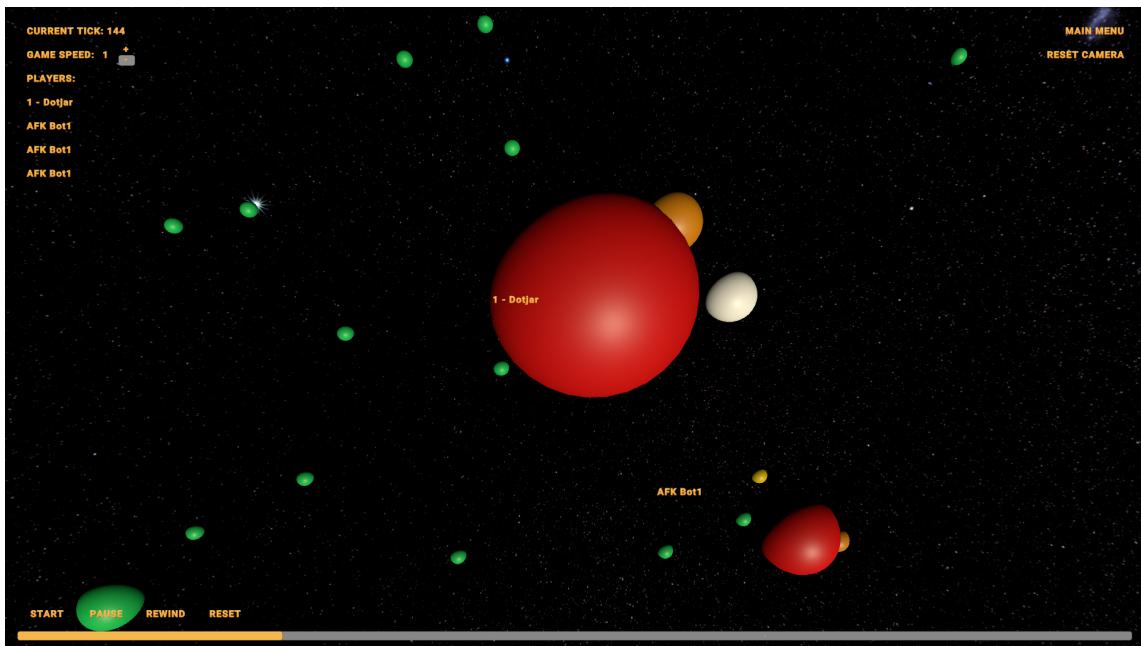
1. Attacker



Gambar 4.1.1 dan 4.1.2 Ilustrasi strategi attacker

Pada tick 79 pada gambar di atas, dapat terlihat aksi strategi attacker yang terjadi pada tick sebelumnya. Strategi attacker tersebut terjadi karena terdapat lawan pada radius yang dekat sehingga kapal player menembakkan torpedo pada kapal lawan yang terdekat.

2. Defender

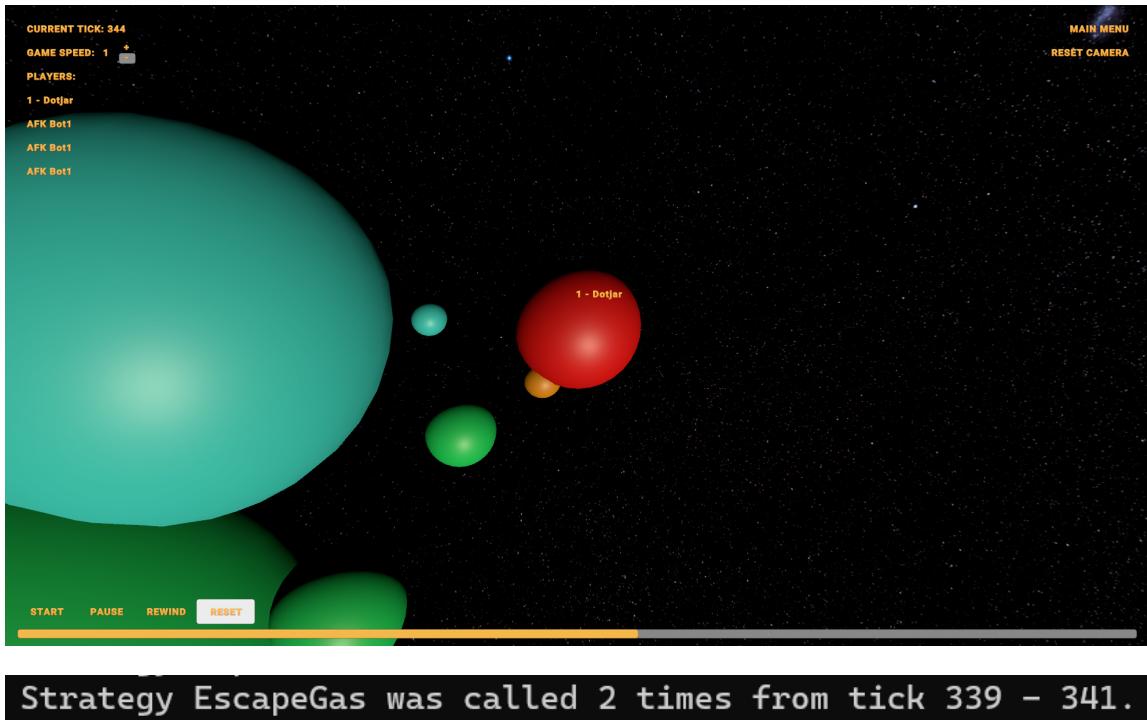


Strategy Defender was called 1 times from tick 144 - 145.

Gambar 4.2.1 dan 4.2.2 Ilustrasi strategi defender

Pada gambar di atas, kapal player menggunakan strategi defender untuk menghindari torpedo (lingkaran putih) yang mengarah ke kapal player. Player menghindari torpedo dengan maju searah tegak lurus lintasan torpedo. Gambar di atas menunjukkan salah satu contoh kegagalan dalam strategi defender karena torpedo yang menyerang sudah terlalu dekat dengan player dan player terlalu besar dan pelan untuk menghindar.

3. Escape Gas



Gambar 4.3.1 dan 4.3.2 Ilustrasi strategi escape gas

Pada gambar di atas, kapal player menggunakan strategi escape gas untuk keluar dari gas cloud yang tadinya ia masuki. Oleh karena itu, player mengarah menjauh dari gas cloud yang paling dekat. Gambar diatas menunjukkan contoh penggunaan strategi escape gas yang efektif.

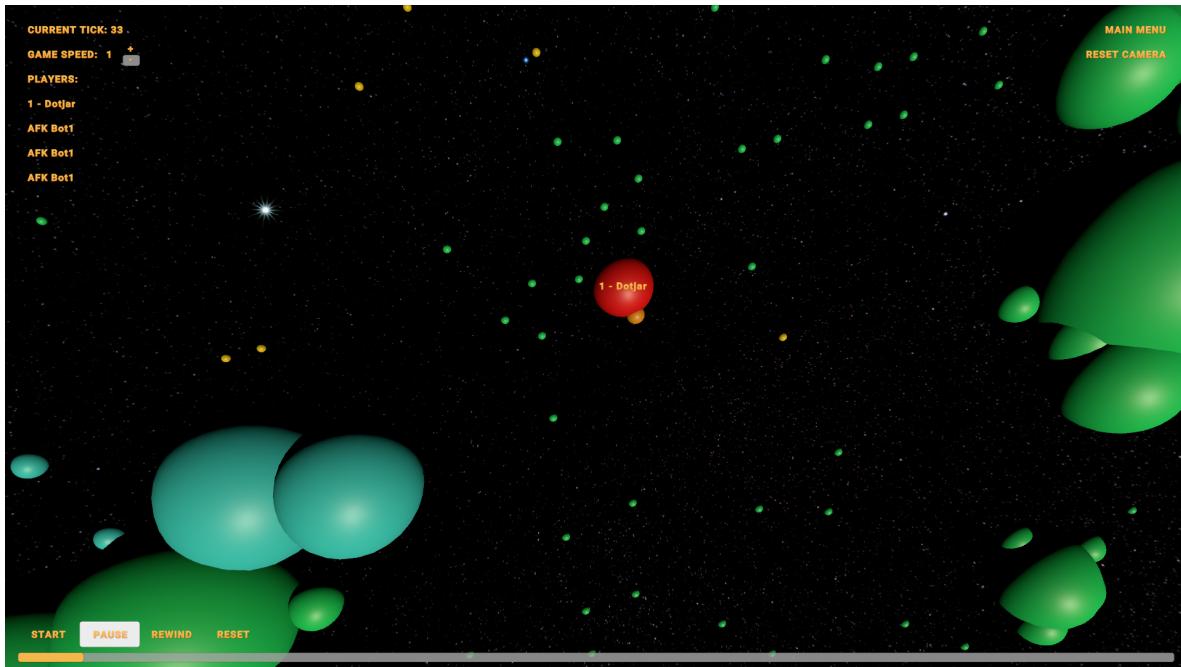
4. Evade Collision



Gambar 4.4.1 dan 4.4.2 Ilustrasi strategi evade collision

Pada gambar diatas, dapat dilihat bahwa kapal dotjar hampir menabrak kapal lawan (agario). Strategi ini berhasil mencegah termakannya kapal dengan cara bergerak menjauh dari kapal lawan.

5. Explorer

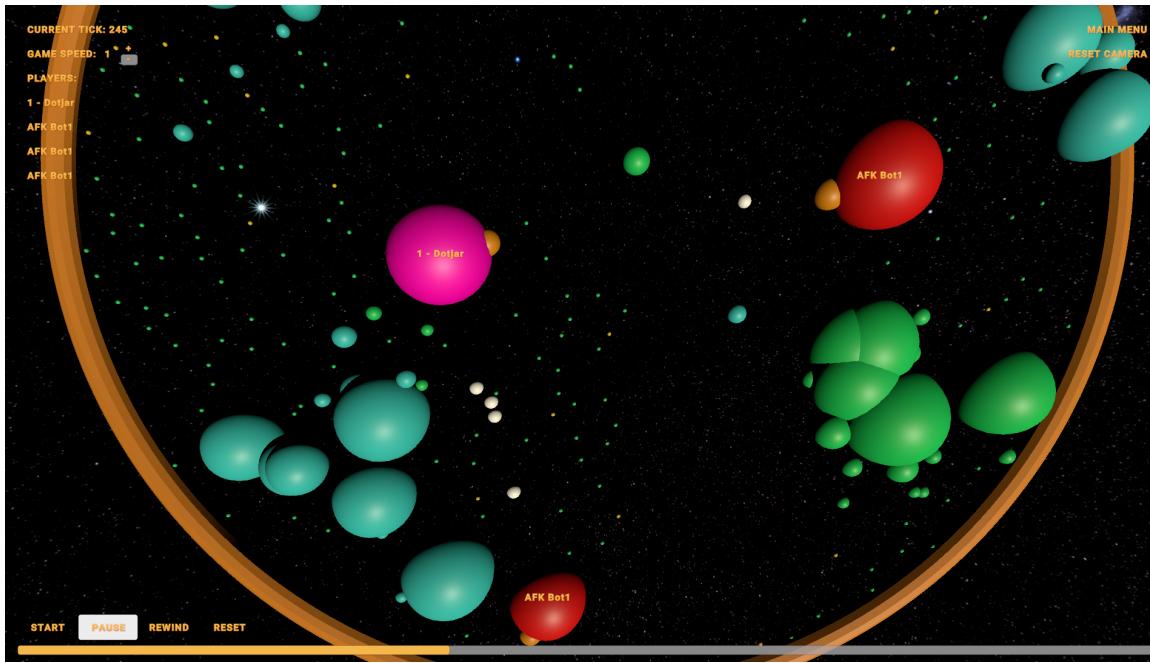


Strategy Explorer was called 105 times from tick 9 – 114.

Gambar 4.5.1 dan 4.5.2 Ilustrasi strategi explorer

Gambar di atas menunjukkan penggunaan strategi explorer ketika kapal masih berukuran kecil pada awal game. Kapal pada strategi explorer akan mencari makanan sebanyak-banyaknya hingga ukurannya cukup dan selama situasi saat itu belum ada ancaman terhadap kapal player.

6. Shield

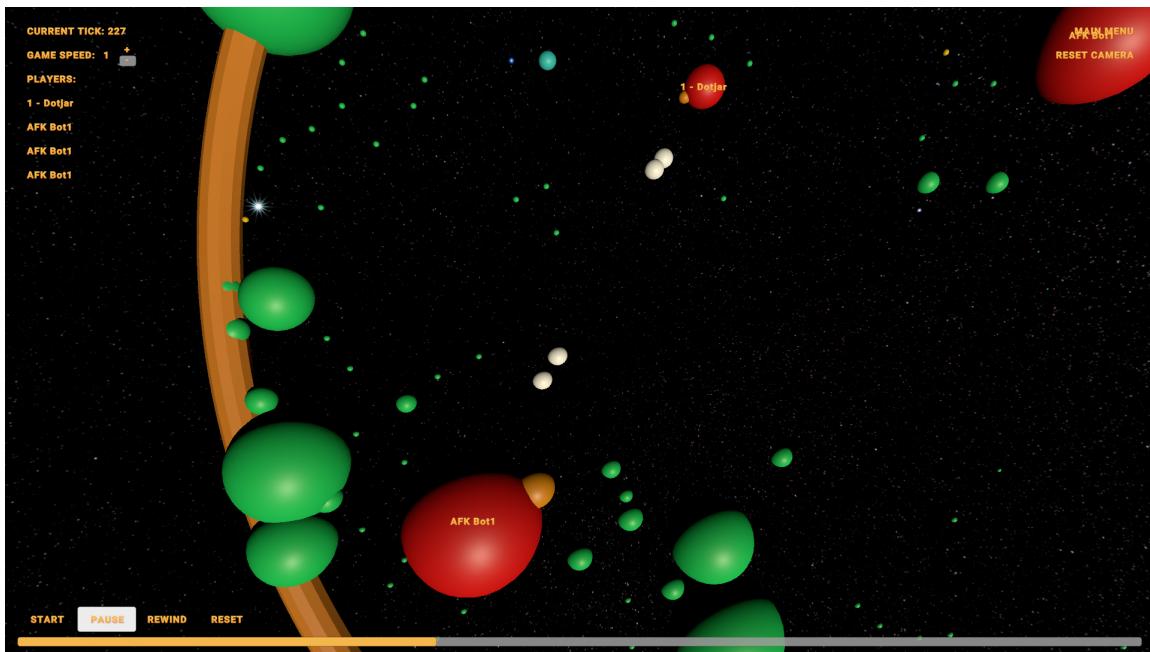


Strategy Shield was called 2 times from tick 242 – 244.

Gambar 4.6.1 dan 4.6.2 Ilustrasi strategi shield

Gambar di atas menunjukkan penggunaan strategi shield oleh kapal player ketika kapal player diserang oleh banyak torpedo sekaligus. Terdapat 5 torpedo yang akan melintasi kapal player. Karena ukuran kapal player sudah cukup besar, strategi shield diimplementasikan dan shield kapal menyala.

7. Sniper

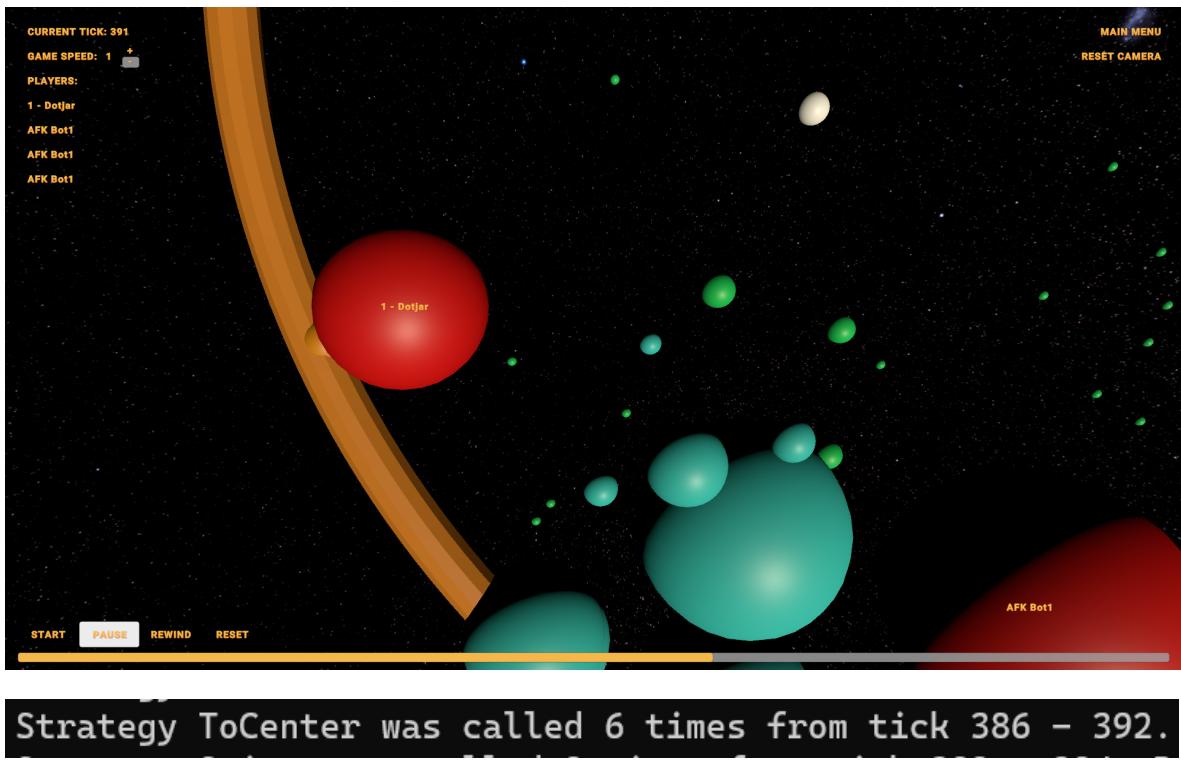


Strategy Sniper was called 2 times from tick 222 – 224.

Gambar 4.7.1 dan 4.7.2 Ilustrasi strategi attacker

Pada gambar diatas, dapat dilihat bahwa bot mengaktifkan strategi *sniper* selama dua tick permainan. Dalam melakukan strategi tersebut, selayaknya seorang *sniper*, bot menembakkan torpedo ke arah kapal lawan dari jarak yang jauh. Berdasarkan gambar di atas, torpedo digambarkan dengan bola-bola putih yang ditembakkan bot *dotjar* kepada bot lawan. Strategi ini berguna untuk mencegah tumbuhnya kapal lawan menjadi sangat besar sehingga kapal lawan tidak bisa menggunakan strategi teleporter.

8. To Center



Gambar 4.8.1 dan 4.8.2 Ilustrasi strategi attacker

Gambar di atas menunjukkan implementasi strategi to center dimana kapal player dekat dengan batas arena dan bergerak mengarah tengah arena. Gambar diatas menunjukkan salah satu contoh strategi to center berjalan secara kurang efektif dikarenakan arena yang menyusut secara cepat

Bab V

Kesimpulan dan Saran

A. Kesimpulan

Dalam tugas besar ini, kami mengimplementasikan strategi algoritma *greedy* untuk membuat bot di dalam permainan Galaxio. Agar algoritma maksimal, program memanfaatkan berbagai strategi *greedy* yang diperoleh berdasarkan prioritas yang diperhatikan dari aspek-aspek dalam permainan saat itu. Algoritma *greedy* yang digunakan bertujuan untuk memanfaatkan *resources* yang dimiliki bot serta kondisi permainan agar dapat menentukan langkah terbaik dan paling menguntungkan agar bot yang kami buat dapat memenangkan permainan.

B. Saran

Saran untuk kelompok kami adalah sebagai berikut,

1. Pembagian tugas, kerjasama, dan perencanaan pengembangan program yang ditingkatkan agar hasil lebih maksimal,
2. Pengembangan algoritma lebih lanjut agar lebih efisien dalam mencapai objektif akhir,
3. Penambahan komentar dan dokumentasi yang lebih baik untuk memudahkan kerja sama dan *maintenance source code* program.

Daftar Pustaka

Entelect Challenge. 2023. “Game Rules”. Diakses 17 Februari 2023, dari <https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/game-engine/game-rules.md>

Munir, Rinaldi. 2018. “Algoritma Greedy”. Diakses 17 Februari 2023, dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Greedy-%282018%29.pdf>

Lampiran

A. *Repository Github*

Bot yang telah dirancang dapat diakses pada *link repository* Github sebagai berikut: https://github.com/akbarmridho/Tubes1_.jar

B. *Link Video YouTube*

Video yang dibuat untuk pemenuhan aspek bonus dapat diakses pada *link* sebagai berikut: <https://youtu.be/MlgjSC9oU6c>