LAPORAN
TUGAS BESAR 1 IF2211 STRATEGI ALGORITMA
IMPLEMENTASI ALGORITMA GREEDY MENGGUNAKAN BOT GALAXIO

oleh

Wildan Ghaly Buchary	**13521015**
Raditya Naufal Abiyu	**13521022**
Fatih Nararya Rashadyfa	**13521060**

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG

2022

DAFTAR ISI

BAB 1

DESKRIPSI TUGAS

1.1 Spesifikasi Tugas

Pada tugas besar pertama mata kuliah Strategi Algoritma, mahasiswa diminta untuk mengimplementasikan algoritma *greedy* untuk membuat sebuah bot untuk permainan Galaxio. Aturan permainannya adalah sebagai berikut :

1. Peta permainan berbentuk kartesius yang memiliki arah positif dan negatif. Peta hanya menangani angka bulat. Kapal hanya bisa berada di integer x,y yang ada di peta. Pusat peta adalah 0,0 dan ujung dari peta merupakan radius. Jumlah ronde maximum pada game sama dengan ukuran radius. Pada peta, akan terdapat 5 objek, yaitu Players, Food, Wormholes, Gas Clouds, Asteroid Fields. Ukuran peta akan mengecil seiring batasan peta mengecil.
2. Kecepatan kapal dilambangkan dengan x. Kecepatan kapal akan dimulai dengan kecepatan 20 dan berkurang setiap ukuran kapal bertambah. Ukuran (radius) kapal akan dimulai dengan ukuran 10. Heading dari kapal dapat bergerak antar 0 hingga 359 derajat. Efek afterburner akan meningkatkan kecepatan kapal dengan faktor 2, tetapi mengecilkan ukuran kapal sebanyak 1 setiap tick. Kemudian kapal akan menerima 1 salvo charge setiap 10 tick. Setiap kapal hanya dapat menampung 5 salvo charge. Penembakan slavo torpedo (ukuran 10) mengurangkan ukuran kapal sebanyak 5.

3. Setiap objek pada lintasan punya koordinat x,y dan radius yang mendefinisikan ukuran dan bentuknya. Food akan disebarkan pada peta dengan ukuran 3 dan dapat dikonsumsikan oleh kapal player. Apabila player mengkonsumsi Food, maka Player akan bertambah ukuran yang sama dengan Food. Food memiliki peluang untuk berubah menjadi Super Food. Apabila Super Food dikonsumsi maka setiap makan Food, efeknya akan 2 kali dari Food yang dikonsumsi. Efek dari Super Food bertahan selama 5 tick.
4. Wormhole ada secara berpasangan dan memperbolehkan kapal dari player untuk memasukinya dan keluar di pasangan satu lagi. Wormhole akan bertambah besar setiap tick game hingga ukuran maximum. Ketika Wormhole dilewati, maka wormhole akan mengecil sebanyak setengah dari ukuran kapal yang melewatinya dengan syarat wormhole lebih besar dari kapal player.
5. Gas Clouds akan tersebar pada peta. Kapal dapat melewati gas cloud. Setiap kapal bertabrakan dengan gas cloud, ukuran dari kapal akan mengecil 1 setiap tick game. Saat kapal tidak lagi bertabrakan dengan gas cloud, maka efek pengurangan akan hilang.
6. Torpedo Salvo akan muncul pada peta yang berasal dari kapal lain. Torpedo Salvo berjalan dalam lintasan lurus dan dapat menghancurkan semua objek yang berada pada lintasannya. Torpedo Salvo dapat mengurangi ukuran kapal yang ditabraknya. Torpedo Salvo akan mengecil

apabila bertabrakan dengan objek lain sebanyak ukuran yang dimiliki dari objek yang ditabraknya.

7. Supernova merupakan senjata yang hanya muncul satu kali pada permainan di antara quarter pertama dan quarter terakhir. Senjata ini tidak akan bertabrakan dengan objek lain pada lintasannya. Player yang menembakannya dapat meledakannya dan memberi damage ke player yang berada dalam zona. Area ledakan akan berubah menjadi gas cloud.
8. Player dapat meluncurkan teleporter pada suatu arah di peta. Teleporter tersebut bergerak dalam direksi dengan kecepatan 20 dan tidak bertabrakan dengan objek apapun. Player tersebut dapat berpindah ke tempat teleporter tersebut. Harga setiap peluncuran teleporter adalah 20. Setiap 100 tick player akan mendapatkan 1 teleporter dengan jumlah maximum adalah 10.
9. Ketika kapal player bertabrakan dengan kapal lain, maka kapal yang lebih besar akan 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 :

Enums	*Corresponding Actions*

FORWARD	Bergerak maju
STOP	Berhenti
START_AFTERBURNER	Nyalakan *afterburner*
STOP_AFTERBURNER	Matikan *afterburner*
FIRE_TORPEDOES	Tembak torpedo
FIRE_SUPERNOVA	Tembak *supernova*
DETONATE_SUPERNOVA	Ledakkan *supernova*
FIRE_TELEPORTER	Luncurkan *teleporter*
TELEPORT	Teleportasi ke lokasi *teleporter*
USE_SHIELD	Menyalakan pelindung

11. Setiap player akan memiliki score yang hanya dapat dilihat jika permainan berakhir.
12. 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

2.1 *Greedy Algorithm*

Greedy algorithm (*henceforth* disebut GA) adalah sebuah kelas algoritma yang paling deskriptif dijelaskan dengan sebuah analogi (*courtesy to* Bu Ulfa atas analoginya) yaitu seperti seorang pendaki gunung mencoba meraih puncak ketika gelap dan berkabut. Puncak gunung dalam analogi tersebut adalah solusi yang ingin dicari oleh algoritma, pendaki adalah si algoritmanya sendiri, dan tiap langkah yang ia buat adalah langkah yang dijalankan algoritma tersebut untuk menghasilkan sebuah solusi. Pendaki gunung pada tiap langkah yang diambilnya hanya bisa melihat satu atau dua meter ke depan, tidak punya *foresight* untuk melihat keseluruhan bentuk gunung. Dari penglihatannya yang terbatas, ia hanya bisa menentukan – langkah demi langkah – arah mana yang paling mungkin membawanya ke puncak gunung. *As such*, GA juga tidak bisa melihat keseluruhan solusi yang ia buat seiring ia berjalan membuat solusi tersebut melalui langkah iteratif. Ia hanya bisa, pada tiap langkahnya, menilai langkah selanjutnya yang mana yang paling mungkin untuk membawanya ke solusi. Singkatnya, algoritma ini punya rabun dekat.

Contoh paling mudah dari sebuah GA adalah algoritma yang biasa kita jalankan dalam kehidupan sehari-hari ketika ingin memberikan kembalian kepada orang lain. Kita tentunya mencari susuan denominasi sedemikian rupa sehingga jumlah lembar uang yang kita berikan seminimal mungkin. Untuk sebuah nilai

uang X, kita akan mengambil pecahan uang paling besar yang masih lebih kecil dari X sebagai salah satu bagian dari pecahan, sebutlah nilai ini Y. X kemudian kita kurangkan terhadap Y, lalu kita ulangi langkah tersebut untuk nilai yang dihasilkan sampai sisa uang kita menjadi 0. Pada setiap langkah, kita – secara intuitif – mencari pecahan yang paling "cepat" "menghabiskan" nilai X.

Short-sightedness dari GA – yang hanya peduli dengan situasi sekarang – adalah pisau bermata dua. Di satu sisi, hal itu membuat GA dapat memiliki kompleksitas waktu yang relatif rendah daripada solusi seperti *brute force algorithm* tanpa memerlukan cara kerja yang kompleks seperti *divide and conquer*. Di sisi lain, rabun dekat yang ia miliki juga membuat solusi yang ia hasilkan tidak terjamin sebagai solusi yang optimal. Kembali ke analogi dalam paragraf sebelumnya, pada suatu waktu sang pendaki gunung bisa saja sampai di sebuah bagian dari gunung yang lebih tinggi daripada bagian sekitarnya langsung, namun sendirinya bukan puncak gunung yang sebenarnya, misalnya rumah semut yang membentuk gundukan tinggi. Jika sudah sampai pada titik itu, pendaki gunung akan berhenti karena merasa bahwa ia sudah berada di puncak. Ekspansi dari analogi tersebut menggambarkan hampir persis bagaimana GA terkadang tidak menemukan solusi yang paling optimal daripada semua solusi yang ada, tetapi hanya solusi yang optimal secara lokal.

Kasus cukup *trivial* yang dapat mendemonstrasikan bagaimana *lack of foresight* yang dimiliki GA ada pada contoh persoalan mencari kembalian yang telah disebutkan. Misalkan kembalian yang ingin dibuat bernilai 9000 dan pecahan uang yang ada adalah 1000, 3000, dan 5000. Secara intuitif, pecahan

yang meminimumkan lembar uang yang diberikan adalah 3 lembar 3000. Tetapi, GA akan memberikan jawaban 1 lembar 5000 dan 4 lembar 1000.

2.2 Galaxio

Permainan ini – seperti kebanyakan permainan lainnya – berjalan di tiap *tick* yang berdurasi 75ms. Pada tiap *tick*, permainan (dan *bot* termasuk di dalamnya) akan berjalan. Sehingga, pada tiap *tick*, *bot* dapat memilih suatu aksi yang ingin dilakukannya pada *tick* selanjutnya dan arah haluan miliknya yang akan menjadi tujuan haluannya di *tick* selanjutnya.

Untuk membuat sebuah *bot*, hanya ada satu *file* dari *game* ini yang perlu menjadi fokus, yaitu `BotService.java` yang berisi sebuah kelas (*you guessed it*) `BotService` yang membungkus *bot* yang kita buat. Kelas inilah yang akan kita atur dan modifikasi untuk membuat *bot* yang kita anggap optimal dan memiliki *sound logic*.

Kelas `BotService` tersebut memiliki akses ke *state* dari permainan melalui instansiasi dari kelas `GameState` yang ada di dalamnya. Dari *instance* tersebut, *bot* dapat mendapatkan akses ke hal-hal seperti ukuran dari *world*, semua objek yang ada di dalam permainan (*list* berisi semua instansiasi kelas `GameObject`), dan lain-lain.. Sementara itu, *behaviour* dari *bot* sendiri ditentukan oleh dua hal yang telah disebutkan sebelumnya. Maka, untuk tiap *tick* terdapat sebuah *tuple* beranggotakan dua yang dengan sempurna menggambarkan aksi dari *bot*. Anggota pertama *tuple* tersebut adalah arah haluan dan anggota keduanya adalah aksi yang dilakukan oleh bot sebagaimana telah diberikan pada tabel di bab 1. Pemrogram kemudian dapat membuat logika yang menggunakan

informasi-informasi status dari permainan yang telah disebutkan untuk menghasilkan *tuple* yang telah disebutkan di tiap *tick* dari permainan.

Game engine dijalankan dengan menjalankan *script* `run.sh` yang ada di dalam *starter-pack* yang diberikan. Menentukan berapa bot yang bermain dalam suatu permainan ditentukan dengan memodifikasi *file* `appsettings.json` di dalam *folder* `engine-publish` dan `logger-publish`. Menentukan *bot* apa yang bermain dilakukan dengan memodifikasi *script* `run.sh` yang telah disebutkan dengan menggunakan jumlah *bot* sesuai yang telah diatur di `appsettings.json` dan *path* yang menunjuk ke *executable* dari *bot* yang ingin dijalankan.

BAB III

APLIKASI GA

3.1 Pemetaan Persoalan

Definisi dari GA yang telah disebutkan pada subbab 2 menggambarkan GA dengan sangat akurat tetapi secara tidak formal. Secara formal, GA memiliki beberapa komponen atau bagian dengan peran yang berbeda-beda. Berikut adalah komponen-komponen tersebut beserta analoginya di dalam permainan ini.

Komponen GA	Perwujudannya dalam Galaxio
Himpunan kandidat (K)	Semua sekuens aksi yang mungkin dan bisa dijalankan oleh *bot* di tiap *tick*-nya ia hidup. Karena aksi dari *bot* telah direpresentasikan oleh sebuah *tuple* yang telah disebutkan di bab sebelumnya, maka K adalah sebuah *list* berisi *tuple* tersebut. Pada banyak *problem* lain, K mudah sekali untuk didefinisikan sehingga *problem* lain dapat diselesaikan secara *brute-force*. Tetapi, dalam permainan ini tidak mungkin untuk membuat K dalam waktu yang pendek karena dua hal berikut : 1. Diberikan sebuah sebarang sekuens aksi *bot*, *virtually* tidak mungkin mengetahui *in advance* apakah *bot* akan mati di tengah sekuens tersebut, di akhir, atau bahkan tidak mati. 2. Panjang sebuah permainan Galaxio bervariasi karena menunggu satu pemain tersisa. Akibatnya, panjang dari *list* yang telah disebutkan juga tidak diketahui.
Himpunan solusi (S)	S adalah sebuah sekuens aksi *bot* yang berakhir dengan *bot* mati ataupun menang.
Fungsi solusi	Pengecekan apakah S telah membentuk solusi – *bot* berakhir, entah menang ataupun mati – telah diselesaikan oleh permainan sendiri.

(r)	Dalam kasus *bot* menang, permainan akan otomatis berhenti dan nama pemenang akan ditampilkan ke layar. Dalam kasus *bot* mati, permainan akan menyingkirkan *bot* dari permainan.
Fungsi seleksi (b)	Fungsi yang menentukan pilihan terbaik dari semua *tuple* yang ada. *Tuple* yang dipilih oleh fungsi ini seharusnya sudah dicek kelayakannya oleh w.
Fungsi kelayakan (w)	Pengecekan apakah suatu aksi mungkin dilakukan atau tidak. Contoh, penembakan torpedo tidak bisa dilakukan jika *bot* tidak memiliki *charge* torpedo. Contoh lain, apakah *bot* telah melakukan aksi lain pada *tick* yang sekarang.
Fungsi objektif (o)	Sejatinya, banyak aksi (atau konsekuensi dari aksi) yang dapat diminimalkan di dalam permainan ini dan semuanya tergantung dari bentuk GA yang diimplementasikan oleh pemrogram. Contoh, jumlah torpedo yang ditembakkan ataupun makanan yang dikonsumsi adalah sesuatu yang bisa dimaksimalkan. Akan tetapi, apakah optimalisasi dari contoh aspek yang baru saja disebutkan menjamin kemenangan? Jawabannya *unlikely*. Torpedo, walau banyak ditembakkan, akan percuma jika tidak ada yang mengenai target. Makanan, walaupun banyak yang dikonsumsi, akan percuma jika *bot* kita malah pergi ke daerah berbahaya seperti awan gas untuk makanan. Maka, ada dua kemungkinan untuk keberadaan fungsi objektif di dalam permainan ini : 1. Tidak ada di dalam *bot* untuk permainan ini karena memang butuh pendekatan yang lebih *nuanced* dibanding "maksimalkan *X*" atau "minimumkan *Y*" 2. Ada di dalam *bot* tapi tidak mudah untuk ditemukan karena telah di-*embed* bersama fungsi-fungsi lainnya dan menjadi satu dengan keseluruhan program.

Pada implementasi *bot* nantinya, pembedaan antara *b*, *w*, dan *o* akan sulit untuk dilakukan karena mereka akan "dijahit" ke dalam satu sama lain supaya program lebih ringkas dan *streamlined*.

3.2 GA Satu Dimensi dan Kelemahannya

Pada persoalan-persoalan *textbook*, umumnya ada suatu kuantitas spesifik yang dioptimalkan menggunakan GA. Misalnya pada persoalan mencari kembalian pada bab sebelumnya, yang ingin dioptimalkan hanyalah satu, yaitu meminimalisasi jumlah kembalian. Akibatnya, GA yang digunakan pun cukup fokus pada satu hal tersebut saja. GA ini akan disebut sebagai satu dimensi karena ia hanya peduli tentang satu hal saja.

Pendekatan satu dimensi membuat GA menjadi sederhana dan optimal dalam menjalankan tugasnya. Tetapi pendekatan ini tidak akan bekerja untuk permainan Galaxio karena persoalan dari permainan ini secara fundamental tidak terjadi dalam satu dimensi. Tidak ada kemenangan yang bisa didapatkan hanya dengan memaksimalkan satu aspek saja. Sebagai contoh, maksimalisasi pencarian makanan akan membuat *bot* kita menjadi pasif dan mudah dibunuh oleh lingkungan maupun pemain lain. Contoh lain, maksimalisasi penyerangan terhadap pemain lain akan berujung dengan *bot* kita menjadi kecil dan mudah mati.

3.3 GA Multi-Dimensi dan GGAP

Jelas dari subbab sebelumnya bahwa permainan ini menuntut kita untuk bisa mengatur dan menyeimbangkan berbagai faktor seperti pencarian makanan, penyerangan pemain, penghindaran dari *environmental damage*, dan lain-lain agar

bisa menang. Akibatnya, dibutuhkanlah suatu GGA (Galaxio's Greedy Algorithm) yang mampu melihat seluruh dimensi dari permainan atau bisa disebut multi-dimensional. Bagaimana kita bisa membuat GGA yang mampu menelisik seluruh bagian dari permasalahan yang ada? Kami yakin bahwa terdapat GGAP (Galaxio's Greedy Algorithm Pattern) dalam membuat sebuah GGA yang bertujuan untuk melakukan hal tersebut. GGAP seperti namanya adalah pola, pola yang digunakan suatu GGA untuk melakukan penyeimbangan faktor-faktor yang ada. Perlu dicatat bahwa GGAP bisa saja digunakan untuk membuat GGA satu dimensi, tetapi GGAP yang ada baru benar-benar tampak berbeda satu sama lain ketika digunakan untuk mengimplementasikan GGA multi-dimensi. Sebuah GGA tidak musti *strictly* menggunakan satu GGAP saja, melainkan ia bisa menggunakan GGAP yang berbeda di bagian-bagian algoritma yang berbeda.

Terdapat dua GGAP yang kami amati keberadaannya : diskrit-khusus (DK) dan kontinu-umum (KU).

3.3.1 Diskrit-Khusus (DK)

GGAP ini bercirikan pencarian sebuah atau banyak kondisi tertentu dari `GameState` yang kemudian ketika terdeteksi akan menyebabkan aksi tertentu dari *bot*. Secara singkat, ada sebuah atau sekumpulan kondisi K yang memiliki korespondensi langsung dengan sebuah aksi A.

Bagaimana contoh algoritma ini? Perhatikan *pseudocode* berikut. Nama fungsi dan nama variabel *self-explanatory* dan menggambarkan persis apa yang ia lakukan.

```
if (distanceToBorder >= maximumDistanceToBorder) {
     headingToRadius = getHeadingToRadius()
```

```
        setHeading((180 + headingToRadius) % 360)
}
```

Pada *pseudocode* tersebut, *bot* secara spesifik mencari apakah ia jauh dari perbatasan atau tidak lalu menjauh dari perbatasan jika iya. Ini adalah karakteristik paling murni dari GGAP DU.

Pertanyaannya kemudian adalah apakah GGAP ini ideal? Tidak. Kelemahan besar dari GGAP ini adalah kurangnya *nuance* dari pengambilan keputusan oleh *bot*. Ambillah contoh algoritma menghindari perbatasan yang telah diberikan sebelumnya, bagaimana jika di arah sebaliknya ada *player* lain yang lebih besar, salvo, atau awan gas? Apakah iya berbalik ke arah sebaliknya adalah pilihan yang bagus? Tidakkah mungkin lebih baik bot mengambil rute kabur dengan keluar dari perbatasan untuk sementara? Mencoba mengimplementasikan *nuance* dengan tetap mengadopsi GGAP ini cukuplah sulit dan terbatas karena faktanya ada jutaan kondisi yang mungkin, tidak mungkin membuat program mampu memperhitungkan secara diskrit dan khusus jutaan kemungkinan kondisi tersebut dengan menaruh jutaan *block* `if else`.

Tetapi, bukan berarti GGAP ini sama sekali harus dihindari. Terdapat kasus di mana pengambilan keputusan oleh *bot* dapat bergantung pada beberapa kondisi sederhana. Misalnya, ketika makanan sudah habis, menggunakan GGAP ini *in moderation* kita dapat membuat *bot* mengambil *behaviour* yang berbeda daripada ketika awal permainan seperti meningkatkan kecenderungannya untuk menembakkan torpedo. Atau contoh lain lagi, ketika terdapat sebuah *player* yang berukuran lebih kecil daripada *bot*, *bot* dapat dibuat untuk mengejar *player* tersebut untuk dimakan.

3.3.2 Kontinu-Umum (KU)

Ujung spektrum ini bercirikan komputasi sebuah *quantifiable metric* untuk semua aksi bot yang mungkin berdasarkan `GameState`, lalu mengaplikasikan fungsi seleksi tertentu untuk memilih aksi mana yang optimal berdasarkan *quantifiable metric* yang telah dikalkulasi tersebut. Semakin umum sifat dari fungsi seleksi yang dipakai, maka semakin dekat sebuah GGA dengan GGAP ini. Singkat cerita, GGAP ini melakukan pemilihan aksi secara *impersonal*, *cold, and calculated*.

Contoh dari GGAP ini cukup sulit untuk ditampilkan secara sederhana karena ia umumnya membutuhkan struktur data dan kelas yang kompleks, tidak cukup hanya dengan sebuah *block* `if else`. Meski begitu, berikut adalah contoh dari algoritma dengan GGAP KU.

```
headingScore = [0, 0, 0, 0, … ,0] // inisialisasi array
sepanjang 360
// iterasikan seluruh derajat yang ada
for (i from 0 to 360) {
   // iterasi seluruh GameObject yang ada
     for (gameObject in gameObjects) {
     if (inTheWay(i, gameObject)) {
     //  jika  obyek  ada  di  arah  derajat  i,  maka
     tambahkan skor di arah i tersebut
         headingScore[i] += scoreCalculator(gameObject)
     }
```

```
    }
}
// dapatkan arah dengan skor paling tinggi
bestHeading = getHighestScore(headingScore)
// set arah bot ke arah tersebut
setHeading(bestHeading)
```

Contoh tersebut adalah miniaturisasi dari GGA yang digunakan oleh *bot* kami *branch* `fatih-augmentation`. Jika ingin mengetahui contoh lebih dalam dari GGAP ini, GGA dalam *branch* tersebut sangat dekat dengan GGAP KU murni.

Dalam *pseudocode* tersebut, GGA sedang mencari arah yang ideal untuk *bot* bergerak maju. Ia mengkalkulasi hal itu dengan menghitung dan menambahkan berat tiap objek yang dihitung berdasarkan sebuah fungsi tertentu. Setelah semua arah yang ada dikalkulasi nilainya, kita tinggal mengambil arah dengan nilai yang paling tinggi saja.

Kelebihan dari GGAP ini adalah modularitas program yang terasa seperti sebuah fitur *built-in*. Misalnya pada contoh yang telah diberikan, perhitungan dari "berat" suatu objek di fungsi scoreCalculator bisa mudah dimodifikasi sesuai dengan yang kita mau sesuai prioritas kita dalam GGA yang diinginkan. Contohnya, dalam mendesain GGA, kita bisa saja menginginkan *bot* kita untuk menjauh dari awan gas, tetapi mendekati makanan. Maka kita bisa membuat kalkulator penghitung skor untuk memberi skor negatif pada awan gas dan skor positif pada makanan.

Kelemahan dari GGAP ini adalah penulis program tidak sepenuhnya tahu dan apa yang sebenarnya dilakukan oleh *bot* karena seperti yang telah disebutkan, GGAP ini *impersonal*. Bisa jadi terdapat banyak *unforeseen consequences* dari apa yang telah ditulis oleh pemrogram, baik yang menghasilkan kemenangan dan yang tidak. Perlu banyak *fine tuning* dan pengujian untuk memastikan bahwa rumusan-rumusan yang dipakai adalah optimal.

Selain itu, jika ada suatu hal spesifik yang pemrogram ingin bot untuk lakukan, GGA ini tidak bisa melakukan hal tersebut dengan GGAP ini, tanpa menggunakan GGAP DK. Seperti namanya, GGAP ini memberikan solusi umum, tidak bisa digunakan untuk melakukan hal yang khusus dalam domain yang diberikan.

3.3.3 Kombinasi GGAP yang Ideal

Implementasi salah satu dari dua GGAP yang ada secara murni tidak akan menghasilkan *bot* yang baik karena kelemahan-kelemahan inheren yang telah disebutkan dari masing-masingnya. Sama halnya dengan implementasi paradigma OO atau fungsional secara murni tidak akan menghasilkan program yang baik. GGA yang baik menggunakan GGAP yang sesuai dengan kebutuhan strategi yang dipakainya, tidak terpaku dengan kepatuhan atau kemurnian untuk salah GGAP yang ada.

Contoh kombinasi GGAP yang ideal adalah penggunaan DK di tingkat tertinggi dari GGA, sehingga *bot* memiliki perubahan *behaviour* ketika misalnya ada awan gas di dekatnya, ada *player* lebih kecil di dekatnya, atau *food* di dalam

map sudah habis. Akan tetapi kemudian *behaviour* eksak ditentukan oleh KU sehingga *bot* pada tingkat paling bawah memiliki sistem pemilihan keputusan yang mampu menangani banyak situasi berbeda.

3.4 GGA Fatih

GGA yang dibuat oleh Fatih condong menggunakan GGAP KU. Kelemahan KU yang membutuhkan *fine tuning* dan banyaknya eksperimen sementara *deadline* sudah dekat menjadi alasan kenapa GGA ini, walau ketika disketsakan di atas kertas terasa seperti GGA yang sangat jenius, tidak jadi diimplementasikan.

Cara kerja GGA ini terdapat 4 langkah :

1. Inisialisasi sebuah kelas penyimpan skor sepanjang 360 dengan nilai awal semua skornya 0.
2. Iterasi tiap GameObject yang ada. Tambahkan skor dari GameObject tersebut ke semua derajat di mana trajektori *bot* jika memilih derajat tersebut akan menyebabkan tabrakan dengan GameObject tersebut.
3. Dari skor penyimpan derajat, carilah interval derajat bernilai positif yang lebarnya paling lebar, lalu pilih derajat tengah interval tersebut sebagai *heading*.

Algoritma ini tidak cukup jauh dalam fase pengembangannya sebelum dibatalkan untuk dipakai sehingga tidak dikembangkan sampai untuk meng-*handle* torpedo, teleporter, dan aksi-aksi lain dari pemain.

3.4.1 Analisis Kompleksitas

Langkah pertama dari algoritma memiliki $T(360)$ sehingga ia punya kompleksitas $O(1)$.

Langkah kedua memiliki beberapa pendekatan berbeda yang walaupun kompleksitas O-nya sama, memiliki T yang cukup jauh berbeda karena besarnya konstan di depan n dengan n banyaknya GameObject.

Pendekatan pertama adalah sebagai berikut :

1. Iterasi untuk tiap derajat
2. Iterasi untuk tiap GameObject
3. Cek apakah akan terjadi tabrakan menggunakan sebuah algoritma yang memiliki kompleksitas $O(1)$ (pengecekan tabrakan menggunakan rumus jarak sebuah titik ke garis dan beberapa ekspresi aritmatik)

Maka waktu aktualnya adalah $T(360n)$ karena jumlah derajatnya tetap yaitu selalu 360. Akibatnya, langkah kedua dengan pendekatan pertama ini memiliki kompleksitas $O(n)$.

Walaupun langkah pertama tampak memiliki kompleksitas yang tidak begitu buruk, perlu diingat bahwa mayoritas dari kalkulasi yang ia lakukan adalah sia-sia. Sebuah objek yang memiliki luas tak hingga dan jarak mendekati nol ke *bot* sekalipun hanya akan menabrak bot di 180 dari 360 arah trajektori yang ada. Kebanyakan objek yang ada pun juga cukup jauh dari memenuhi ½ jumlah arah trayektori seperti yang diberikan di kasus ekstrim tersebut.

Pendekatan kedua memangkas secara substansial komputasi yang percuma di pendekatan pertama. Langkahnya adalah sebagai berikut :

1. Kita iterasi GameObject.
2. Cari letaknya dalam derajat dari sudut pandang pemain dan simpan sebagai d.

3. Iterasi derajat d searah jarum jam sebagai derajat k dan tambahkan skor GameObject ke k jika GameObject mengalami tabrakan pada k. Lakukan iterasi sampai GameObject tidak lagi mengalami tabrakan di k.
4. Iterasi derajat d berlawanan jarum jam sebagai derajat r dan tambahkan skor GameObject ke r jika GameObject mengalami tabrakan pada r. Lakukan iterasi sampai GameObject tidak lagi mengalami tabrakan di r.

Misalkan objek rata-rata mengalami tabrakan dengan c buah derajat yang ada, Maka pendekatan kedua ini memiliki waktu aktual sebesar $T(cn)$. Walaupun akhirnya kompleksitas waktunya masih $O(n)$, cukup intuitif untuk berpikir bahwa $c \ll 360$ sehingga pendekatan kedua ini menghemat banyak komputasi.

3.4.2 Analisis Efektivitas

Karena GGA ini hanya dibuat sampai mengatur pergerakan saja, maka analisis efektivitas yang dilakukan hanya akan berfokus pada apakah ia mampu memberikan pergerakan yang masuk akal.

Misalkan saja (walau tidak mungkin) *bot* ada di *map* sendirian. GGA ini tidak akan efektif dalam menuju ke arah yang memiliki banyak makanan karena ia tidak memilih derajat dengan skor yang paling tinggi, melainkan derajat di tengah interval.

Meski begitu, GGA ini akan cukup pintar untuk memilih "jalan keluar" terbaik dalam kasus ketika ada beberapa *player* lain yang lebih besar darinya atau awan gas atau *asteroid field* mengelilinginya. Hal ini karena ia mampu mendapatkan celah paling besar di sekelilingnya dan pergi lewat celah tersebut.

Akan tetapi, jika skor semuanya adalah negatif maka *bot* ini akan secara praktik *short-circuited* dan gagal untuk memilih arah apapun.

Alasan besar kenapa GGA ini juga tidak jadi diimplementasikan selain alasan yang telah disebutkan tadi adalah seringnya terjadi ‘osilasi’ pada *bot*.

3.5 GGA Willy-Radit

GGA yang dibuat oleh Willy dan Radit sangat dekat ke GGAP KD. KD memang tidak bisa meng-*handle* banyak kondisi sekaligus, tetapi merupakan MVP yang bagus. Sehingga dalam kasus ketika sudah mepet dengan *deadline* GGA ini masih dapat memberikan hasil yang cukup.

Cara kerja GGA ini terdapat 7 prioritas:

1. Jika *bot* berada dekat dengan perbatasan, maka bergerak ke titik tengah peta
2. Jika ukuran *bot* cukup besar dan belum ada teleporter, maka tembakkan teleporter ke arah musuh.
3. Jika sudah ada teleporter dan teleporter berada di dekat *player* yang besar, maka jangan lakukan teleportasi. Tetapi, jika teleporter berada di dekat *player* yang kecil, maka lakukan teleportasi.
4. Jika ada torpedo yang dekat dan mengarah ke diri kita, maka aktifkan pelindung jika ukuran *bot* terlalu besar untuk bisa kabur dengan mudah. Tetapi, jika ukuran *bot* masih cukup kecil sehingga masih cepat maka *bot* akan menghindar dengan berbelok sejauh 90 derajat.

5. Jika bot dekat dengan musuh dan bot masih punya size untuk menembak maka bot akan menembak. Jika bot tidak punya size untuk menembak maka bot akan melarikan diri.
6. Akan ada validasi kedua untuk border, jika bot sudah berada di daerah borde maka bot akan mengarah ke tengah.
7. Jika bot berada di sekitar gass cloud maka bot akan berbelok 120 derajat.
8. Jika bot berada di sekitar asteroid field maka bot akan berbelok 120 derajat.
9. Jika semua kondisi sebelumnya tidak terpenuhi maka bot akan ,mencari makanan yang terdekat dengannya.

3.5.1 Analisis Kompleksitas

Banyak sekali penyaringan dan pemrosesan daftar GameObject yang dilakukan oleh *bot* seiring GGA berjalan melakukan pengecekan kondisi. Namun, masing-masing hanya memiliki kompleksitas waktu berupa $O(n)$. Sehingga, GGA ini secara total memiliki kompleksitas $O(n)$ pula.

3.5.2 Analisis Efektivitas

Terdapat banyak *edge cases* yang tidak mampu untuk di-*handle* oleh GGA ini karena memang itulah kelemahan dari GGAP DK yang dipakai. Misalkan, ada *player* yang mengejar *bot* sampai ke perbatasan, *bot* akan tetap memilih untuk pergi mengarah ke tengah meskipun itu berarti mati dimakan *player* yang lebih besar. Masih banyak kasus-kasus khusus yang tidak di-*cover* oleh GGA ini.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Greedy Algorithm Dalam Pseudo Code

```
procedure computeNextPlayerAction(var playerAction: PlayerAction);
var
foodNotNearEdgeList,     playerList,     gasCloudList,     asteroidFieldList,
teleporterList,
torpedoSalvoList: TList<GameObject>;
foodFarEdgeDist,   gasDist,   asteDist,   botToMid,   nearestPlayer,   botsize,
enemySize: Double;
ticker,    headGas,    headAster,    headPlayer,    headMid,    foodFarEdgeHead,
teleportHeading: Integer;
done, shouldTeleport: Boolean;
dummy: GameObject;
begin
playerAction.action := FORWARD;
playerAction.heading := Random(360);
if gameState.getGameObjects().Count > 0 then
begin
foodNotNearEdgeList := TList<GameObject>.Create;
playerList := TList<GameObject>.Create;
gasCloudList := TList<GameObject>.Create;
asteroidFieldList := TList<GameObject>.Create;
teleporterList := TList<GameObject>.Create;
torpedoSalvoList := TList<GameObject>.Create;
gameState.getGameObjects().Where(
    function(item: GameObject): Boolean
    begin
              Result   :=   (item.getGameObjectType()   =   ObjectTypes.FOOD)   or
(item.getGameObjectType() = ObjectTypes.SUPERFOOD)
          and  (Double(gameState.getWorld().radius)  -  getDistanceBetween(item,
gameState.getWorld())
      >= 1.2 * Double(bot.size));
    end)
    .OrderBy(
      function(item: GameObject): Double
      begin
        Result := getDistanceBetween(bot, item);
      end)
    .ToList<GameObject>(foodNotNearEdgeList);

  gameState.getPlayerGameObjects().Where(
    function(item: GameObject): Boolean
    begin
      Result := item.getId() <> bot.getId();
    end)
    .OrderBy(
      function(item: GameObject): Double
      begin
        Result := getDistanceBetween(bot, item);
```

```
    end)
  .ToList<GameObject>(playerList);

gasCloudList := gameState.getGameObjects().Where(
  function(item: GameObject): Boolean
  begin
    Result := item.getGameObjectType() = ObjectTypes.GASCLOUD;
  end)
  .OrderBy(
    function(item: GameObject): Double
    begin
      Result := getDistanceBetween(bot, item);
    end)
  .ToList<GameObject>;

asteroidFieldList := gameState.getGameObjects().Where(
  function(item: GameObject): Boolean
  begin
    Result := item.getGameObjectType() = ObjectTypes.ASTEROIDFIELD;
  end)
  .OrderBy(
    function(item: GameObject): Double
    begin
      Result := getDistanceBetween(bot, item);
    end)
  .ToList<GameObject>;

teleporterList := gameState.getGameObjects().Where(
  function(item: GameObject): Boolean
  begin
    Result := item.getGameObjectType() = ObjectTypes.TELEPORTER;
  end)
  .OrderBy(
    function(item: GameObject): Double
    begin
      Result := getDistanceBetween(bot, item);
    end)
  .ToList<GameObject>;

torpedoSalvoList := gameState.getGameObjects().Where(
  function(item: GameObject): Boolean
  begin
    Result := item.getGameObjectType() = ObjectTypes.TORPEDOSALVO;
  end)
  .OrderBy(
    function(item: GameObject): Double
    begin
      Result := getDistanceBetween(bot, item);
    end)
  .ToList<GameObject>;

{print tick}
ticker := gameState.getWorld().getCurrentTick();
WriteLn('Tick: ', ticker);
```

```
  { Mendapatkan nilai radius dari map saat ini }
  var bordeRadius := gameState.world.getRadius();

  { Mendapatkan posisi tengah dari map }
  var mid: Position := new Position(0, 0);

  { Membuat game object yang berposisi di tengah map }
  var dummy: GameObject := new GameObject(nil, nil, nil, nil, mid, nil);

  { Mendapatkan jarak bot ke tengah }
  var botToMid := getDistanceBetween(Self.bot, dummy);

  { Mendapatkan jarak player terdekat }
  var nearestPlayer := getDistanceBetween(Self.bot, playerList[0]);

  { Melakukan pengecekan dan mendapatkan jarak dan heading pada gas cloud }
  var gasDist: Real;
  var headGas: Integer;
  if (gasCloudList.Count > 0) then
  begin
      gasDist := getDistanceBetween(Self.bot, gasCloudList[0]);
      headGas := getHeadingBetween(gasCloudList[0]);
  end
  else
  begin
      gasDist := 99999;
      headGas := -1;
  end;

  { Melakukan pengecekan dan mendapatkan jarak dan heading pada asteroid field
}
  var asteDist: Real;
  var headAster: Integer;
  if (asteroidFieldList.Count > 0) then
  begin
      asteDist := getDistanceBetween(Self.bot, asteroidFieldList[0]);
      headAster := getHeadingBetween(asteroidFieldList[0]);
  end
  else
  begin
      asteDist := 99999;
      headAster := -1;
  end;

  { Melakukan printing info ke layar }
  var headPlayer := getHeadingBetween(playerList[0]);
  writeln('rd: ', gameState.getWorld().getRadius(), ' Gs: ', gasDist, ' As: ',
asteDist, ' Pl: ', nearestPlayer, ' Md: ', botToMid, ' sz: ', Self.bot.size);
  var headMid := getHeadingBetween(dummy);

  { Melakukan pengecekan dan mendapatkan jarak dan heading pada makanan }
  var foodFarEdgeDist: Real := 99999;
  var foodFarEdgeHead: Integer := -1;
  if (foodNotNearEdgeList.Count > 0) then
  begin
```

```
      foodFarEdgeDist := getDistanceBetween(Self.bot, foodNotNearEdgeList[0]);
      foodFarEdgeHead := getHeadingBetween(foodNotNearEdgeList[0]);
  end
  else
  begin
      foodFarEdgeDist := 99999;
  end;
  var teleportHeading: Integer := -1;

  { Mendapatkan ukuran dari bot }
  var botsize := Self.bot.size;
  var test := 0;

  { Mendapatkan ukuran dari musuh terdekat }
  var enemySize := playerList[0].size;

  { Membuat boolean untuk memastikan bot hanya melakukan satu aksi }
  var done: Boolean := False;

  { Membuat boolean untuk pengecekan apakah bot harus teleport }
  var shouldTeleport: Boolean := False;

  if (bordeRadius < botToMid + 1.2 * botsize) then
  begin
      { Bot menghindar dari border dan menuju ke tengah }
      writeln('Bot is running from border!');
      playerAction.heading := headMid;
      playerAction.action := PlayerActions.FORWARD;
      done := True;
  end;

  { Membuat kondisi penembakan teleported }
if (botsize > 90) and (not haveTeleporter) and (not done) then
begin
    writeln('Bot sent teleporter!');
    playerAction.heading := getHeadingBetween(playerList[0]);
    playerAction.action := PlayerActions.FIRETELEPORT;
    teleportHeading := getHeadingBetween(playerList[0]);
    haveTeleporter := true;
    done := true;
end
else if (haveTeleporter) and (not done) then
begin
    var j, k: integer;
    var shouldTeleport: boolean := true;
       {Mendapatkan kondisi teleporter, jika berbahaya maka bot tidak akan
teleport}
    for j := 0 to teleporterList.Count - 1 do
    begin
        k := 0;
        while (k < playerList.Count) and (not done) do
        begin
               if (getDistanceBetween(teleporterList[j], playerList[k]) < 100)
and (playerList[k].size > botsize) and (teleporterList[j].currentHeading =
teleportHeading) then
```

```
            begin
                writeln('Bot is NOT going for TELEPORT!');
                shouldTeleport := false;
                break;
            end
            else
            begin
                k := k + 1;
            end;
        end;
    end;
    {Mendapatkan kondisi teleporter apabila bot aman untuk teleport}
    for j := 0 to teleporterList.Count - 1 do
    begin
        k := 0;
        while (k < playerList.Count) and (not done) do
        begin
               if (getDistanceBetween(teleporterList[j], playerList[k]) < 100)
and (playerList[k].size < botsize) and (teleporterList[j].currentHeading =
teleportHeading) then
            begin
                writeln('Bot is going for TELEPORT!');
                playerAction.heading := getHeadingBetween(playerList[k]);
                playerAction.action := PlayerActions.TELEPORT;
                haveTeleporter := false;
                done := true;
            end
            else
            begin
                k := k + 1;
            end;
        end;
    end;
end
else if (teleporterList.Count > 0) then
begin
    for var i := 0 to teleporterList.Count - 1 do
    begin
                if (teleporterList[i].currentHeading = teleportHeading) and
(getDistanceBetween(bot,                  teleporterList[i])                  >
(this.gameState.getWorld().getRadius() * 1.3)) then
        begin
            haveTeleporter := false;
        end;
    end;
end
else
begin
    {do nothing}
end;

{Melakukan pengecekan apakah bot harus menghindari torpedo}
if (not done) and (torpedoSalvoList.size() > 0) then
begin
    salvoDist := getDistanceBetween(this.bot, torpedoSalvoList.get(0));
```

```
    headSalvo := getHeadingBetween(torpedoSalvoList.get(0));
    writeln('Salvo: ', salvoDist, ' Head: ', headSalvo);
    if (salvoDist < 100 + 1.2 * botsize) and (botsize > 100) then
    begin
            { Jika bot berukuran besar dan ada peluru mendekat maka bot akan
mengaktifkan shield }
        writeln('Bot is activating shield!');
        playerAction.heading := foodFarEdgeHead;
        playerAction.action := PlayerActions.ACTIVATESHIELD;
        done := true;
    end
    else if (salvoDist < 100 + 1.2 * botsize) then
    begin
            { Jika bot berukuran kecil dan ada peluru mendekat maka bot akan
melarikan diri dari peluru }
        writeln('Bot is running from salvo!');
        playerAction.heading := (headSalvo + 90) mod 360;
        playerAction.action := PlayerActions.FORWARD;
        done := true;
    end
    else
    begin
        { do nothing }
    end;
end;

{ Jika kondisi teleporter atau defending belum memberikan aksi maka akan masuk
ke kondisional di bawah }
if not done then
begin
    if (nearestPlayer < 200) and (botsize > 20) then
    begin
        { Bot akan menembak dengan kondisi tertentu }
        writeln('Bot is attacking!');
        playerAction.action := PlayerActions.FIRETORPEDOES;
        playerAction.heading := headPlayer;
    end
     else if (nearestPlayer < 150 + 2 * enemySize) and (this.bot.getSize() <
playerList.get(0).getSize()) then
    begin
        { Bot melarikan diri dari musuh yang terlalu besar }
        writeln('Bot is running for his life!');
        playerAction.action := PlayerActions.FORWARD;
        playerAction.heading := (headPlayer + 120) mod 360;
    end
    else if (bordeRadius < botToMid + 1.3 * botsize) then
    begin
        { Bot menghindar dari border dan menuju ke tengah }
        writeln('Bot is running from border!');
        playerAction.heading := headMid;
        playerAction.action := PlayerActions.FORWARD;
    end
    else if (gasDist < 100 + 1.2 * botsize) then
    begin
        { Bot berbelok menghindari gas cloud }
```

```
        writeln('Bot is running from gass');
        playerAction.heading := (headGas + 120) mod 360;
        playerAction.action := PlayerActions.FORWARD;
    end
    else if (asteDist < 100 + 1.2 * botsize) then
    begin
        { Bot berbelok menghindari asteroid }
        writeln('Bot is running from asteroid');
        playerAction.heading := (headAster + 120) mod 360;
        playerAction.action := PlayerActions.FORWARD;
    end
    else
    begin
            { Jika tidak ada kondisi di atas maka bot akan memakan makanan
terdekat }
        writeln('Bot is now eating');
        playerAction.action := PlayerActions.FORWARD;
        playerAction.heading := foodFarEdge
    end
    this.PlayerAction := playerAction
    end
end
```

4.2 Struktur Data

a. Package Enums

i. Modul ObjectTypes

Modul ini berisi tipe-tipe objek yang terdapat pada game seperti player, food, wormhole, gascloud, asteroidfield, torpedosalvo, superfood, supernovapickup, supernovabomb, teleporter, dan shield.

ii. Modul PlayerActions

Modul ini berisi tipe-tipe action yang bisa dilakukan oleh player seperti forward, stop, startafterburner, stopafterburner, firetorpedoes, firesupernova, detonatesupernova, fireteleport, teleport, activateshield.

b. Package models

i. GameObject

Game object berfungsi untuk menginisiasi sebuah game object dengan beberapa atribut seperti ID, size, speed, currentHeading, position, gameobjectType, dan torpedoSalvoCount.

ii. GameState

Game state berfungsi untuk membuat world dan game object yang ada di dalamnya.

iii. GameStateDto

GamestateDto memiliki fungsi yang mirip dengan gamestate.

iv. PlayerAction

PlayerAction digunakan untuk menentukan dan mendapatkan atribut yang berkaitan dengan player action seperti heading.

v. Position

Position berfungsi untuk menentukan posisi sebuah objek melalui kordinat x dan y pada peta.

vi. World

World berguna untuk mendapatkan atribut yang berkaitan dengan world seperti radius.

c. Package service

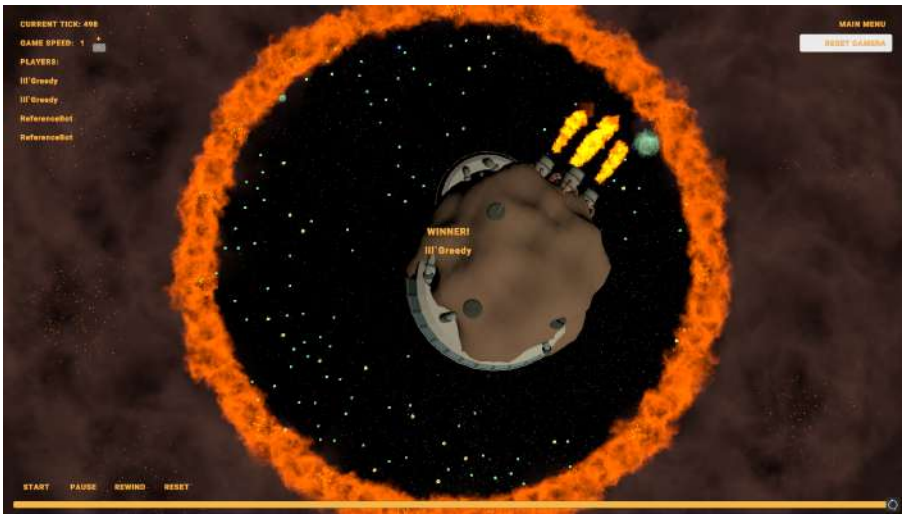
Package ini berisi file BotService dimana bot akan mengkalkulasi semua hal dan menentukan apa yang akan bot tersebut lakukan pada tick berikutnya.

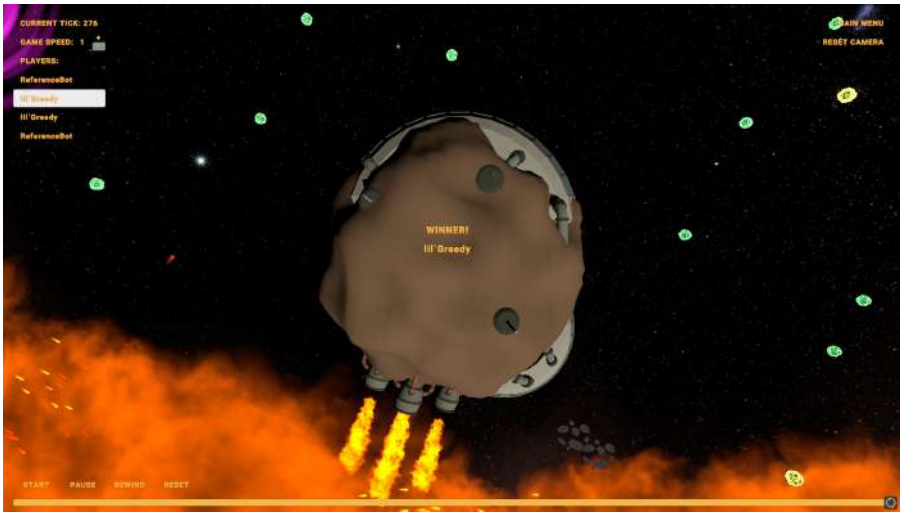
d. main

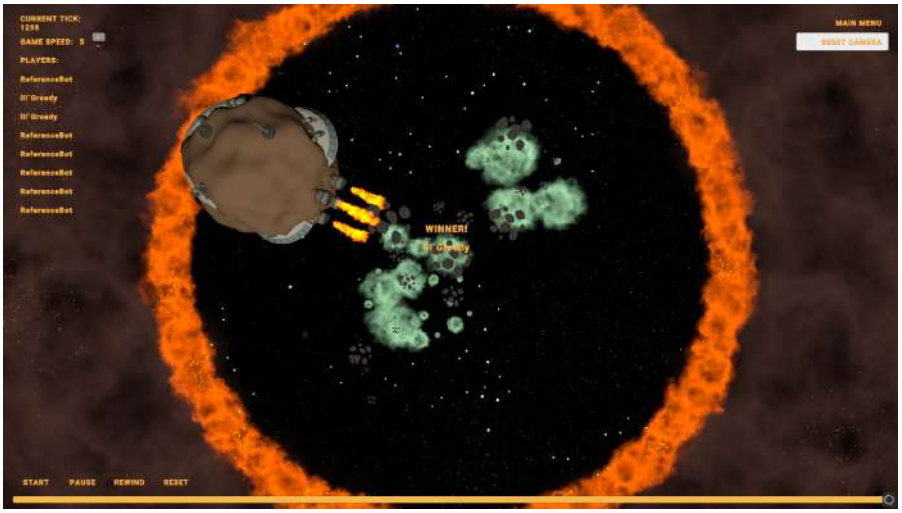
Main disini berfungsi sebagai penghubung sebuah bot dengan runner dari galaxio.

4.3 Pengujian

a. Pengujian hasil yang menang

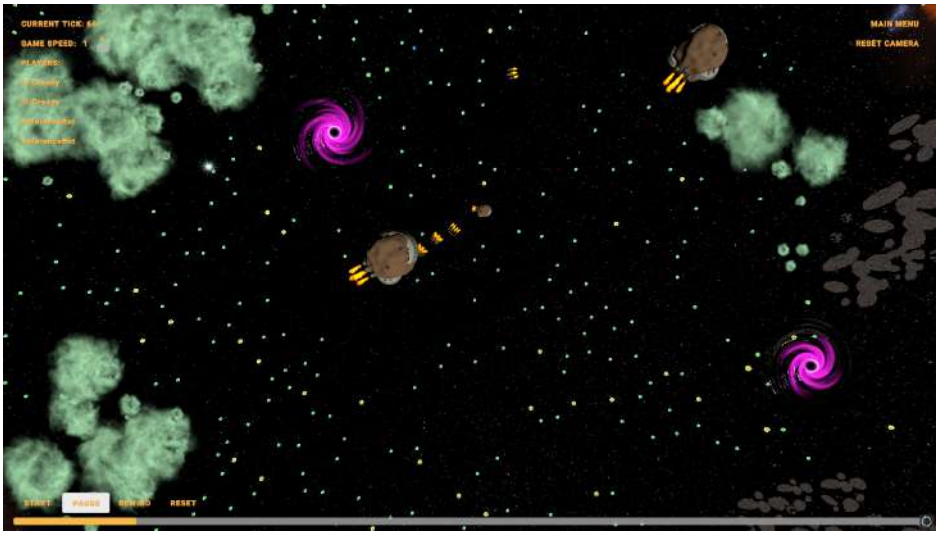


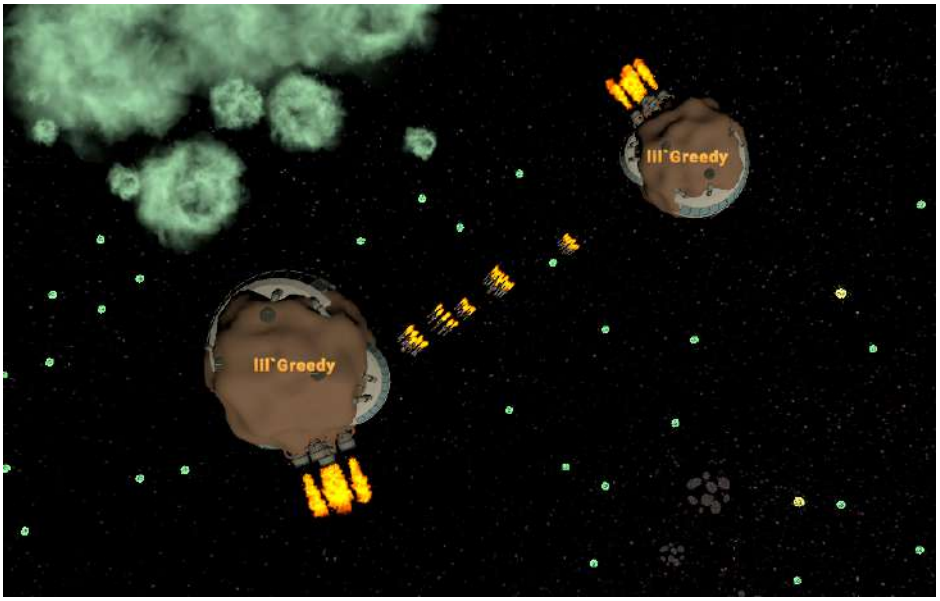




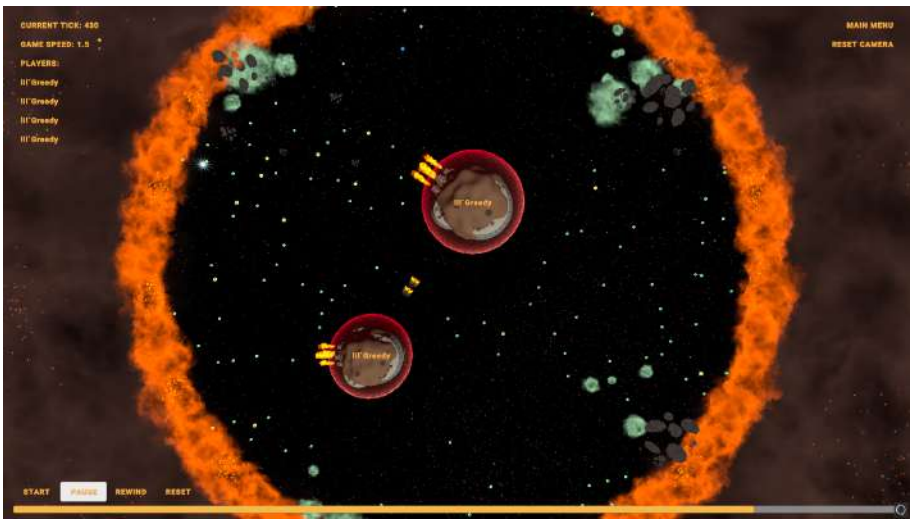
b. Pengujian penembakan teleporter

c. Pengujian menembak musuh



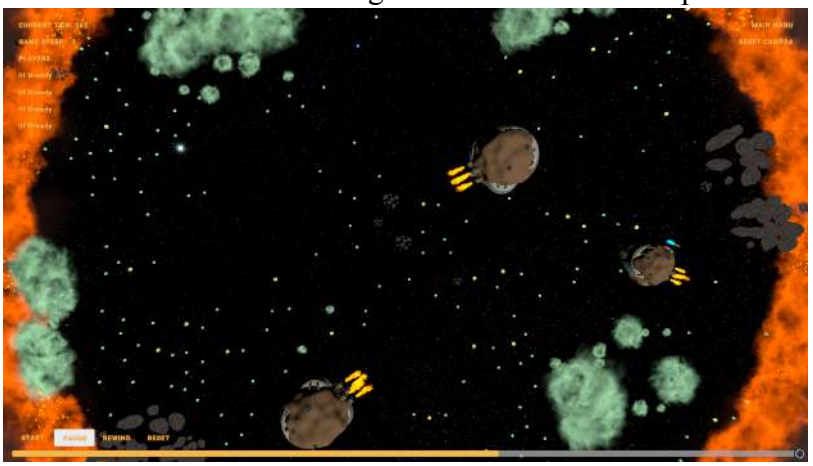


d. Pengujian pengaktifan shield bot

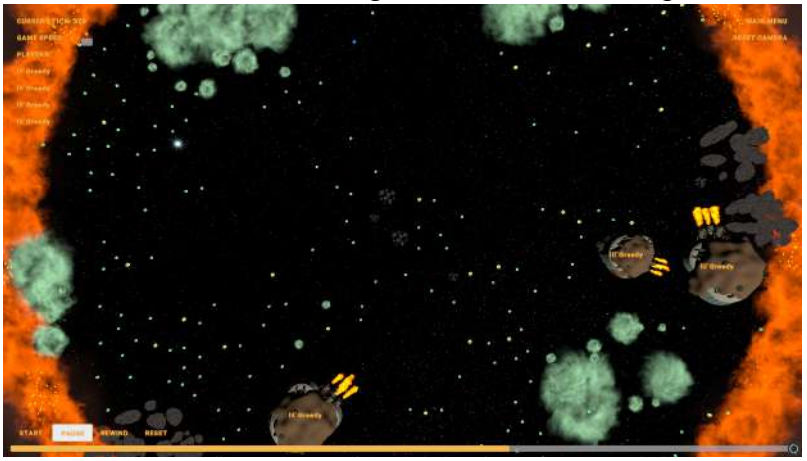


e. Pengujian berpindah dengan teleport

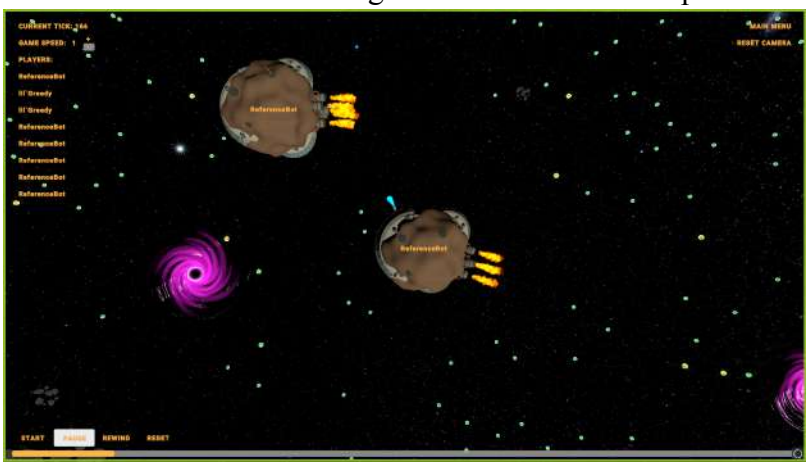
Gambar di bawah ini adalah gambar sebelum bot teleport



Gambar di bawah ini adalah gambar setelah bot teleport



Gambar di bawah ini adalah gambar sebelum bot teleport



Gambar di bawah ini adalah gambar setelah bot teleport

BAB 5

KESIMPULAN DAN SARAN

5.1 Kesimpulan

1. Bagi orang yang masih pemula akan Java, bahasa ini akan terasa *unnecessarily verbose*. Tetapi setelah menghabiskan waktu dengan bahasa ini, Java menjadi intuitif dan mudah dipakai karena sifatnya yang *strongly-typed* dan aspek lain dari bahasa ini seperti semua error yang mungkin harus selalu di-*catch* sehingga program menjadi lebih stabil dan tidak rawan kesalahan.
2. Dua algoritma yang implementasi dan cara kerja aktualnya sama sekali berbeda bisa jadi berasal dari teknik desain algoritma yang sama. Dari 56 kelompok tugas besar 1 Stima yang ada, algoritmanya semuanya berbeda tetapi prinsip dasarnya sama, yaitu GA.
3. *Short-sightedness* dari GA benar-benar terasa dampak negatifnya. Persoalan seperti *bot* yang bolak-balik di tempat sejatinya *trivial* untuk dipecahkan jika kita tidak menggunakan GA. Tetapi, karena kita dibatasi untuk menggunakan GA, maka permasalahan seperti itu bisa jadi menuntut desain ulang dari strategi *greedy* yang dipakai.

5.2 Saran

1. Program yang ditulis, meskipun seratus persen fungsional, masih dapat dirapikan baik dari penyederhanaan logika dari fungsi-fungsi di dalamnya maupun penulisan dokumentasi serta *comments* yang lebih deskriptif.

2. Program ini dapat dirapikan dan ditingkatkan modularitasnya melalui cara seperti mengekstrak konstanta-konstanta yang tersebar di dalam algoritmanya ke dalam *file* terpisah atau memecah bagian-bagian diskrit dari algoritmanya menjadi fungsi maupun kelas khusus yang terpisah.

DAFTAR PUSTAKA

1. https://github.com/EntelectChallenge/2021-Galaxio diakses pada 2 Februari 2023

LAMPIRAN

Tautan repositori Github

Link video singkat