

Tugas Besar 1 IF2211 Strategi Algoritma

Pemanfaatan Algoritma Greedy dalam Aplikasi Permainan

“Galaxio”



Disusun oleh :
Kelompok sukatumak

13521052 Melvin Kent Jonathan

13521056 Daniel Egiant Sitanggang

13521079 Hobert Anthony Jonatan

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

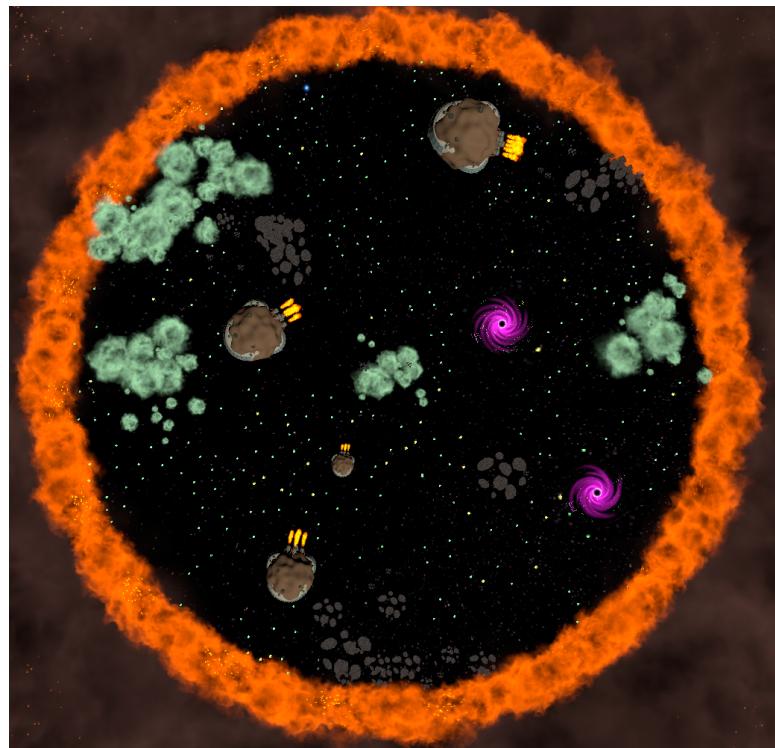
DAFTAR ISI	1
Bab 1 Deskripsi Tugas	3
Bab 2 Landasan Teori	6
2.1 Algoritma Greedy	6
2.1.1 Permasalahan yang Ditangani oleh Algoritma Greedy	6
2.1.2 Elemen-elemen Algoritma Greedy	6
2.1.3 Pembuktian Optimalitas	7
2.2 Cara Kerja Program	7
2.2.1 Game Engine	7
2.2.2 Komponen untuk Menjalankan Game Galaxio	8
2.2.3 Struktur Game Engine	8
Bab 3 Aplikasi Strategi Greedy	11
3.1 Eksplorasi Alternatif Solusi Algoritma Greedy	11
3.1.1 Algoritma Greedy by Nearest Food	11
3.1.2 Algoritma Greedy by Densest Food Area	12
3.1.3 Algoritma Greedy by Safest Place	14
3.1.4 Algoritma Greedy by Safest Trajectory	16
3.1.5 Pemetaan Elemen Algoritma Greedy by Feeding on Smaller Bot	17
3.1.6 Algoritma Greedy by Destroying Opponents	19
3.1.7 Algoritma Greedy by Nearest-Safe Food	20
3.2 Strategi Greedy dengan Belajar dari Hukum Alam - Gravitasi	22
3.3 Strategi Greedy yang Dipilih	23
Bab 4 Implementasi dan Pengujian	25
4.1 Implementasi Algoritma Greedy dalam PseudoCode	25
4.2 Pengujian Program	37
4.2.1 Bot Menghindari Gas Cloud	37
4.2.2 Bot Menghindari Boundary Map	38
4.2.3 Bot Menghindari Lawan yang Lebih Besar	39
4.2.4 Bot Meluncurkan Torpedo ke Lawan yang Lebih Besar	40
4.2.5 Bot Meluncurkan Torpedo ke Lawan yang Lebih Kecil	41
4.2.6 Bot Menuju Makanan Aman Terdekat	42
4.2.7 Bot Mengejar dan Memangsa Lawan yang Lebih Kecil	43
4.3 Ketercapaian Algoritma	43

Bab 5 Kesimpulan dan Saran	45
5.1 Kesimpulan	45
5.2 Saran	45
DAFTAR PUSTAKA	46
LINK PENTING	47

Bab 1

Deskripsi Tugas

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



Gambar 1. Ilustrasi permainan *Galaxio*

Pada tugas besar pertama Strategi Algoritma ini, gunakanlah sebuah *game engine* yang mengimplementasikan permainan *Galaxio*. *Game engine* dapat diperoleh pada laman berikut:

<https://github.com/EntelectChallenge/2021-Galaxio>

Tugas mahasiswa adalah mengimplementasikan bot kapal dalam permainan *Galaxio* dengan menggunakan **strategi greedy** untuk memenangkan permainan. Untuk mengimplementasikan bot tersebut, mahasiswa disarankan melanjutkan program yang terdapat pada *starter-bots* di dalam *starter-pack* pada laman berikut ini:

<https://github.com/EntelectChallenge/2021-Galaxio/releases/tag/2021.3.2>

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

1. Peta permainan berbentuk kartesius yang memiliki arah positif dan negatif. Peta hanya menangani angka bulat. Kapal hanya bisa berada di *integer* x,y yang ada di peta. Pusat peta adalah 0,0 dan ujung dari peta merupakan radius. Jumlah ronde maximum pada game sama dengan ukuran radius. Pada peta, akan terdapat 5 objek, yaitu *Players*, *Food*, *Wormholes*, *Gas Clouds*, *Asteroid Fields*. Ukuran peta akan mengecil seiring batasan peta mengecil.
2. Kecepatan kapal dilambangkan dengan x. Kecepatan kapal akan dimulai dengan kecepatan 20 dan berkurang setiap ukuran kapal bertambah. Ukuran (radius) kapal akan dimulai dengan ukuran 10. *Heading* dari kapal dapat bergerak antar 0 hingga 359 derajat. Efek *afterburner* akan meningkatkan kecepatan kapal dengan faktor 2, tetapi mengecilkan ukuran kapal sebanyak 1 setiap tick. Kemudian kapal akan menerima 1 salvo charge setiap 10 tick. Setiap kapal hanya dapat menampung 5 salvo charge. Penembakan slavo torpedo (ukuran 10) mengurangkan ukuran kapal sebanyak 5.
3. Setiap objek pada lintasan punya koordinat x,y dan radius yang mendefinisikan ukuran dan bentuknya. *Food* akan disebarluaskan pada peta dengan ukuran 3 dan dapat dikonsumsi oleh kapal *player*. Apabila *player* mengkonsumsi *Food*, maka *Player* akan bertambah ukuran yang sama dengan *Food*. *Food* memiliki peluang untuk berubah menjadi *Super Food*. Apabila *Super Food* dikonsumsi maka setiap makan *Food*, efeknya akan 2 kali dari *Food* yang dikonsumsi. Efek dari *Super Food* bertahan selama 5 tick.
4. *Wormhole* ada secara berpasangan dan memperbolehkan kapal dari *player* untuk memasukinya dan keluar di pasangan satu lagi. *Wormhole* akan bertambah besar setiap tick game hingga ukuran maximum. Ketika *Wormhole* dilewati, maka *wormhole* akan mengecil sebanyak setengah dari ukuran kapal yang melewatkannya 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 10.
9. Ketika kapal *player* bertabrakan dengan kapal lain, maka kapal yang lebih besar akan mengonsumsi oleh kapal yang lebih kecil sebanyak 50% dari ukuran kapal yang lebih besar hingga ukuran maximum dari ukuran kapal yang lebih kecil. Hasil dari tabrakan akan mengarahkan kedua dari kapal tersebut lawan arah.
10. Terdapat beberapa *command* yang dapat dilakukan oleh *player*. Setiap tick, *player* hanya dapat memberikan satu *command*. Untuk daftar *commands* yang tersedia, bisa merujuk ke tautan [panduan](#) di spesifikasi tugas
11. Setiap *player* akan memiliki score yang hanya dapat dilihat jika permainan berakhir. Score ini digunakan saat kasus *tie breaking* (semua kapal mati). Jika mengonsumsi kapal *player* lain, maka score bertambah 10, jika mengonsumsi *food* atau melewati wormhole, maka score bertambah 1. Pemenang permainan adalah kapal yang bertahan paling terakhir dan apabila *tie breaker* maka pemenang adalah kapal dengan score tertinggi.

Bab 2

Landasan Teori

2.1 Algoritma Greedy

Persoalan optimasi menjadi salah satu konsiderasi ataupun target dari pembuatan sebuah program. Sebuah program yang optimal diharapkan dapat memecahkan sebuah persoalan dengan 2 jenis konsiderasi, yakni maksimasi (*maximization*) dan minimasi (*minimization*). Kedua macam konsiderasi tersebut dapat saling bebas (independen) maupun saling memengaruhi (dependen). Terdapat berbagai jenis algoritma yang dapat digunakan untuk memecahkan masalah optimasi ini, salah satunya adalah algoritma *greedy*.

Algoritma *greedy* merupakan sebuah strategi untuk memecahkan persoalan optimasi yang menangani permasalahan secara langkah per langkah dengan memilih solusi optimum lokal (local optimum) pada setiap iterasi yang diharapkan dapat mengarah ke solusi optimum global (global optimum). Cara kerja algoritma *greedy* dapat disandingkan dengan prinsip *greedy*, yakni “*take what you can get now!*” .

Suatu persoalan tepat untuk diselesaikan dengan algoritma *greedy* apabila persoalan tersebut memiliki dua sifat berikut :

- Solusi optimal dari persoalan dapat ditentukan dari solusi optimal sub persoalan tersebut
- Pada setiap sub persoalan (langkah untuk menyelesaikan persoalan besar yang awal), terdapat suatu langkah yang dapat dilakukan yang mana langkah tersebut menghasilkan solusi optimal pada sub persoalan tersebut. Dalam terminologi algoritma *greedy*, langkah ini disebut sebagai *greedy choice*.

2.1.1 Permasalahan yang Ditangani oleh Algoritma *Greedy*

Sebuah permasalahan dapat terbagi ke dalam langkah-langkah dan pada setiap langkah terdapat banyak opsi yang perlu dievaluasi. Algoritma *greedy* pada umumnya menangani permasalahan yang tidak memberikan kesempatan kepada kita untuk kembali ke langkah sebelumnya. Oleh karena itu, pada setiap langkah kita harus dapat mengambil keputusan yang terbaik dalam membuat pilihan.

2.1.2 Elemen-elemen Algoritma *Greedy*

- **Himpunan Kandidat (C)**

Himpunan kandidat merupakan himpunan yang berisi kandidat opsi yang akan dipilih pada setiap langkah.

- **Himpunan Solusi (S)**

Himpunan solusi merupakan himpunan yang berisi kandidat yang telah dipilih oleh program.

- **Fungsi Solusi (*solution function*)**

Fungsi solusi merupakan fungsi yang menentukan apakah himpunan kandidat yang dipilih telah menghasilkan solusi.

- **Fungsi Seleksi (*selection function*)**

Fungsi seleksi merupakan fungsi yang berfungsi memilih kandidat berdasarkan strategi *greedy* tertentu yang bersifat heuristik.

- **Fungsi Kelayakan (*feasibility function*)**

Fungsi kelayakan merupakan fungsi yang memeriksa apakah kandidat yang dipilih dapat dimasukan ke dalam himpunan solusi (layak atau tidak).

- **Fungsi Objektif (*objective function*)**

Fungsi objektif merupakan fungsi yang memaksimumkan atau meminimumkan.

2.1.3 Pembuktian Optimalitas

Pembuktian optimalitas dari algoritma *greedy* pada suatu persoalan dapat dilakukan secara matematis. Namun, pembuktian secara matematis ini menimbulkan tantangan sendiri. Di sisi lain, membuktikan bahwa algoritma *greedy* tidak optimal dapat dilakukan dengan mencari *counterexample*, yakni sebuah kasus yang menunjukkan bahwa solusi optimal tidak diperoleh.

2.2 Cara Kerja Program

2.2.1 Game Engine

Game engine adalah perangkat lunak atau *platform* yang digunakan untuk membangun, merancang, dan mengembangkan *video game*. *Game engine* ini menyediakan sejumlah fitur dan fungsionalitas yang dibutuhkan oleh pengembang *game* untuk membuat *game*, seperti grafis 3D, animasi, fisika, suara, *script*, dll.

Dengan menggunakan *game engine*, pengembang dapat menghemat waktu dan biaya dalam mengembangkan *game*, karena tidak perlu membangun fitur-fitur dasar dari awal. Sebagai gantinya, mereka dapat fokus pada pengembangan *game* yang lebih spesifik dan kreatif, seperti *gameplay*, cerita, dan fitur-fitur unik lainnya. Beberapa contoh *game engine* populer di antaranya Unity, Unreal Engine, CryEngine, dan GameMaker. Permainan Galaxio sendiri dibangun menggunakan *game engine* Unity.

2.2.2 Komponen untuk Menjalankan Game Galaxio

Dalam permainan Galaxio, terdapat beberapa komponen yang diperlukan untuk menjalankan permainan, antara lain

1. Game engine

Game engine berperan sebagai *platform* untuk *bot* bermain. *Game engine* akan meneruskan perintah yang dikeluarkan oleh *bot* ke dalam *game* untuk diproses, tentunya dengan memerhatikan aturan dalam *game* yang telah didefinisikan sebelumnya.

2. Game Runner

Game runner berperan sebagai perangkat lunak yang menjalankan atau memulai pertandingan antar pemain dalam *game* Galaxio dengan konfigurasi yang telah ditentukan.

3. Logger

Logger berperan sebagai perangkat lunak yang menyimpan *log* (catatan) keberjalanannya, yaitu seluruh aktivitas, kondisi *map*, serta seluruh *command* yang dilakukan oleh setiap *bot* dalam suatu pertandingan dalam *game* Galaxio.

4. Reference bot

Reference bot adalah *bot* yang telah disediakan oleh pembuat Galaxio sebagai lawan untuk mengetes kemampuan *bot* yang kita kembangkan.

5. Starter bot

Starter bot adalah kerangka *bot* atau *template* yang dapat kita gunakan untuk mulai menulis *code* untuk *bot* yang akan kita kembangkan. *Starter bot* sendiri telah memiliki logika sederhana untuk memainkan game Galaxio. *Starter bot* tersedia dalam bahasa C++, Java, JavaScript, Kotlin, NETCore, dan Python.

File luaran dari *logger* kemudian dapat kita pakai untuk melakukan visualisasi permainan dengan menggunakan *visualiser* yang telah kita dapatkan saat mengunduh folder starter-pack Galaxio pertama kali. *Visualiser* tersedia dalam tiga Operating System, yaitu Windows, MacOS, dan Linux. Dengan menjalankan *log* permainan dalam *visualiser*, kita dapat melihat visualisasi seluruh pergerakan *bot* dan kondisi permainan dalam sebuah pertandingan *game* Galaxio.

2.2.3 Struktur Game Engine

Untuk mulai membangun *bot* untuk *game* Galaxio, pengembang perlu mengunduh terlebih dahulu starter-pack yang telah disediakan oleh pembuat *game* Galaxio, melalui tautan <https://github.com/EntelectChallenge/2021-Galaxio/releases/tag/2021.3.2>. Pada tugas besar ini, akan dibangun sebuah *bot* yang akan dikembangkan menggunakan bahasa Java, sehingga yang akan dijelaskan dalam dokumen ini hanyalah *bot* terkait dengan Java.

Starter pack berisi beberapa folder dan file, yaitu :

- Folder engine-publish : berisi *game engine* untuk menjalankan permainan.

- Folder logger-publish : berisi perangkat lunak untuk mencatat *log* permainan.
- Folder reference-bot-publish : berisi *reference bot*.
- Folder runner-publish : berisi *game runner* yang telah dijelaskan sebelumnya.
- Folder starter-bots : berisi starter bot dalam berbagai bahasa pemrograman.
- Folder visualiser : berisi perangkat lunak untuk memvisualisasikan permainan berdasarkan masukan berupa *log* permainan.
- Markdown File building-a-bot : berisi *text* yang merupakan panduan untuk membangun *bot* untuk Galaxio *Entelect Challenge* ini.
- File README : berisi *text* yang menjelaskan beberapa hal yang perlu diketahui oleh peserta *Challenge* sebelum mulai mengembangkan *bot*.
- File run.sh : berisi perintah (*shell script*) untuk menjalankan permainan pada sistem operasi berbasis Unix / Linux.

Pada *folder* starter-bot, terdapat *template* untuk memulai pengembangan *bot* dalam berbagai bahasa pemrograman. Pada tugas besar kali ini, *bot* dikembangkan dalam bahasa Java, sehingga kita hanya akan menggunakan *file* dalam *folder* berjudul JavaBot. Pada *folder* JavaBot terdapat *folder* src yang berisi *source code* dari *bot*, lebih spesifiknya terdapat pada *folder* src/main/java, yang berisi :

- Folder Enums
Folder ini berisi file ObjectTypes.java dan PlayerActions.java, kedua file ini berisi enum yang menyatakan entitas yang memiliki nilai konstan dari kelas ObjectTypes dan juga kelas PlayerActions.
- Folder Models
Folder ini berisi beberapa file, antara lain :
 - File GameObject.java yang berisi pendefinisan kelas GameObject atau seluruh *object* yang ada dalam game.
 - File GameState.java yang berisi pendefinisan kelas GameState atau *state* dari suatu saat dalam game.
 - File GameStateDto.java yang berisi pendefinisan kelas GameStateDto untuk menginisialisasi *state* dalam game.
 - File PlayerAction.java yang berisi pendefinisan kelas PlayerAction atau aksi-aksi yang dapat oleh dilakukan oleh seorang *player* dalam setiap tick game.
 - File Position.java yang berisi pendefinisan kelas Position yang merupakan atribut dari seluruh kelas yang menyatakan *object* dalam game.
 - File World.java yang berisi pendefinisan kelas World dalam game yang merupakan kelas pembentuk *map* atau *world* dalam game Galaxio.
- Folder Services
Folder ini berisi file BotService.java yang merupakan file dimana kita akan melakukan pengembangan *bot*, spesifiknya yaitu dikhususkan dalam fungsi computeNextPlayerAction.

- File Main.java

File ini berisi *source code* yang berfungsi untuk menjalankan *bot* kita dan menghubungkan *bot* dengan komponen lain dalam *game*, seperti *Game Engine*, *Game Runner*, dll.

Bab 3

Aplikasi Strategi *Greedy*

3.1 Eksplorasi Alternatif Solusi Algoritma Greedy

3.1.1 Algoritma Greedy by Nearest Food

a. Pemetaan Elemen Algoritma Greedy

Pada permainan Galaxio, tujuan akhir untuk memenangkan permainan dapat diraih dengan menjadi bot dengan ukuran paling besar sehingga tidak dapat dimakan oleh bot lain. Maksimalisasi ukuran bot dengan waktu tercepat sebanding dengan maksimalisasi makanan yang dikonsumsi dalam periode waktu sesingkat-singkatnya. Untuk dapat mengonsumsi makanan secara cepat, lintasan yang ditempuh oleh bot dari satu tempat ke posisi makanan haruslah terpendek. Oleh karena itu, algoritma *greedy by nearest food* membuat bot menuju makanan terdekat agar dapat sesegera mungkin memperbesar ukurannya. Pada opsi algoritma ini, tidak dipertimbangkan faktor lain (bot lawan, gas cloud, dll) selain jarak terdekat ke makanan.

Elemen Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh kemungkinan pergerakan bot ke arah makanan yang tersedia pada peta dalam setiap tick.
Himpunan Solusi	Seluruh opsi pergerakan bot ke arah makanan yang terdekat terhadap bot pada setiap tick.
Fungsi Solusi	Melakukan pengecekan apakah seluruh makanan di dalam map sudah tidak ada lagi. Hal ini karena bot tidak akan berhenti mencari makan hingga tidak ada makanan lagi yang tersisa. Algoritma akan selalu menghasilkan solusi selama masih terdapat makanan pada area map.
Fungsi Seleksi	Memilih himpunan makanan terdekat dengan bot dalam setiap tick.
Fungsi Kelayakan	Memeriksa apakah makanan yang dipilih merupakan makanan terdekat dengan bot.
Fungsi Objektif	Menempuh lintasan terpendek untuk

	memperoleh makanan.
--	---------------------

b. Analisis Efisiensi Solusi

Algoritma greedy by nearest food bekerja dengan menghitung masing-masing jarak dari seluruh makanan yang ada pada map terhadap posisi bot kita pada tick tersebut. Dalam menghitung efisiensi dari algoritma ini, jumlah makanan menjadi faktor penentu banyaknya komputasi yang terjadi. Misal banyaknya makanan pada map dinyatakan sebagai n . Maka, proses perhitungan jarak masing-masing makanan pada map terhadap posisi bot memiliki kompleksitas waktu $O(n)$. Setelah seluruh jarak tersebut dihitung, jarak kemudian akan diurutkan dari yang terkecil ke yang terbesar untuk memenuhi fungsi seleksi. Pengurutan jarak yang menggunakan algoritma *quick sort* akan memiliki kompleksitas waktu sebesar $O(n \log n)$. Oleh karena itu, berdasarkan kedua proses yang terjadi tersebut, strategi *greedy by nearest food* ini memiliki kompleksitas waktu sebesar $O(n \log n)$.

c. Analisis Efektivitas Solusi

Strategi algoritma *greedy by nearest food* ini bertujuan untuk menjadi *player* dengan ukuran terbesar dalam waktu yang cepat dalam rangka memperkecil kemungkinan tereliminasi dari permainan. Namun, perlu diperhatikan bahwa strategi ini menghiraukan potensi bahaya lain dalam memperoleh makanan terdekat. Terdapat kasus dimana makanan berada dalam posisi yang merugikan, misalnya berada dalam gas cloud, maupun berada pada area jangkauan player lain yang lebih besar, sehingga strategi ini menjadi tidak efektif dalam mencapai tujuan akhirnya, yakni memperkecil kemungkinan tereliminasi dari permainan.

3.1.2 Algoritma Greedy by Densest Food Area

a. Pemetaan Elemen Algoritma Greedy

Sama seperti pada sub bab sebelumnya, yaitu bot memiliki tujuan untuk mengonsumsi makanan sebanyak mungkin dalam waktu sesingkat mungkin agar menjadi bot yang terbesar, tetapi kali ini dengan pendekatan bahwa makanan terbanyak dapat diperoleh dengan bergerak ke area dengan tingkat kepadatan makanan paling tinggi di dalam map. Tingkat kepadatan makanan diukur berdasarkan area seluas ukuran dari bot kita. Apabila terdapat beberapa area dengan tingkat kepadatan makanan yang sama, maka bot akan memilih area dengan jarak terdekat untuk menjadi tujuan pergerakan dari bot ini.

Elemen Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh kemungkinan makanan per luas area yang bersesuaian dengan ukuran bot di dalam

	map pada setiap tick
Himpunan Solusi	Seluruh kemungkinan area (yang bersesuaian dengan ukuran bot) dengan food density terbesar dalam map pada setiap tick
Fungsi Solusi	Melakukan pengecekan apakah seluruh makanan di dalam map sudah tidak ada lagi. Hal ini karena bot tidak akan berhenti mencari makan hingga tidak ada makanan lagi yang tersisa. Algoritma akan selalu menghasilkan solusi selama masih terdapat makanan pada area map.
Fungsi Seleksi	Memilih area dengan food density terbesar.
Fungsi Kelayakan	Melakukan pemeriksaan apakah area yang dipilih merupakan area dengan food density terbesar yang memiliki jarak terkecil terhadap bot dalam setiap tick.
Fungsi Objektif	Mendapat makanan terbanyak dalam satu waktu.

b. Analisis Efisiensi Solusi

Algoritma *greedy by densest food area* bekerja dengan menghitung satu per satu kepadatan makanan per luas area seluas besar bot kita pada peta. Jumlah area yang perlu dihitung secara kasar dapat diperoleh dengan membagi luas area peta dengan luas permukaan bot kita. Oleh karena luas peta akan mengecil seiring keberjalanannya permainan, serta luas permukaan bot yang dinamis, jumlah area yang perlu dihitung kepadatannya tidaklah tetap sepanjang keberjalanannya game. Namun, untuk menghitung efisiensi dari algoritma ini, perlu diperhatikan bahwa perhitungan yang terjadi akan sebanding dengan jumlah makanan pada area map karena untuk menghitung densitas makanan pada area-area yang ada, diperlukan untuk menghitung jumlah makanan yang ada pada area tersebut. Secara akumulatif, proses ini sama saja dengan menghitung jumlah seluruh makanan yang ada pada map. Oleh karena itu, perhitungan densitas makanan sendiri memiliki kompleksitas waktu sebesar $O(n)$ di mana n adalah jumlah makanan pada map.

Tidak hanya sampai di situ, strategi ini akan mempertimbangkan faktor jarak apabila terdapat beberapa kandidat solusi yang memiliki *food density* sama besar. Kasus seperti ini akan sering terjadi pada periode awal mula game, di mana luas area bot masih relatif sangat kecil terhadap map, serta makanan masih terdistribusi secara merata pada map. Secara heuristik, kita mengetahui ukuran luas bot pada awal permainan relatif sama dengan ukuran luas area satu buah makanan, yang berarti bisa saja terdapat kandidat area yang sebanding dengan jumlah makanan

(n) itu sendiri. Diperlukan algoritma pengurutan berdasarkan jarak, yang menggunakan algoritma *quick sort* dengan kompleksitas waktu sebesar $O(n \log n)$, Maka dari itu, strategi algoritma *greedy by densest food area* akan memiliki kompleksitas waktu sebesar $O(n)$ pada *best-case scenario* dan $O(n \log n)$ pada *worst-case scenario*.

c. Analisis Efektivitas Solusi

Seperi pada strategi algoritma *greedy by densest food area* bertujuan untuk menjadi *player* dengan ukuran terbesar dalam waktu yang cepat (“sekali gebrak”) dalam rangka memperkecil kemungkinan tereliminasi dari permainan. Namun, dalam pergerakan bot menuju area dengan *food density* paling besar yang dilakukannya secara lurus tanpa memedulikan rintangan yang berada di lintasan lurus tersebut, keamanan lintasan tidaklah diperhitungkan. Bisa saja terdapat rintangan pada lintasan lurus tersebut yang berpotensi mengurangi ukuran bot secara lebih signifikan ketimbang apa yang bisa bot dapatkan dengan menuju ke arah *densest food area* tersebut.

Selain itu, lintasan yang lurus bisa membuat bot mengabaikan kesempatan-kesempatan lain untuk mendapatkan makanan dalam perjalanannya ke area yang dituju atau yang biasa dikenal dengan *opportunity cost*. Hal ini bisa berarti pertumbuhan ukuran bot kita relatif lambat hingga ia mencapai area yang dituju, sedangkan bot lawan bisa saja bertumbuh lebih cepat dalam periode waktu tersebut sehingga menjadi suatu bahaya bagi bot kita.

3.1.3 Algoritma Greedy by Safest Place

a. Pemetaan Elemen Algoritma Greedy

Salah satu paradigma yang dapat dipakai untuk memperoleh kemenangan adalah tidak melakukan kontak dengan objek yang dapat mengurangi ukuran bot kita, seperti lawan yang lebih besar, gas cloud, dan torpedo salvo lawan. Tujuan ini dapat diraih dengan berbagai cara, salah satunya adalah menjauh dari kumpulan bahaya yang ada, yakni pergi ke tempat teraman dalam map. Tempat teraman/*safest place* dalam map ini dievaluasi dengan membobotkan setiap bahaya yang ada dan merumuskan progresi pengurangan bobot bahaya dengan mempertimbangkan faktor jarak. Bobot bahaya ini kemudian diakumulasikan untuk setiap titiknya.

Elemen Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh kemungkinan posisi dengan jarak terjauh dari ancaman, dapat berupa player lain dengan ukuran lebih besar, torpedo salvo, dan gas cloud.
Himpunan Solusi	Seluruh kemungkinan posisi yang terjauh dari berbagai ancaman dengan pembobotan

	ancaman pada setiap tick.
Fungsi Solusi	Melakukan pengecekan apakah sudah tidak ada lagi ancaman bergerak. Hal ini karena solusi hanya berubah apabila terdapat ancaman bergerak (player dan torpedo salvo) yang mengubah posisi teraman dalam tiap tick. Apabila sudah tidak ada ancaman bergerak, posisi teraman akan selalu sama dengan lokasi yang telah dipilih sebelumnya.
Fungsi Seleksi	Memilih area dengan kondisi teraman dalam setiap tick.
Fungsi Kelayakan	Melakukan pemeriksaan apakah area yang dipilih memiliki jarak terdekat terhadap bot.
Fungsi Objektif	Mencari posisi dengan (bobot) ancaman paling minimum.

b. Analisis Efisiensi Solusi

Algoritma greedy by Safest Place bekerja dengan cara mencari tempat atau area paling aman dalam map dalam setiap tick. Terdapat beberapa elemen dalam permainan yang dapat mengancam bot kita (dapat mengeliminasi ataupun mengurangi ukuran bot), yaitu torpedo salvo, gas cloud, dan tentunya lawan yang memiliki ukuran lebih besar dari bot kita. Di antara elemen-elemen yang baru disebutkan sebelumnya, terdapat 2 elemen yang dapat bergerak di dalam map, yaitu gas cloud dan lawan, yang menyebabkan perubahan area-area paling aman dalam map dalam setiap ticknya.

Evaluasi area paling aman dilakukan dengan memerhatikan posisi dari elemen-elemen berbahaya yang dapat mengancam bot kita, evaluasi akan dilakukan untuk setiap gas cloud, torpedo salvo, dan lawan yang ada dalam map, setelah itu akan dilakukan pembobotan akumulatif untuk menentukan area mana yang memiliki status paling aman dalam map (setiap area akan diberi bobot keamanan berdasarkan pengecekan ketiga hal sebelumnya dalam map, kemudian bobot tersebut akan diakumulasi untuk mencari area paling aman). Untuk mengevaluasi setiap area gas cloud akan memerlukan kompleksitas waktu sebesar $O(n)$, dengan n adalah jumlah gas cloud dalam map. Untuk mengevaluasi setiap area dengan torpedo salvo akan memerlukan kompleksitas waktu sebesar $O(m)$, dengan m adalah jumlah torpedo salvo dalam map, sedangkan untuk mengevaluasi setiap area dengan lawan yang lebih besar akan memerlukan kompleksitas waktu sebesar $O(k)$, dengan k adalah jumlah lawan yang lebih besar dibanding bot kita di dalam map. Secara akumulatif ketiga langkah sebelumnya memiliki kompleksitas waktu sebesar $O(n+m+k)$ yang hasilnya serupa dengan $O(n)$. Setelah mendapat kumpulan area paling aman dalam map, bot perlu mencari area dengan jarak terdekat untuk menjadi tujuan pergerakan selanjutnya, maka diperlukan algoritma sorting dengan efisiensi terbaik, yaitu *quick sort* dengan kompleksitas $O(n \log n)$, dengan n adalah jumlah solusi area yang memiliki status paling aman. Didapat bahwa kompleksitas waktu akhir adalah $O(n \log n)$.

c. Analisis Efektivitas Solusi

Strategi algoritma Greedy by Safest Place bertujuan untuk selalu mencari tempat paling aman untuk didatangi dalam setiap tick pada permainan. Tetapi masih terdapat kekurangan pada strategi ini karena bot tidak mempertimbangkan jalur paling aman untuk bergerak menuju ke area paling aman tersebut, dalam pergerakannya menuju ke area paling aman bisa saja bot malah melewati area yang paling berbahaya dalam map dan malah membuat bot tereliminasi seketika. Namun, untuk kasus terbaik dimana area paling aman juga selalu ditempuh oleh bot dengan melewati jalur yang aman, maka algoritma ini dinilai cukup efektif untuk memenangkan permainan Galaxio atau setidaknya menjadi juara ke-2 dalam permainan. Hal ini karena bot yang menjalankan algoritma ini akan selalu lari dari bahaya dan membiarkan lawan saling mengeliminasi satu sama lain.

3.1.4 Algoritma Greedy by Safest Trajectory

a. Pemetaan Elemen Algoritma Greedy

Sama seperti pada sub bab sebelumnya, untuk meminimalisasi kontak dengan objek yang dapat mengurangi ukuran bot kita ataupun mengeliminasi bot kita dari permainan, kita dapat juga memakai strategi dengan cara selalu mencari area pergerakan teraman sesuai dengan kondisi permainan pada setiap tick. Caranya adalah dengan mengevaluasi setiap proyeksi posisi satu tick kedepannya.

Elemen Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh arah yang dapat dituju oleh bot pada setiap tick, yakni 0 - 359 derajat dengan syarat trajectory atau proyeksi posisi dalam satu tick ke depan masih terdapat dalam area map.
Himpunan Solusi	Seluruh arah dengan trajectory atau proyeksi posisi paling aman dari bahaya untuk setiap tick.
Fungsi Solusi	Memeriksa apakah sudah tidak ada lagi area dalam map yang dapat dikunjungi bot tanpa menjadi mati atau keluar dari map.
Fungsi Seleksi	Memilih arah dengan trajectory paling aman dari bahaya (yang paling sedikit mengurangi bobot dari bot).
Fungsi Kelayakan	Memeriksa apakah arah yang dipilih tidak membuat bot keluar dari map.

Fungsi Objektif	Menjadi bot terakhir yang bertahan dalam game.
-----------------	--

b. Analisis Efisiensi Solusi

Algoritma Greedy by Safest Trajectory bekerja dengan cara selalu mengevaluasi kemungkinan pergerakan bot kita dalam satu tick kedepannya (proyeksi pergerakan bot), dan mencari pergerakan paling aman untuk satu tick kedepan. Serupa dengan evaluasi pada algoritma Greedy by Safest Place, algoritma Greedy by Safest Trajectory juga akan melakukan evaluasi terhadap 3 komponen dalam permainan yang dapat membahayakan bot kita, yaitu torpedo salvo, gas cloud, dan lawan yang lebih besar. Perbedaannya adalah evaluasi ini hanya dilakukan dalam lingkup area yang dapat ditempuh oleh bot kita pada satu tick kedepan. Evaluasi yang dilakukan serupa dengan algoritma Greedy by Safest Place dimana akan dilakukan pembobotan keamanan untuk setiap area dalam jangkauan kita dalam satu tick kedepan. Area dengan bobot keamanan paling tinggi akan menjadi area yang dipilih untuk menjadi arah pergerakan selanjutnya.

Untuk mengevaluasi setiap torpedo salvo dalam trajectory bot kita, diperlukan kompleksitas waktu sebesar $O(n)$ dengan n adalah jumlah torpedo salvo dalam trajectory. Untuk mengevaluasi setiap gas cloud dalam trajectory bot kita, diperlukan kompleksitas waktu sebesar $O(m)$ dengan m adalah jumlah gas cloud dalam trajectory. Terakhir, untuk mengevaluasi jumlah lawan yang memiliki ukuran lebih besar dari kita dalam trajectory bot kita, diperlukan kompleksitas waktu sebesar $O(k)$ dengan k adalah jumlah lawan yang lebih besar dalam trajectory bot kita. Akumulasi seluruh proses evaluasi akan menghasilkan langkah teraman berikutnya untuk dilalui oleh bot kita (dalam output berupa arah). Seluruh proses ini memakan kompleksitas waktu sebesar $O(n) + O(m) + O(k)$ atau setara dengan $O(n)$ saja.

c. Analisis Efektivitas Solusi

Algoritma Greedy by Safest Trajectory bertujuan untuk selalu melewati area paling aman dalam setiap tick pada permainan. Dengan selalu melewati trajectory atau jalur paling aman, maka bot kita akan selalu menghindari kemungkinan ancaman yang akan datang dalam setiap tick. Algoritma ini cukup efektif untuk menjadi bot yang bertahan hingga akhir, tetapi karena bot tidak mengejar makanan maka akan sulit untuk memenangkan pertandingan dalam late game, mengingat bahwa ukuran bot yang menerapkan strategi ini pasti akan cukup kecil karena tidak pernah mengejar food dan dengan mempertahankan ukuran yang lebih kecil maka bot akan dapat selalu bergerak relatif lebih cepat terhadap bot lainnya. Strategi ini akan cukup baik jika bot hanya ingin memenangkan posisi ke-2 dalam permainan.

3.1.5 Pemetaan Elemen Algoritma Greedy by Feeding on Smaller Bot

a. Pemetaan Elemen Algoritma Greedy

Salah satu cara untuk meningkatkan peluang menang adalah dengan meningkatkan ukuran bot kita dan mengeliminasi musuh di saat yang bersamaan. Hal ini dapat dilakukan

dengan cara memangsa lawan sehingga ukuran bot kita bertambah dan sekaligus mengurangi jumlah musuh dalam permainan. Kita hanya akan mengonsumsi lawan yang berukuran lebih kecil dari bot kita, dengan prioritas lawan yang harus dikonsumsi / dieliminasi terlebih dahulu adalah lawan terbesar di antara lawan yang memiliki ukuran lebih kecil dari bot kita. Hal ini untuk mencegah lawan terbesar tersebut nantinya akan memiliki ukuran yang lebih besar dari bot kita dalam waktu dekat dan dapat mengeliminasi kita nantinya.

Elemen Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh lokasi dari lawan yang memiliki ukuran lebih kecil dibanding bot kita dalam setiap tick.
Himpunan Solusi	Seluruh lokasi dari lawan yang memiliki ukuran lebih kecil dari bot kita dalam setiap tick
Fungsi Solusi	Melakukan pengecekan apakah tidak ada lagi bot yang memiliki ukuran lebih kecil dibanding bot kita.
Fungsi Seleksi	Memilih lawan memiliki ukuran yang lebih kecil dibanding bot kita.
Fungsi Kelayakan	Melakukan pemeriksaan apakah lawan yang dipilih merupakan yang terbesar di antara seluruh kandidat solusi.
Fungsi Objektif	Mengeliminasi lawan dengan ukuran yang lebih kecil dengan berharap ukuran kumulatif hasil memakan lawan dapat membawa bot kita menjadi bot yang paling besar dan bertahan sampai akhir.

b. Analisis Efisiensi Solusi

Algoritma Greedy by Feeding on Smaller bot bekerja dengan cara selalu mencari bot terbesar yang lebih kecil dari ukuran bot kita untuk dikejar dan dimangsa. Dalam setiap tick, algoritma ini akan selalu mencari seluruh lawan yang memiliki ukuran yang lebih kecil dari bot kita dan kemudian mencari lawan terbesar yang masih memiliki ukuran lebih kecil dari bot kita. Untuk mencari seluruh lawan dengan ukuran lebih kecil dari ukuran bot kita maka diperlukan kompleksitas waktu sebesar $O(n)$, dengan n adalah jumlah lawan dalam map. Untuk mencari lawan terbesar di antara seluruh lawan yang memiliki ukuran lebih kecil dari bot kita, diperlukan kompleksitas waktu $O(m)$, dengan m adalah jumlah lawan yang lebih kecil dari ukuran bot kita. Seluruh proses dalam algoritma ini akan memerlukan waktu $O(n) + O(m)$, yang mana kurang lebih setara dengan $O(n)$.

c. Analisis Efektivitas Solusi

Algoritma Greedy by Feeding on Smaller bot bertujuan untuk mencari bot terbesar yang lebih kecil dari ukuran bot kita untuk dikejar dan kemudian dimangsa atau dieliminasi. Namun, perlu diperhatikan bahwa bot lawan yang lebih kecil memiliki kecepatan yang lebih cepat dibandingkan kecepatan kita. Permasalahan ini dapat diakali dengan adanya efek after burner yang membuat kecepatan bot kita menjadi 2 kali lebih cepat. Akan tetapi, perlu diingat bahwa lawan yang lebih kecil juga memiliki kemampuan ini dan akan menghasilkan kecepatan yang lebih cepat juga. Oleh karena itu, kesuksesan strategi ini juga bergantung pada algoritma bot lawan dalam menghindari musuh yang lebih besar. Apabila algoritma bot lawan tidak memiliki kemampuan yang baik dalam menghindari lawan yang lebih besar, besar kemungkinan strategi ini dapat berjalan dengan efektif. Di sisi lain, pada fase awal dari permainan, perlu dipastikan bahwa bot kita memiliki ukuran yang lebih besar daripada bot lain agar dapat memangsa lawan. Hal ini berarti strategi algoritma ini perlu dikombinasikan dengan strategi dalam mendapat makanan yang efektif sehingga kedua strategi tersebut dapat membuat bot memenuhi tujuannya.

3.1.6 Algoritma Greedy by Destroying Opponents

a. Pemetaan Elemen Algoritma Greedy

Paradigma lain yang dapat dipakai dalam memenangkan permainan Galaxio adalah dengan memakai strategi yang paling merugikan lawan, yakni maksimalisasi pengurangan bobot lawan. Selain itu, pemilihan lawan yang perlu dirugikan juga menjadi strategi dari algoritma ini. Lawan yang akan dipilih adalah lawan yang paling besar ukurannya dengan harapan lawan tersebut akan berukuran semakin kecil dan dapat tereliminasi dari permainan, yang secara tidak langsung mengurangi ancaman terhadap bot kita .

Element Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh command yang dapat mengurangi ukuran lawan dalam setiap tick.
Himpunan Solusi	Seluruh command yang dapat mengurangi ukuran lawan dengan kerugian terbesar bagi lawan dalam setiap tick.
Fungsi Solusi	Memeriksa apakah sudah tidak ada lagi lawan di dalam permainan.
Fungsi Seleksi	Memilih lawan dengan ukuran terbesar yang ada dalam permainan serta command yang paling merugikan bagi lawan tersebut.
Fungsi Kelayakan	Memeriksa apakah command yang dipilih

	membuat bot kita menjadi tereliminasi dari game karena ukuran yang terlalu kecil mengingat penggunaan command torpedo salvo dapat mengurangi ukuran bot kita
Fungsi Objektif	Mengurangi ukuran lawan dengan tujuan untuk mengeliminasi lawan dari permainan.

b. Analisis Efisiensi Solusi

Dalam strategi algoritma *greedy by destroying opponents*, bot kita perlu menghitung seluruh bot yang ada untuk mencari bot yang ukurannya yang paling besar sebagai prioritas untuk diserang. Anggap jumlah bot yang berada dalam map adalah m . Maka, untuk menghitung ukuran seluruh bot akan memiliki kompleksitas waktu $O(m)$. Setelah itu, bot perlu menuju lokasi dari bot lawan tersebut dan perlu memperhitungkan jarak yang tepat agar torpedo dapat memberikan *damage* kepada lawan. Jarak yang tepat tersebut ditentukan dengan jumlah makanan yang berada di laju torpedo menuju bot lawan. Batas maksimum makanan yang berada dalam jalur torpedo adalah setengah dari ukuran torpedo agar torpedo masih memiliki ukuran yang cukup signifikan untuk merusak bot lawan. Namun, untuk menghitung jumlah makanan yang berada pada jalur torpedo, sistem harus menilik seluruh makan yang ada pada map. Oleh karena itu, dibutuhkan kompleksitas waktu sebesar $O(n)$ dengan n adalah jumlah makanan pada map untuk melakukan proses ini. Maka dari itu, strategi ini akan memiliki kompleksitas waktu sebesar $O(m + n)$.

c. Analisis Efektivitas Solusi

Algoritma Greedy by Destroying Opponents bertujuan untuk mengurangi ukuran musuh atau mengeliminasi seluruh musuh dalam keberjalanannya permainan sehingga dapat menjadi bot terakhir yang bertahan dalam map dan menjadi pemenang. Proses eliminasi musuh dimulai dari musuh yang paling mungkin mengancam bot kita dalam permainan, yaitu musuh yang paling besar di dalam map, hal ini bertujuan untuk mengurangi kemungkinan kita dimangsa oleh musuh terbesar tersebut dan meningkatkan waktu bertahan bot kita dalam permainan. Strategi ini bisa jadi kurang efektif karena bot hanya bertujuan untuk terus menerus menyerang musuh tanpa memikirkan keamanan dari bot itu sendiri, bot bisa saja tereliminasi saat berusaha untuk terus menerus menyerang musuh. Namun, apabila dalam keberjalanannya game, bot selalu berada dalam posisi yang aman dan bisa terus menerus menyerang musuh, maka strategi ini dapat menjadi cukup efektif untuk memenangkan permainan.

3.1.7 Algoritma Greedy by Nearest-Safe Food

a. Pemetaan Elemen Algoritma Greedy

Masih dengan konsiderasi bahwa untuk meningkatkan kemungkinan memenangkan permainan ini, kita harus menjadi player dengan ukuran terbesar tanpa mengambil resiko yang dapat mengurangi ukuran bot kita. Salah satu pendekatan yang dapat dilakukan untuk mencapai hal ini adalah dengan selalu mengonsumsi makanan terdekat yang tidak memiliki resiko untuk mengurangi ukuran bot kita. Caranya adalah dengan mencari makanan terdekat dan mengkalkulasi apakah ada resiko yang dapat kita terima dengan mengonsumsi makanan tersebut, jika tidak ada, maka makanan tersebut akan dipilih untuk dikonsumsi.

Element Algoritma Greedy	Pemetaan
Himpunan Kandidat	Seluruh kemungkinan pergerakan bot ke arah makanan yang tersedia pada peta dalam setiap tick.
Himpunan Solusi	Seluruh opsi pergerakan bot ke arah makanan yang terdekat terhadap bot tanpa pengurangan ukuran pada setiap tick.
Fungsi Solusi	Melakukan pengecekan apakah sudah tidak ada lagi makanan yang bebas dari bahaya di dalam map.
Fungsi Seleksi	Memilih himpunan makanan terdekat dengan bot dalam setiap tick.
Fungsi Kelayakan	Memeriksa apakah makanan yang dipilih merupakan makanan yang paling aman untuk dikonsumsi dalam setiap tick.
Fungsi Objektif	Mengonsumsi makanan terdekat tanpa resiko mengurangi ukuran bot kita ataupun resiko tereliminasi dari permainan.

b. Analisis Efisiensi Solusi

Sama seperti pada algoritma *greedy by nearest food*, algoritma ini bekerja dengan menghitung masing-masing jarak dari seluruh makanan yang ada pada map terhadap posisi bot kita pada tick tersebut. Maka, proses perhitungan jarak masing-masing makanan pada map terhadap posisi bot memiliki kompleksitas waktu $O(n)$. Setelah seluruh jarak tersebut dihitung, jarak kemudian akan diurutkan dari yang terkecil ke yang terbesar untuk memenuhi fungsi seleksi. Pengurutan jarak yang menggunakan algoritma *quick sort* akan memiliki kompleksitas waktu sebesar $O(n \log n)$. Selanjutnya, makanan yang paling dekat dengan posisi bot kita akan dicek tingkat keamanannya. Apabila terdapat sejumlah m bot lawan dan sebagian besar makanan berada pada posisi yang berbahaya (berada pada area *trajectory* lawan di tick berikutnya), maka dibutuhkan pengecekan

sebanyak $n \times m$. Maka, strategi *greedy by nearest-safest food* ini akan memiliki kompleksitas waktu sebesar $O(n \log n + mn)$.

c. Analisis Efektivitas Solusi

Algoritma Greedy by Nearest-Safe-Food bertujuan untuk memperoleh makanan terdekat dengan tetap mempertimbangkan keamanan untuk memperoleh makanan tersebut dengan tujuan untuk menambah ukuran bot kita secepat mungkin tanpa meresikkan kehilangan ukuran ataupun tereliminasi oleh lawan yang lebih besar. Strategi ini bisa cukup efektif untuk membuat bot kita tetap bertahan dalam permainan dan sekaligus menjadi bot dengan ukuran terbesar di dalam permainan. Hal ini meningkatkan kemungkinan kita untuk bertahan dan menjadi pemenang di late game.

3.2 Strategi Greedy dengan Belajar dari Hukum Alam - Gravitasi

Dalam permainan Galaxio ini, mayoritas tick permainan akan dilaksanakan perintah untuk berjalan (FORWARD). Pemilihan arah dan lintasan dengan tepat ini juga dapat dikatakan sebagai salah satu strategi untuk memenangkan permainan Galaxio. Secara garis besar, bot akan mendatangi objek-objek yang dapat menambahkan ukurannya serta menghindari objek-objek yang dapat mengurangi ukurannya. Untuk memudahkan kita memahaminya, objek-objek dalam permainan Galaxio dapat kita anggap sebagai sebuah massa yang nilainya sesuai dengan ukurannya. Objek-objek yang dapat menambah ukuran bot kita (*food*, *superfood*, dan *player* lain yang lebih kecil) dianggap memiliki massa positif sedangkan objek-objek yang dapat mengurangi massa kita dianggap memiliki massa negatif. Bot kita akan cenderung menghampiri objek yang bermassa positif dan menghindari objek yang bermassa negatif.

Dalam menghampiri ataupun menghindari objek-objek ini, dibutuhkan strategi yang optimal agar *bot* mendapat tambahan massa (ukuran) dalam waktu secepatnya. Dengan kata lain, *bot* membutuhkan lintasan yang optimal dalam menambah atau mempertahankan ukurannya. Strategi pada umumnya menawarkan bot untuk pergi ke arah makanan yang terdekat (*nearest food*) tetapi strategi seperti ini tidak mempertimbangkan densitas makanan di tempat yang dituju. Adapun strategi lain yang umum pula digunakan adalah bot ke arah area dengan densitas makanan terbesar (*densest food area*). Namun, strategi seperti ini tidak mempertimbangkan *cost* / efisiensi perjalanan menuju ke area tersebut.

Alternatif strategi yang ingin kami eksplorasi dalam subbab ini adalah strategi *greedy* yang kami beri nama *Greedy by Gravitational Object*. Pada prinsipnya, setiap objek dalam permainan Galaxio dapat kita anggap menghasilkan sebuah medan gravitasinya masing-masing. Konsep ini akan menghasilkan akumulasi medan gravitasi pada setiap posisi oleh setiap objek. Alhasil, terbentuklah garis-garis medan gravitasi di sepanjang luas area map permainan Galaxio. Strategi ini akan membuat *bot* berjalan ke arah garis medan gravitasi yang lebih tinggi pada setiap titiknya sehingga *bot* akan tetap memakan makanan yang ada di dekatnya, sembari tertarik ke arah medan gravitasi yang lebih besar (tempat dengan densitas makanan yang tertinggi). Dengan demikian, bot akan mengikuti lintasan yang efektif dan

optimum dalam memakan makanan yang ada. Sebaliknya, objek-objek dengan massa negatif akan menghasilkan medan gravitasi yang mendorong bot untuk menjauh dari objek tersebut.

Implementasi algoritma *Greedy by Gravitational Object* ini dapat dilakukan dengan membuat sebuah persamaan 3 dimensi (z) yang bergantung pada posisi x dan y. Besarnya nilai z atau medan pada suatu titik akan diperoleh dari akumulasi perhitungan medan gravitasi setiap objek dalam permainan terhadap titik tersebut. Setelah mendapat persamaan 3 dimensi tersebut, akan dicari turunan berarah untuk menemukan arah dengan perubahan medan paling besar pada koordinat tersebut sehingga *bot* dalam bergerak sesuai dengan arah tersebut. Persamaan bidang akan berubah setiap ticknya oleh karena objek-objek pada map yang dinamis dan dapat menghilang setelah terjadi aksi ataupun *event* tertentu. Strategi ini akan membawa *bot* menuju makanan terdekatnya karena makanan terdekat akan menghasilkan medan gravitasi yang paling berarti pada suatu titik. Di sisi lain, akumulasi dari medan gravitasi yang dihasilkan oleh area dengan kepadatan makanan yang tinggi akan menuntun *bot* untuk sekaligus ke arah area tersebut.

Namun, algoritma ini belum dapat kami implementasikan pada tugas kali ini mengingat segala keterbatasan waktu serta kemampuan bahasa pemrograman Java dalam mengeksekusi rancangan algoritma yang ada. Bahasa Java tidak memiliki fitur yang mumpuni untuk menyelesaikan persamaan multivariabel yang kompleks serta untuk menyelesaikan persoalan turunan berarah pada persamaan tiga dimensi. Sejatinya hal ini dapat diakali dengan menghubungkan program dengan MathLab. Namun, penggeraan seperti ini tidak dimungkinkan untuk tugas kali ini.

3.3 Strategi Greedy yang Dipilih

Dalam permainan Galaxio ini, terdapat banyak faktor yang perlu dipertimbangkan untuk memutuskan sebuah langkah yang diharapkan dapat menuntun pemain menuju kemenangan. Hal ini tertuang dalam berbagai strategi yang telah dipaparkan pada sub bab 3.1. Berangkat dari hal tersebut, kami memutuskan untuk menggunakan pendekatan strategi algoritma *greedy* yang mempertimbangkan masing-masing faktor yang ada serta penanganan optimal yang dapat diambil.

Agar strategi-strategi yang ada dapat dipadupadankan, kami terlebih dahulu memformulasikan tingkatan prioritas untuk setiap strategi yang kami pakai. Tingkatan prioritas tersebut dibutuhkan agar terdapat urutan atau aturan eksekusi untuk setiap langkah dalam persoalan, yang dalam hal ini merupakan setiap tick dalam permainan Galaxio.

Strategi *greedy* yang kami pilih untuk diimplementasikan dalam *bot* kami adalah kombinasi dari beberapa strategi *greedy* yang telah dijabarkan pada sub bab 3.1. Beberapa strategi tersebut beserta tingkat prioritasnya dari tertinggi ke terendah adalah sebagai berikut:

1. Sub-Algoritma *Greedy by Safest Place*;
2. Sub-Algoritma *Greedy by Destroying Opponents*;
3. Sub-Algoritma *Greedy by Safest Trajectory*;
4. Sub-Algoritma *Greedy by Nearest-Safe Food*;

5. Sub-Algoritma *Greedy by Feeding on Smaller Bot*.

Tingkat prioritas tersebut kami susun berdasarkan prioritas untuk meminimasi resiko terlebih dahulu, kemudian dilanjut dengan memaksimasi keuntungan. Sub-algoritma yang bertanggung jawab atas meminimasi resiko adalah sub-algoritma nomor 1, 2, dan 3. Ketiga algoritma tersebut meminimasi risiko dengan menempatkan *bot* di area yang minim bahaya, mereduksi ukuran lawan sembari menambah ukuran sendiri (*Destroying Opponents*) agar semakin sedikit ancaman yang ada, serta memilih lintasan yang tidak mereduksi ukuran *bot*. Apabila dalam suatu *tick*, tidak ada yang dapat dieksekusi karena sudah memenuhi seluruh kondisi yang diinginkan oleh *bot*, algoritma akan beralih ke strategi maksimasi, yang ditangani oleh sub-algoritma nomor 4 dan 5. Maksimasi dilakukan dengan cara memakan makanan yang terdekat serta memangsa *bot* lain yang lebih kecil.

Bab 4

Implementasi dan Pengujian

4.1 Implementasi Algoritma *Greedy* dalam PseudoCode

4.1.1 Implementasi fungsi utama: computeNextPlayerAction

```
function computeNextPlayerAction(PlayerAction playerAction)

    /* updating state dari permainan */
    call updateSelfState()

    /* jika permainan belum dimulai, fungsi akan langsung return / tidak dieksekusi*/
    if gameState.world.getCurrentTick = 0 then
        return

    /* boolean isExecuted sebagai mark apakah aksi sudah ada aksi yang diputuskan untuk dieksekusi */
    boolean isExecuted;

    /* melakukan pemetaan terhadap teleporter yang ada di map ke pemiliknya, agar kemudian dapat diidentifikasi*/
    call mappingTeleporter

    /* pada fungsi utama yang menentukan */

    /*
     * memanggil fungsi tryTeleport, akan melakukan teleportasi jika player memiliki teleport yang telah ditembakkan dan
     * koordinat player setelah teleportasi aman dan memberi keuntungan kepada player (memakan player lain)
     */
    isExecuted <- call tryTeleport
    if isExecuted then
        return

    /*
     * memanggil fungsi torpedo defense, akan melakukan handling jika terdapat torpedo yang mengarah ke player
     */
    isExecuted <- call torpedoDefense
    if isExecuted then
        return

    /*
     * memanggil fungsi torpedo defense, akan melakukan handling jika terdapat teleporter yang mengarah ke player dan
```

```

        * teleport tersebut dimiliki oleh player yang lebih besar
    */
    isExecuted <- call avoidTeleporter
    if isExecuted then
        return

    /*
     * memanggil fungsi escape, akan mengarahkan player menjauh dari
     * ancaman jika terdapat ancaman di batas aman player
    */
    isExecuted <- call escape
    if isExecuted then
        return

    /*
     * memanggil fungsi smallerTarget, akan menembakkan torpedo ke
     * player yang lebih kecil jika terdapat player yang bisa ditembak
    */
    isExecuted <- call smallerTarget
    if isExecuted then
        return

    /*
     * memanggil fungsi getCentered, akan mengarahkan player kembali
     * ke tengah map jika player mengarah ke luar map/ berada pada batas
     * aman pinggiran map
    */
    isExecuted <- call getCentered
    if isExecuted then
        return

    /*
     * memanggil fungsi getCollectable, akan mengarahkan player ke
     * collectable dengan bobot tertinggi, berdasarkan jarak dan jenisnya
    */
    isExecuted <- call gotoCollectable
    if isExecuted then
        return

    /*
     * memanggil fungsi hunting, akan menyerang player yang lebih
     * besar dari bot dalam batas aman (jika ada), jika tidak ada
     * akan mengarah ke player yang lebih kecil
    */
    isExecuted <- call hunting
    if isExecuted then
        return

    /* langkah default saat tidak ada greedy yang berhasil
     * menemukan langkah yang tepat */

```

```

playerAction.action <- FORWARD
playerAction.heading <- Random().nextInt(360)
return

```

4.1.2 Implementasi Fungsi Tamabahan

```

// Declare a function named "sum" that takes in an array of
integers and returns an integer
FUNCTION sum(array: integer[]) RETURNS integer
    // Declare a variable named "result" and initialize it to 0
    result = 0
    // Loop through each integer in the array
    FOR i = 0 TO length(array) - 1
        // Add the current integer to the "result" variable
        result = result + array[i]
    END FOR
    // Return the final value of the "result" variable
    RETURN result
END FUNCTION

// Declare an array of integers
DECLARE myArray = [1, 2, 3, 4, 5]

// Call the "sum" function with the "myArray" array as an argument
// and print the result
PRINT sum(myArray)

```

```

// Declare a function named "isInside_NT" that takes in a
"GameObject" item and returns a boolean value
FUNCTION isInside_NT(item: GameObject) RETURNS boolean
    // Declare a variable named "safe_radius" and set it to the sum
    // of the bot size, item size, and item speed
    safe_radius = this.bot.getSize() + item.getSize() +
item.getSpeed()
    // Calculate the distance between this bot and the item, and
    // check if it's less than the safe radius
    IF getDistanceBetween(this.bot, item) < safe_radius THEN
        // Return true if the item is inside the safe radius
        RETURN true
    ELSE

```

```

    // Return false if the item is outside the safe radius
    RETURN false
END IF
END FUNCTION

```

```

// Declare a function named "isSafeToHead" that takes in an integer
heading and returns a boolean value
FUNCTION isSafeToHead(heading: integer) RETURNS boolean
    // Call the "isSafeToHead" function with the "heading"
parameter and a value of 1 for the "distance" parameter
    RETURN isSafeToHead(heading, 1)
END FUNCTION

```

```

function isSafeToHead(heading, sensitivity) returns boolean:
    heading = heading mod 360
    mapCenterPoint_x = gameState.world.centerPoint.x
    mapCenterPoint_y = gameState.world.centerPoint.y
    mapRadius = gameState.world.radius

    biggerPlayerList = gameState.getPlayerGameObjects()
        .filter(item -> item.getSize() >=
bot.getSize() && !item.id.equals(bot.id))
        .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
        .collect(Collectors.toList())
    supernovaBombList = gameState.getGameObjects()
        .filter(item -> item.getGameObjectType() ==
ObjectTypes.SUPERNOVABOMB)
        .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
        .collect(Collectors.toList())
    gasCloudList = gameState.getGameObjects()
        .filter(item -> item.getGameObjectType() ==
ObjectTypes.GASCLOUD)
        .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
        .collect(Collectors.toList())

    dummy_x = bot.getPosition().x + cos(toRadians(heading)) *
bot.getSpeed()

```

```

dummy_y = bot.getPosition().y + sin(toRadians(heading)) * 
bot.getSpeed()
dummy_size = bot.getSize()

for each item in biggerPlayerList:
    dif_x = dummy_x - item.getPosition().x
    dif_y = dummy_y - item.getPosition().y
    safe_radius = dummy_size + item.getSize() + item.getSpeed()

    if dif_x * dif_x + dif_y * dif_y <= safe_radius *
safe_radius * sensitivity:
        return false

for each item in gasCloudList:
    dif_x = dummy_x - item.getPosition().x
    dif_y = dummy_y - item.getPosition().y
    safe_radius = dummy_size + item.getSize() + item.getSpeed()

    if dif_x * dif_x + dif_y * dif_y <= safe_radius *
safe_radius * sensitivity:
        return false

for each item in supernovaBombList:
    dif_x = dummy_x - item.getPosition().x
    dif_y = dummy_y - item.getPosition().y
    safe_radius = dummy_size + item.getSize() + item.getSpeed()

    if dif_x * dif_x + dif_y * dif_y <= safe_radius *
safe_radius * sensitivity:
        return false

    dif_x = dummy_x - mapCenterPoint_x
    dif_y = dummy_y - mapCenterPoint_y
    safe_radius = mapRadius - dummy_size

    if dif_x * dif_x + dif_y * dif_y > safe_radius * safe_radius /
sensitivity:
        return false

return true

```

```

// Get list of torpedoes and sort them by distance from the bot
torpedoList = gameState.getGameObjects().stream()
    .filter(item -> item.getGameObjectType() equals
ObjectTypes.TORPEDOSALVO)
    .sorted(Comparator.comparing(item -> getDistanceBetween(bot,
item)))
    .collect(Collectors.toList())

// Iterate over the list of torpedoes
for i in range(torpedoList.size()):
    // Calculate hit degree
    hitDegree = Math.atan((this.bot.getSize() + 10) /
getDistanceBetween(this.getBot(), torpedoList.get(i)) * 180 /
Math.PI)

        // Check if the torpedo is inside the bot's no-torpedo zone and
heading towards the bot
        if isInside_NT(torpedoList.get(i)) and -45 <=
torpedoList.get(i).getHeading() -
torpedoList.get(i).getHeadingBetween(this.bot) and
(torpedoList.get(i).getHeading() -
torpedoList.get(i).getHeadingBetween(this.bot)) <= 45:
            // Activate shield if available and not already activated
            if this.bot.getShieldCount() equals 1 and
this.bot.getArrEffects()[4] equals 0:
                playerAction.action = PlayerActions.ACTIVATESHIELD
                this.playerAction = playerAction
                return true
            // Fire torpedoes at the incoming torpedo
            else if not (this.bot.getShieldCount() equals 0 or
this.bot.getArrEffects()[4] equals 1):
                playerAction.heading =
this.getBot().getHeadingBetween(torpedoList.get(i))
                playerAction.action = PlayerActions.FIRETORPEDOES
                return true
            else:
                // Do nothing
                pass
        // Check if the torpedo is heading towards the bot within the
hit degree
        else if -1*hitDegree <= torpedoList.get(i).getHeading() -
torpedoList.get(i).getHeadingBetween(this.bot) and
torpedoList.get(i).getHeading() -

```

```

torpedoList.get(i).getHeadingBetween(this.bot) <= hitDegree:
    // Start afterburner and move away from the torpedo
    if this.bot.getArrEffects()[0] equals 0:
        playerAction.action = PlayerActions.STARTAFTERBURNER
        playerAction.heading = torpedoList.get(0).getHeading()
+ 90 // + 90 to move in a tangential direction for faster escape
        return true
    else:
        playerAction.action = PlayerActions.FORWARD
        playerAction.heading = torpedoList.get(0).getHeading()
+ 90 // + 90 to move in a tangential direction for faster escape
        return true

return false

```

```

function escape(playerAction):
    biggerPlayerList = gameState.getPlayerGameObjects()
                    .filter(item -> item.getGameObjectType() ==
ObjectTypes.PLAYER and item.getSize() > bot.getSize())
                    .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
                    .collect(Collectors.toList())

    if biggerPlayerList.size() != 0:
        print("Bigger Player")
        if getDistanceBetween(biggerPlayerList.get(0), getBot()) / 60
<=
biggerPlayerList.get(0).getSize() / biggerPlayerList.get(0).getSpeed():
            if getBot().torpedoSalvoCount != 0 and
getBot().getSize() >= 30:
                playerAction.heading =
getHeadingBetween(biggerPlayerList.get(0))
                print("Torpedo 2")
                print(playerAction.heading)
                print(getBot().getSize())
                playerAction.action = PlayerActions.FIRETORPEDOES
                return true
            else:
                #kabur
                playerAction.heading =

```

```

getHeadingBetween(biggerPlayerList.get(0)) + 180
        print("Kabur bang")
        print(playerAction.heading)
        print(getBot().getSize())
        if getBot().effects != 1:
            playerAction.action =
PlayerActions.STARTAFTERBURNER
        else:
            playerAction.action = PlayerActions.FORWARD
            return true
    else:
        # ga nyerang
        if getBot().effects == 1:
            playerAction.action = PlayerActions.STOPAFTERBURNER
            return true
    return false

```

```

function escape(playerAction):
    biggerPlayerList = gameState.getPlayerGameObjects()
        .stream()
        .filter(item -> item.getGameObjectType() ==
ObjectTypes.PLAYER && item.getSize() > bot.getSize())
        .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
        .collect(Collectors.toList())

    if biggerPlayerList.size() != 0:
        print "Bigger Player"
        if getDistanceBetween(biggerPlayerList.get(0), getBot()) /
60 <= biggerPlayerList.get(0).getSize() /
biggerPlayerList.get(0).getSpeed():
            if getBot().torpedoSalvoCount != 0 &&
getBot().getSize() >= 30:
                playerAction.heading =
getHeadingBetween(biggerPlayerList.get(0))
                playerAction.action = PlayerActions.FIRETORPEDOES
                print "Torpedo 2"
                print playerAction.heading
                print getBot().getSize()
                return true
    else:

```

```

        playerAction.heading =
getHeadingBetween(biggerPlayerList.get(0)) + 180
            print "Kabur bang"
            print playerAction.heading
            print getBot().getSize()
            if getBot().effects != 1:
                playerAction.action =
PlayerActions.STARTAFTERBURNER
            else:
                playerAction.action = PlayerActions.FORWARD
            return true
        else:
            if getBot().effects == 1:
                playerAction.action = PlayerActions.STOPAFTERBURNER
            return true
    return false

function smallerTarget(playerAction):
    smallerPlayerList = gameState.getPlayerGameObjects()
        .stream()
        .filter(item -> item.getGameObjectType() ==
ObjectTypes.PLAYER && item.getSize() < bot.getSize())
        .sorted(Comparator.comparing(item ->
getDistanceBetween(bot, item)))
        .collect(Collectors.toList())

    if smallerPlayerList.size() != 0:
        print "Smaller Player"
        if getDistanceBetween(smallerPlayerList.get(0), getBot()) /
60 <= smallerPlayerList.get(0).getSize() /
smallerPlayerList.get(0).getSpeed():
            if getBot().torpedoSalvoCount != 0 &&
getBot().getSize() >= 30:
                playerAction.heading =
getHeadingBetween(smallerPlayerList.get(0))
                playerAction.action = PlayerActions.FIRETORPEDOES
                print "Torpedo 3"
                print playerAction.heading
                print getBot().getSize()
                return true
        else:
            return false

```

```

function unhandled(playerAction):
    getCentered(playerAction, 1, true)

function getCentered(playerAction):
    getCentered(playerAction, 1)

function getCentered(playerAction, sensitivity):
    getCentered(playerAction, sensitivity, false)

function getCentered(playerAction, sensitivity, pass):
    dif_x = 0
    dif_y = 0

    x = bot.getPosition().x
    y = bot.getPosition().y
    mapCenterPoint_x = gameState.world.centerPoint.x
    mapCenterPoint_y = gameState.world.centerPoint.y
    mapRadius = gameState.world.radius

    dif_x = x - mapCenterPoint_x
    dif_y = y - mapCenterPoint_y
    safe_radius = mapRadius - bot.size

    if dif_x * dif_x + dif_y * dif_y <= safe_radius *
    safe_radius / sensitivity or pass:
        return false

    direction = toDegrees(Math.atan2(mapCenterPoint_y - y,
    mapCenterPoint_x - x))
    direction = (direction + 360) % 360

```

```

gotoCollectable(playerAction):
    return gotoCollectable(playerAction, 1)

gotoCollectable(playerAction, sensitivity):
    if gameState.getGameObjects().isEmpty():
        print "unable to compute next move: empty game objects"
        playerAction.action = STOP
    superFoodWeight = 2
    foodWeight = 1

```

```

supernovaPickupWeight = 2
collectableList = gameState.getGameObjects()
    .filter(item -> item.getGameObjectType() == FOOD or
           item.getGameObjectType() == SUPERFOOD or
           item.getGameObjectType() == SUPERNOVAPICKUP)
    .sorted(item ->
        if (item.getGameObjectType() == FOOD):
            return getDistanceBetween(this.bot,
item) * foodWeight
        else if (item.getGameObjectType() == SUPERFOOD):
            return getDistanceBetween(this.bot,
item) * superFoodWeight
        else:
            return getDistanceBetween(this.bot,
item) * supernovaPickupWeight
    )
    .collect(toList())
for i in range(collectableList.size()):
    heading = getHeadingBetween(collectableList.get(i))
    if isSafeToHead(heading, sensitivity):
        playerAction.action = FORWARD
        playerAction.heading = heading
        this.playerAction = playerAction
        return true
    playerAction.action = STOP
    this.playerAction = playerAction
return false

avoidTeleporter(playerAction):
    teleporterList = gameState.getGameObjects()
        .filter(item -> item.getGameObjectType() == TELEPORTER)
        .sorted(item -> getDistanceBetween(bot, item))
        .collect(toList())
    if teleporterList.size() == 0:
        return false
    nearTeleporterPlayerList = gameState.getGameObjects()
        .filter(item ->
item.getGameObjectType() == PLAYER and item != this.bot)

```

```

        .sorted(item ->
getDistanceBetween(teleporterList.get(0), item))
        .collect(toList())
    if nearTeleporterPlayerList.size() == 0:
        return false
    if nearTeleporterPlayerList.get(0).getSize() <= this.bot.size:
        return false
    hitDegree = atan((this.bot.getSize() + 10) /
                      getDistanceBetween(this.getBot(),
teleporterList.get(0)) * 180 / PI )
    if (-1 * hitDegree <= teleporterList.get(0).getHeading() -
teleporterList.get(0).getHeadingBetween(this.bot) and
        teleporterList.get(0).getHeading() -
teleporterList.get(0).getHeadingBetween(this.bot) <= hitDegree):
        if this.bot.getArrEffects()[0] == 0:
            playerAction.action = STARTAFTERBURNER
            playerAction.heading =
teleporterList.get(0).getHeading() + 90
            return true
        else:
            playerAction.action = FORWARD
            playerAction.heading =
teleporterList.get(0).getHeading() + 90
            return true
    return false

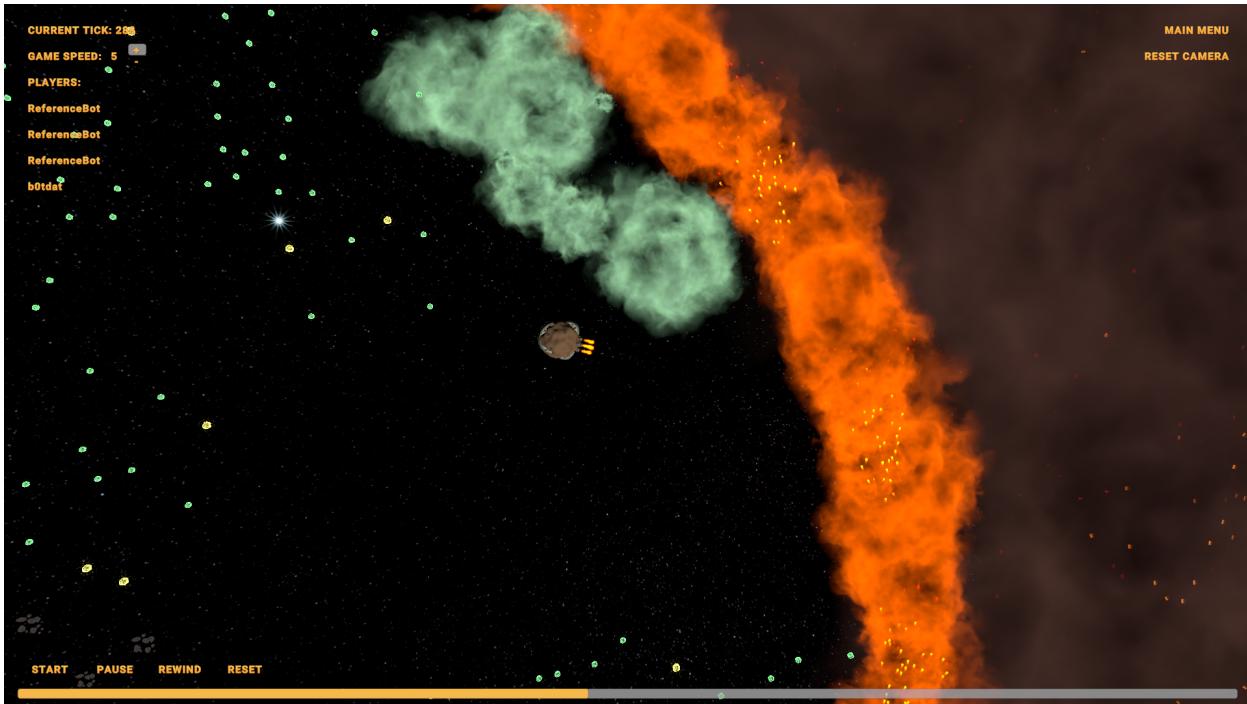
isIn(x, y, object_x, object_y, radius):
    return ((x - object_x) * (x - object_x) + (y - object_y) * (y - object_y)) <= radius * radius

getSafeHeadingBetween(object):
    initialHeading = getHeadingBetween(object)
    deflect = 0
    while deflect < 180:
        if isSafeToHead(initialHeading + deflect):
            return initialHeading + deflect
        else if isSafeToHead(initialHeading - deflect):
            return initialHeading - deflect

```

4.2 Pengujian Program

4.2.1 Bot Menghindari Gas Cloud

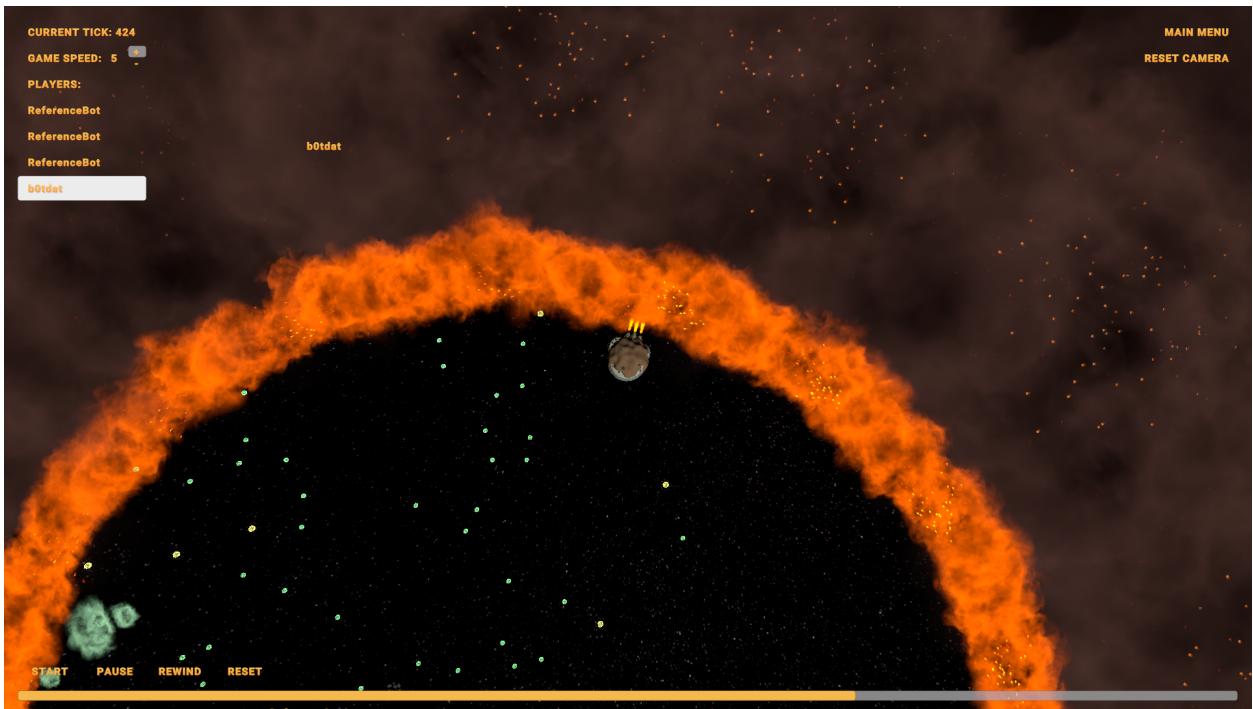


Gambar 4.1 Pengujian 1

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas merupakan peristiwa dimana bot menghindari gas cloud yang dapat mengurangi ukuran dari bot. Hal ini membuktikan bahwa algoritma *greedy by safest trajectory* bekerja dengan baik. Algoritma bekerja dengan menghitung proyeksi posisi pada next tick sesuai dengan *heading* serta kecepatannya. Apabila proyeksi posisi dari bot tidak berada pada area yang aman, bot akan melakukan pencarian arah yang lebih aman.

4.2.2 Bot Menghindari Boundary Map

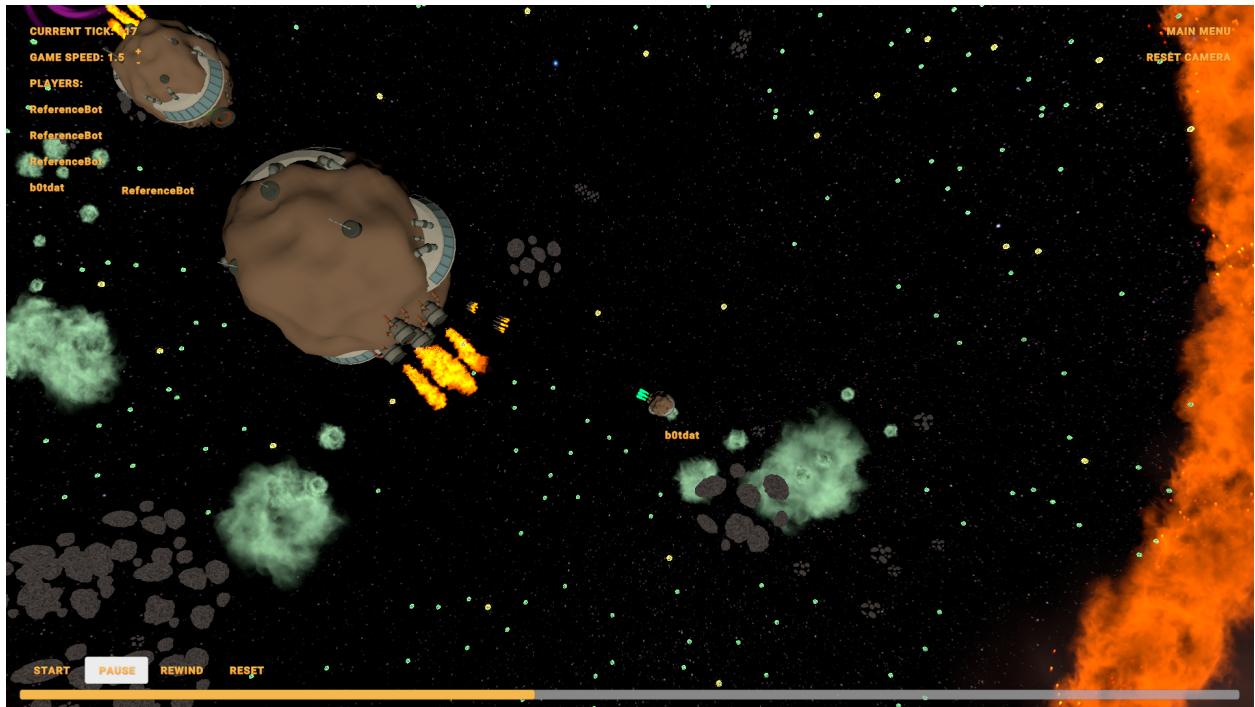


Gambar 4.2 Pengujian 2

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas merupakan peristiwa dimana bot menghindari boundary map yang dapat mengurangi ukuran bot. Hal ini membuktikan bahwa algoritma *greedy by safest place* dan *safest trajectory* bekerja dengan baik. Bot yang pada mulanya berada di area yang selanjutnya akan menjadi boundary map akan ter dorong untuk bergerak ke dalam area map. Begitu pula apabila bot sedang menuju makanan yang berada dekat dengan *boundary map*, bot akan mengambil lintasan yang tidak membuatnya bersinggungan dengan *boundary map*.

4.2.3 Bot Menghindari Lawan yang Lebih Besar



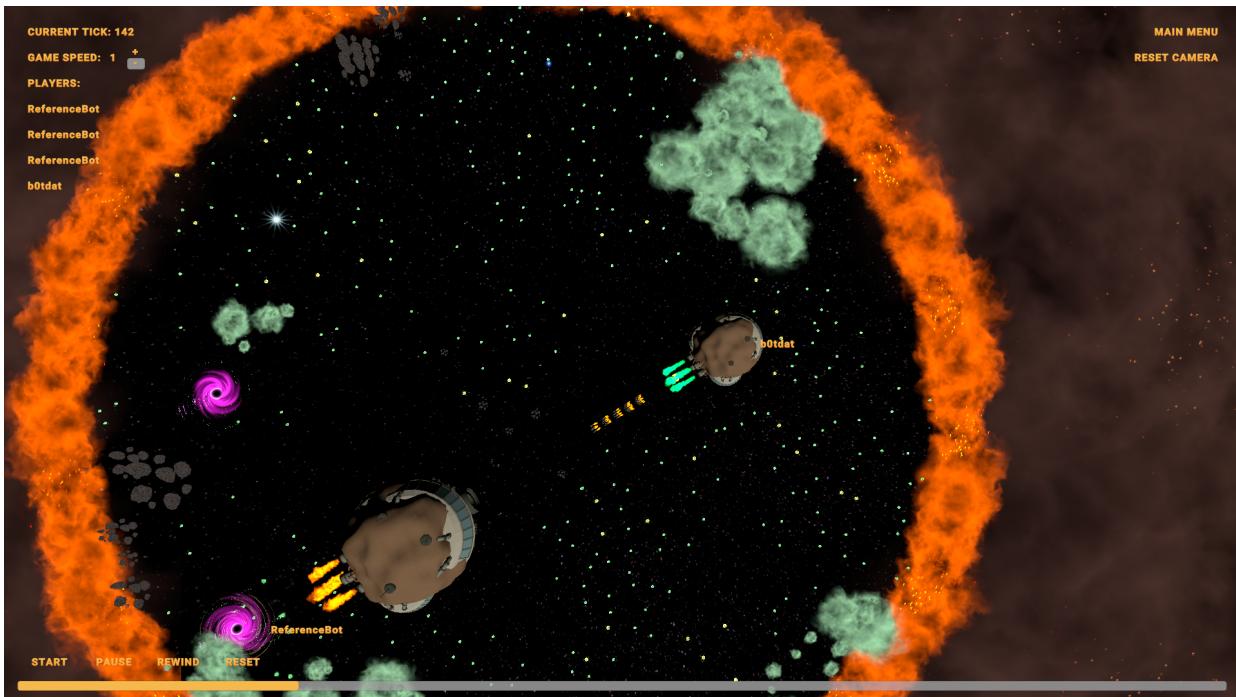
Gambar 4.3 Pengujian 3

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas merupakan peristiwa dimana bot pergi menghindari bot yang lebih besar pada jarak tertentu. Hal ini dilakukan dengan mengaktifkan afterburner yang dapat mempercepat bot untuk pergi dari area berbahaya. Adapun bot juga memilih arah yang radial dengan bot lawan yang lebih besar tersebut sehingga mempercepat waktu keluar dari zona berbahaya. Hal ini membuktikan bahwa algoritma *greedy by safest place* bekerja dengan baik.

Di sisi lain, bot juga bergerak menjauh sambil menembakkan torpedo salvo ke arah lawan yang lebih besar tersebut. Hal ini dilakukan untuk mengurangi ukuran lawan serta memperbesar ukuran bot sehingga bobot ancamannya menjadi berkurang. Peristiwa ini membuktikan bahwa algoritma *greedy by destroying opponents* juga bekerja dengan baik.

4.2.4 Bot Meluncurkan Torpedo ke Lawan yang Lebih Besar

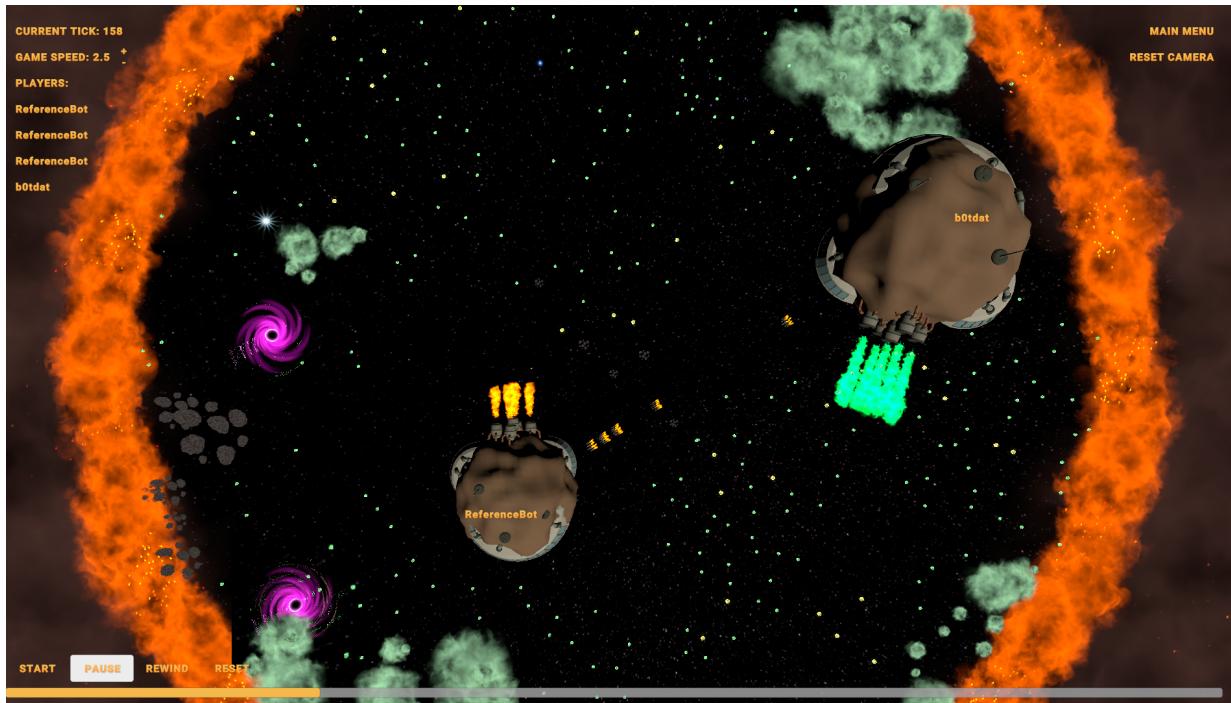


Gambar 4.4 Pengujian 4

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas memperlihatkan bot yang sedang meluncurkan torpedo salvo ke arah lawan yang lebih besar. Kejadian ini dipicu oleh perhitungan matematis yang mempertimbangkan kecepatan torpedo, jarak bot dengan lawan, ukuran lawan, dan kecepatan lawan sehingga meningkatkan peluang torpedo mengenai lawan. Selain itu, bot juga mempertimbangkan ketersediaan torpedo yang dimilikinya. Hal ini kembali menunjukkan bahwa algoritma *greedy by destroying opponents* bekerja dengan baik.

4.2.5 Bot Meluncurkan Torpedo ke Lawan yang Lebih Kecil



Gambar 4.5 Pengujian 5

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas memperlihatkan bot yang sedang meluncurkan torpedo salvo ke arah lawan yang lebih kecil. Kejadian ini dipicu oleh perhitungan matematis yang mempertimbangkan kecepatan torpedo, jarak bot dengan lawan, ukuran lawan, dan kecepatan lawan sehingga meningkatkan peluang torpedo mengenai lawan. Selain itu, bot juga mempertimbangkan ketersediaan torpedo yang dimilikinya. Hal ini kembali menunjukkan bahwa algoritma *greedy by destroying opponents* bekerja dengan baik.

4.2.6 Bot Menuju Makanan Aman Terdekat



Gambar 4.6 Pengujian 6

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas memperlihatkan bahwa bot sedang bergerak ke arah makanan yang aman dengan jarak terdekat darinya. Gerakan ini terjadi karena bot sudah berada pada area yang aman belum cukup besar untuk meluncurkan torpedo salvo ke arah lawan, serta berada dalam trajectory yang aman. Hal ini membuktikan bahwa tingkatan prioritas pada program secara keseluruhan bekerja dengan baik. Selain itu, bot berhasil memilih makanan yang aman dengan jarak terdekat darinya yang membuktikan bahwa algoritma *greedy by nearest safe food* bekerja dengan baik.

4.2.7 Bot Mengejar dan Memangsa Lawan yang Lebih Kecil



Gambar 4.7 Pengujian 7

Analisis Ketercapaian Tujuan Algoritma

Gambar di atas merupakan peristiwa bot yang mengejar lawan yang lebih kecil. Hal ini membuktikan bahwa algoritma greedy by feeding on smaller opponents bekerja dengan baik.

4.3 Ketercapaian Algoritma

Komponen Algoritma	Tercapai	Tidak Tercapai
Sub-algoritma Greedy by Safest Place	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Sub-algoritma Greedy by Destroying Opponents	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Sub-algoritma Greedy by Safest Trajectory	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Sub-algoritma Greedy by Nearest-Safe Food	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Sub-algoritma Greedy by Feeding on Smaller Bot	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Tingkatan Prioritas Algoritma	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Pemanfaatan Ability Shield	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Pemanfaatan Ability Fire Torpedoes	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Pemanfaatan Ability After Burner	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Pemanfaatan Ability Teleporter	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Pemanfaatan Ability Supernova	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Tabel 4.1 Tracing Ketercapaian Algoritma

Bab 5

Kesimpulan dan Saran

5.1 Kesimpulan

Algoritma greedy merupakan algoritma yang digunakan dalam menangani persoalan optimasi, yakni minimasi atau maksimasi. Dalam pengimplementasiannya, permasalahan yang ditangani algoritma *greedy* akan dibagi menjadi langkah-langkah persoalan yang harus dicari solusinya secara lokal dengan harap runtunan pemilihan solusi optimum lokal akan mengarah kepada solusi optimum global.

Dalam permainan Galaxio, persoalan yang ditangani adalah memenangkan permainan dengan cara menjadi *last bot standing*. Persoalan besar ini dipecah ke dalam langkah-langkah sebagaimana algoritma *greedy* memecah permasalahan. Dalam hal ini, langkah-langkah tersebut disandingkan dengan setiap tick dalam permainan. Beberapa hal yang perlu dipertimbangkan dalam mengimplementasikan algoritma *greedy* dalam permainan ini yakni objek yang dapat menguntungkan dan merugikan bot kita. Oleh karena itu, bot harus memilih strategi yang dapat memaksimasi keuntungan (pertambahan ukuran dan eliminasi musuh) dan meminimasi kerugian (menghindar dari objek yang merugikan).

Oleh karena *state* dari permainan yang tidak dapat diprediksi karena adanya bot lain yang juga memiliki berbagai strategi dalam memenangkan permainan, algoritma yang digunakan perlu mengkombinasikan strategi *greedy* dalam berbagai komponen. Akan tetapi, algoritma yang kita pakai tidak menjadi faktor tunggal dalam memenangkan permainan. Strategi algoritma yang dipakai musuh juga menjadi faktor penentu kemenangan bot dalam game.

5.2 Saran

Saran dari kelompok kami bagi para developer bot permainan Galaxio adalah:

1. Pengembang harus mempertimbangkan berbagai komponen dalam permainan untuk dapat meminimasi kerugian dan memaksimasi keuntungan di saat yang bersamaan.
2. Faktor kecepatan bot dalam mewujudkan objektifnya juga menjadi penentu kemenangan dalam permainan karena bot harus bersaing dengan bot-bot lawan.
3. Adapun dalam proses merancang dan mengimplementasikan strategi *greedy* dalam permainan Galaxio, ada baiknya pengembang memahami secara utuh komponen-komponen yang ada dalam permainan agar dapat mempertimbangkan segala faktor penentu kemenangan.
4. Hal di atas dilanjutkan dengan perancangan strategi *greedy* yang dikhususkan untuk masing-masing komponen yang dianggap penting agar dapat dikombinasikan di kemudian waktu untuk mewujudkan algoritma *greedy* yang terbaik.

DAFTAR PUSTAKA

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf)
(diakses pada 16 Februari 2022).

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-\(2021\)-Bag2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf)
(diakses pada 16 Februari 2022).

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-\(2022\)-Bag3.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf)
(diakses pada 16 Februari 2022).

<https://github.com/EntelectChallenge/2021-Galaxio/releases/tag/2021.3.2>
(diakses pada 16 Februari 2022).

<https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/game-engine/game-rules.md>
(diakses pada 16 Februari 2022).

<https://www.programiz.com/dsa/greedy-algorithm>
(diakses pada 16 Februari 2022).

LINK PENTING

Link Github Kode : https://github.com/melvinkj/Tubes1_sukatamak

Link Video : https://youtu.be/_mg5jN18BzM