Tugas Besar I IF2211 Strategi Algoritma
Semester II Tahun 2022/2023
Pemanfaatan Algoritma Greedy dalam Aplikasi Permainan
“Galaxio”

Disusun oleh:
Kelompok Solar Sailers

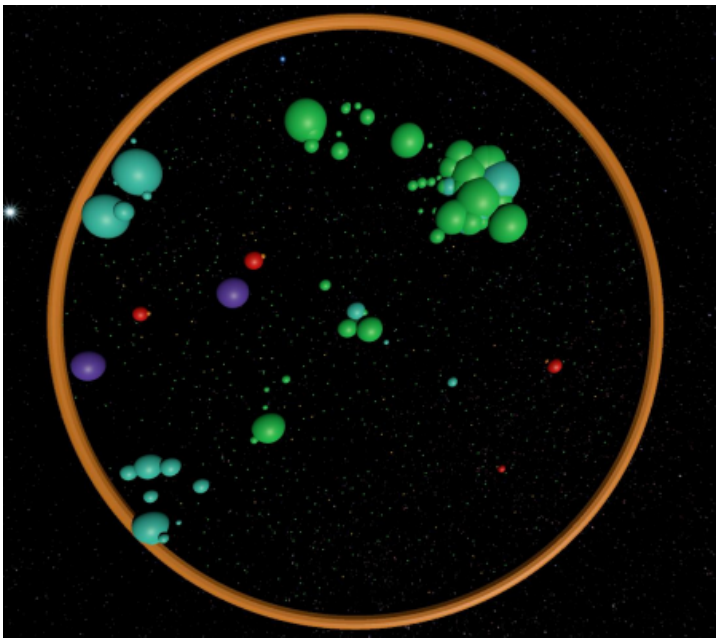
13521085 - Addin Munawwar Yusuf
13521088 - Puti Nabilla Aidira
13521130 - Althaaf Khasyi Atisomya

Program Studi Teknik Informatika
Institut Teknologi Bandung
2023

DAFTAR ISI

BAB 1
DESKRIPSI MASALAH

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

Galaxio adalah game yang diperlombakan pada Entelect Challenge yang diadakan oleh Entelect. Inti dari perlombaan ini adalah membuat sebuah bot yang dapat berinteraksi dengan game engine yang telah disediakan, dan akan diadu dengan bot lain. Pada Tugas Besar I IF2211 Strategi Algoritma ini kami akan mengimplementasikan bot kapal dalam permainan Galaxio dengan menggunakan strategi greedy untuk memenangkan permainan.

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

1. Peta permainan berbentuk kartesius yang memiliki arah positif dan negatif. Peta hanya menangani angka bulat. Kapal hanya bisa berada di integer x,y yang ada di peta. Pusat peta adalah 0,0 dan ujung dari peta merupakan radius. Jumlah ronde maximum pada game sama dengan ukuran radius. Pada peta, akan terdapat 5 objek, yaitu Players, Food, Wormholes, Gas Clouds, Asteroid Fields. Ukuran peta akan mengecil seiring batasan peta mengecil.

2. Kecepatan kapal dilambangkan dengan x. Kecepatan kapal akan dimulai dengan kecepatan 20 dan berkurang setiap ukuran kapal bertambah. Ukuran (radius) kapal akan dimulai dengan ukuran 10. Heading dari kapal dapat bergerak antar 0 hingga 359 derajat. Efek afterburner akan meningkatkan kecepatan kapal dengan faktor 2, tetapi mengecilkan ukuran kapal sebanyak 1 setiap tick. Kemudian kapal akan menerima 1 salvo charge setiap 10 tick. Setiap kapal hanya dapat menampung 5 salvo charge. Penembakan slavo torpedo (ukuran 10) mengurangkan ukuran kapal sebanyak 5.
3. Setiap objek pada lintasan punya koordinat x,y dan radius yang mendefinisikan ukuran dan bentuknya. Food akan disebarkan pada peta dengan ukuran 3 dan dapat dikonsumsikan oleh kapal player. Apabila player mengkonsumsi Food, maka Player akan bertambah ukuran yang sama dengan Food. Food memiliki peluang untuk berubah menjadi Super Food. Apabila Super Food dikonsumsi maka setiap makan Food, efeknya akan 2 kali dari Food yang dikonsumsi. Efek dari Super Food bertahan selama 5 tick.
4. Wormhole ada secara berpasangan dan memperbolehkan kapal dari player untuk memasukinya dan keluar di pasangan satu lagi. Wormhole akan bertambah besar setiap tick game hingga ukuran maximum. Ketika Wormhole dilewati, maka wormhole akan mengecil sebanyak setengah dari ukuran kapal yang melewatinya dengan syarat wormhole lebih besar dari kapal player.
5. Gas Clouds akan tersebar pada peta. Kapal dapat melewati gas cloud. Setiap kapal bertabrakan dengan gas cloud, ukuran dari kapal akan mengecil 1 setiap tick game. Saat kapal tidak lagi bertabrakan dengan gas cloud, maka efek pengurangan akan hilang.
6. Torpedo Salvo akan muncul pada peta yang berasal dari kapal lain. Torpedo Salvo berjalan dalam lintasan lurus dan dapat menghancurkan semua objek yang berada pada lintasannya. Torpedo Salvo dapat mengurangi ukuran kapal yang ditabraknya. Torpedo Salvo akan mengecil apabila bertabrakan dengan objek lain sebanyak ukuran yang dimiliki dari objek yang ditabraknya.
7. Supernova merupakan senjata yang hanya muncul satu kali pada permainan di antara quarter pertama dan quarter terakhir. Senjata ini tidak akan bertabrakan dengan objek lain pada lintasannya. Player yang menembakannya dapat meledakannya dan memberi damage ke player yang berada dalam zona. Area ledakan akan berubah menjadi gas cloud.
8. Player dapat meluncurkan teleporter pada suatu arah di peta. Teleporter tersebut bergerak dalam direksi dengan kecepatan 20 dan tidak bertabrakan dengan objek apapun. Player tersebut dapat berpindah ke tempat teleporter tersebut. Harga setiap peluncuran teleporter adalah 20. Setiap 100 tick player akan mendapatkan 1 teleporter dengan jumlah maximum adalah 10.
9. Ketika kapal player bertabrakan dengan kapal lain, maka kapal yang lebih besar akan dikonsumsi oleh kapal yang lebih kecil sebanyak 50% dari ukuran kapal yang lebih besar hingga ukuran maximum dari ukuran kapal yang lebih kecil. Hasil dari tabrakan akan mengarahkan kedua dari kapal tersebut lawan arah.

10. Terdapat beberapa command yang dapat dilakukan oleh player. Setiap tick, player hanya dapat memberikan satu command. Berikut jenis-jenis dari command yang ada dalam permainan:
    a. FORWARD
    b. STOP
    c. START_AFTERBURNER
    d. STOP_AFTERBURNER
    e. FIRE_TORPEDOES
    f. FIRE_SUPERNOVA
    g. DETONATE_SUPERNOVE
    h. FIRE_TELEPORTER
    i. TELEPORT
    j. USE_SHIELD
11. Setiap player akan memiliki score yang hanya dapat dilihat jika permainan berakhir. Score ini digunakan saat kasus tie breaking (semua kapal mati). Jika mengonsumsi kapal player lain, maka score bertambah 10, jika mengonsumsi food atau melewati wormhole, maka score bertambah 1. Pemenang permainan adalah kapal yang bertahan paling terakhir dan apabila tie breaker maka pemenang adalah kapal dengan score tertinggi.

BAB 2
LANDASAN TEORI

2.1 Algoritma *Greedy*

Algoritma *greedy* merupakan algoritma yang digunakan untuk memecahkan masalah secara langkah per langkah dengan aturan tertentu sehingga setiap langkah akan mengambil pilihan yang terbaik. Penggunaan algoritma *greedy* hanya memperhatikan kemungkinan-kemungkinan yang ada pada saat itu tanpa memperhatikan konsekuensi di masa depan serta tidak dapat melakukan *backtrack*. Harapannya, dengan memilih keputusan yang terbaik saat itu (optimum lokal) akan dicapai langkah yang paling optimum (optimum global). Algoritma *greedy* adalah metode yang sederhana dan paling populer digunakan untuk memecahkan persoalan optimasi. Persoalan optimasi ini dapat berupa persoalan maksimasi maupun minimasi.

2.2 Elemen-Elemen Algoritma Greedy

Algoritma *greedy* diterapkan dengan mencari sebuah himpunan bagian S dari himpunan kandidat C. Dengan syarat, S harus memenuhi kriteria-kriteria yang ditentukan, yakni S menyatakan suatu solusi yang dioptimisasi dengan suatu fungsi objektif. Secara detail, elemen-elemen tersebut didefinisikan sebagai berikut.

1. Himpunan kandidat, C: berisi kandidat yang akan dipilih pada setiap langkah.
2. Himpunan solusi, S: berisi kandidat yang sudah dipilih.
3. Fungsi solusi: fungsi yang menentukan apakah himpunan kandidat pilihan sudah memberikan solusi.
4. Fungsi seleksi: fungsi yang memilih kandidat berdasarkan strategi *greedy* heuristik tertentu.
5. Fungsi kelayakan: fungsi yang memeriksa apakah kandidat pilihan layak dimasukkan ke dalam himpunan solusi.
6. Fungsi objektif : fungsi optimisasi.

2.3 Cara Kerja Program

Struktur program terdiri dari *engine, runner, logger, reference-bot, starter-bots* dan *visualiser*. *Engine* bertugas untuk menjalankan aturan game dengan menerapkan *command* bot jika valid. *Runner* bertugas untuk menjalankan pertandingan antar pemain. *Runner* memanggil *command* yang sesuai dengan yang diberikan bot kemudian menyerahkannya ke

engine untuk dijalankan. *Logger* bertugas untuk mencatat semua perubahan *state* game dan menuliskannya ke file log di akhir game. *Starter-bots* adalah folder yang berisi struktur bot dengan *logic* sederhana yang dapat digunakan sebagai *starting-point* perancangan bot. *Starter-bots* menyediakan struktur bot dalam beberapa bahasa di antaranya, C++, Java, Javascript, Kotlin, C#, dan Python. Selain itu, folder *starter-bots* juga menyediakan ReferenceBot sebagai bot referensi yang dapat digunakan untuk menguji bot rancangan. *Visualiser* dapat digunakan untuk memvisualisasikan pertandingan dengan membaca file logger dari pertandingan yang sudah selesai. Visualiser disediakan untuk operasi sistem Linus, MacOS, dan Windows.

Untuk menjalankan *game*, *engine, runner, logger,* dan bot harus dijalankan secara bersamaan. Framework yang digunakan untuk menjalankan game adalah .NET versi 5.0. Tata cara menjalankan *game* adalah sebagai berikut.

1. Jalankan *runner* dengan *command* seperti berikut pada folder "runner-publish".

```
% dotnet GameRunner.dll
```

2. Jalankan *engine* dengan *command* seperti berikut pada folder "engine-publish".

```
% dotnet Engine.dll
```

3. Jalankan *logger* dengan *command* seperti berikut pada folder "logger-publish".

```
% dotnet Logger.dll
```

4. Jalankan bot dengan *command* seperti berikut pada folder "reference-bot-publish".

```
% dotnet ReferenceBot.dll //untuk menjalankan
reference bot
% java -jar "<path_to_jar>" //untuk menjalankan bot
rancangan dalam bentuk java jar
```

Untuk sistem operasi berbasis Unix, dapat menggunakan *script* "run.sh" yang disediakan pada *starter-pack* program.

Implementasi algoritma pada bot dapat dilakukan menggunakan *starter-bot* sesuai bahasa yang dipilih. *Starter-bot* sudah berisi enumerasi tipe objek game dan aksi *player*, model untuk objek *game*, state *game*, aksi *player*, posisi, dan *world*, bot *service*, serta sebuah file *main*.

File *main* pada *starter-bot* telah mengimplementasikan signalR, koneksi, dan registrasi dengan *runner*. Untuk menghubungkan bot dengan *runner* dibutuhkan IP address dari *environement*. Jika ingin menggunakan *cloud infrastructure*, dibutuhkan *environment variable* "RUNNER_IPV4". Jika ingin menggunalan mesin lokal, gunakan localhost dengan *port* 5000 (*default*). Selanjutnya, *build* koneksi ke hub, hub akan selalu berada pada *port* 5000/runnerhub. Kemudian, hub akan mengumumkan *event-event* yang harus didengarkan oleh seluruh bot, di antaranya: “Registered”, “Disconnect”, “ReceiveGameState”, “ReceivePlayerConsumed” (opsional), “ReceiveGameComplete” (opsional). Setelah seluruh *listener* terdaftar ke koneksi, koneksi ke SignalR Hub milk *runner* dapat dimulai. Koneksi ini akan mengizinkan *runner* untuk berkomunikasi dengan bot serta memungkinkan pengiriman aksi. Setelah koneksi terbentuk, dibutuhkan token untuk meregistrasi *nickname* bot. Pada pertandingan yang menggunakan *cloud infrastructure,* token digunakan untuk keperluan *security*. Tetapi, jika menggunakan mesin lokal, token dapat di-*generate* dengan GUID. Kemudian, gunakan token dengan mengirimkan pesan “Register” serta *nickname* ke *runner* melalui koneksi.

Selanjutnya, setelah semua bot teregistrasi, *game state* “Tick 0” akan diumumkan ke *listener* “ReceiveGameState”. Kemudian, *engine* akan berhenti selama 5 detik sebelum *game* dimulai. Setelah game dimulai, bot dapat mengirimkan aksi *player* berupa objek playerAction dengan mengirimkan pesan "SendPlayerAction" (atau "InvokePlayerAction", tergantung *client library* yang dipakai) ke *runner*. Aksi player yang dikirimkan ditentukan dari hasil fungsi computeNextPlayerAction pada BotService yang sudah disediakan pada *starter-bot.* Sehingga, implementasi algoritma *greedy* dapat dilakukan/dipanggil di dalam fungsi computeNextPlayerAction ini. Setelah seluruh algoritma selesai diimplementasikan, untuk bahasa java, *build* file jar.

BAB 3
APLIKASI STRATEGI GREEDY

3.1 Mapping Persoalan *Galaxio* Menjadi Elemen-Elemen Algoritma Greedy

3.1.1 Mapping *Galaxio* secara Umum

Tabel 1. Mapping *Galaxio* secara Umum

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan aksi pemain berupa jenis (FORWARD, STOP, START_AFTERBURNER , STOP_AFTERBURNER , FIRE_TORPEDOES , FIRE_SUPERNOVA , DETONATE_SUPERNOVA , FIRE_TELEPORTER , TELEPORT) dan headingnya
Himpunan Solusi	Aksi pemain terpilih untuk tick ini
Fungsi Solusi	Memeriksa apakah aksi yang dipilih berupa aksi yang valid dan layak (berdasarkan fungsi kelayakan).
Fungsi Seleksi	Pilih aksi yang paling menguntukan saat ini, yaitu aksi yang menuju skor tertinggi dan menghindari kematian (dijabarkan lebih lanjut pada fungsi seleksi masing-masing sub persoalan).
Fungsi Kelayakan	Memeriksa apakah aksi layak dilakukan berdasarkan fungsi kelayakan (dijabarkan lebih lanjut pada fungsi seleksi masing-masing sub persoalan).
Fungsi Objektif	Memaksimalkan skor pada pertandingan dan bertahan selama mungkin.

3.1.1 Mapping Persoalan Pemilihan *Choose State*

Karena persoalan *Galaxio* diselesaikan dengan memecahnya menjadi sub-sub persoalan berupa state-state aksi, dibutuhkan strategi tertentu untuk memilih state yang aksinya akan dilakukan pada tick ini. Strategi pemilihan dilakukan dengan pemilihan *state* dengan skor prioritas tertinggi pada *state* permainan terkini.

Tabel 2. Mapping Persoalan Pemilihan *Choose State*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan *state* sub persoalan (*GatherFood, FireTeleport, Teleport, TorpedoAttack, UseShield, Run*)
Himpunan Solusi	Pilihan state sub persoalan
Fungsi Solusi	Memeriksa apakah pilihan sudah berupa pilihan state dengan aksi yang layak (dicek dengan fungsi kelayakan)
Fungsi Seleksi	Pilih state sub persoalan dengan *priority score* terbesar
Fungsi Kelayakan	Cek apakah aksi pada *state* yang dipilih layak (fungsi kelayakan dijabarkan lebih lanjut pada fungsi seleksi masing-masing sub persoalan)
Fungsi Objektif	*State* sub persoalan yang dipilih paling menguntungkan sesuai objektif *game* (menuju skor maksimum dan bertahan selama mungkin) berdasarkan state game saat ini.

3.1.2 Mapping Persoalan *GatherFood*

Tabel 3. Mapping Persoalan *GatherFood*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan aksi untuk melakukan getClosestFood, getNewArea, dodgeBoundary, dodgeGasCloud
Himpunan Solusi	Aksi yang dipilih (getClosestFood, goToNewArea, dodgeBoundary, dodgeGasClod)
Fungsi Solusi	Memeriksa apakah aksi yang dipilih valid dan layak untuk gamestate saat ini.
Fungsi Seleksi	Apabila ukuran bot >= 15 atau terdapat food dengan jarak yang sangat dekat dengan dengan bot atau terdapat >= 5 food pada jarak dekat dengan bot maka akan memilih getclosestFood, jika tidak memenuhi kondisi tersebut maka bot akan

	pergi ke area baru yang memiliki lebih banyak makanan (goToNewArea). Namun, apabila disekitar bot terdapat gas cloud (dodgeGasCloud), maka bot akan menghindar dan apabila jarak bot dengan boundary dekat maka bot akan menghindari boundary (dodgeBoundary).
Fungsi Kelayakan	GetClosestFood dan goToNewArea dinyatakan layak apabila masih terdapat food di world.
Fungsi Objektif	Mengambil makanan sebanyak-banyaknya dan menghindari objek rintangan.

3.1.3 Mapping Persoalan *FireTeleport*

Persoalan penembakan teleporter adalah salah satu sub permasalahan dalam persoalan *Galaxio*. Pemain dapat menembakan teleporter dengan harga 20 ukuran dirinya. Teleporter kemudian akan bergerak ke arah tertentu dalam kecepatan 20 dan tidak menabrak apapun.

Tabel 4. Mapping Persoalan *FireTeleport*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan posisi target-target penembakan teleporter dan pilihan tidak memilih target.
Himpunan Solusi	Posisi target yang dipilih atau tidak memilih target.
Fungsi Solusi	Memeriksa apakah target yang dipilih berupa posisi valid atau tidak memilih target.
Fungsi Seleksi	Pilihlah target *enemy* yang berukuran paling kecil.
Fungsi Kelayakan	Memeriksa apakah target yang dipilih merupakan diri sendiri, berada di luar *boundary map*, atau berukuran lebih besar dari ukuran bot dikurangi biaya penembakan (20), memeriksa apakah bot memiliki teleporter tersedia, dan memeriksa apakah ukuran bot aman untuk melakukan penembakan.

Fungsi Objektif	*Enemy* dapat dikonsumsi habis setelah melakukan teleportasi (ukuran *enemy* minimum).

3.1.4 Mapping Persoalan *Teleport*

Teleport adalah salah satu sub permasalahan dalam persoalan *Galaxio*. Pemain dapat melakukan teleportasi dengan aksi "Teleport" ke teleporter miliknya yang sudah ditembakan.

Tabel 5. Mapping Persoalan *Teleport*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan aksi melakukan teleportasi atau tidak.
Himpunan Solusi	Pilihan untuk melakukan teleportasi atau tidak.
Fungsi Solusi	Memeriksa apakah solusi berupa pilihan untuk melakukan teleportasi (dan layak, memiliki teleporter yang sudah ditembakkan) atau tidak melakukan teleportasi.
Fungsi Seleksi	Pilih lakukan teleportasi jika teleporter milik bot berada di dekat (pada radius 100 + ukuran bot saat ini) *enemy* yang cukup kecil (kurang dari ukuran bot saat ini).
Fungsi Kelayakan	Memeriksa apakah bot memiliki teleporter yang telah ditembakkan sehingga kayak memilih melakukan teleportasi.
Fungsi Objektif	Teleportasi dilakukan tepat ketika berada didekat *enemy* yang dapat dikonsumsi.

3.1.5 Mapping Persoalan *TorpedoAttack*

Sub permasalahan lain pada permainan *Galaxio* adalah persoalan menembakan *torpedo salvo*. *Torpedo salvo* adalah senjata yang ditembakan oleh pemain pada arah tertentu kemudian bergerak dalam garis lurus pada arah tersebut. Torpedo bergerak dalam kecepatan 60 dan berukuran 10 sesaat setelah ditembakan. Torpedo akan menabrak objek apapun dalam *map* (kecuali *wormhole*), yang akan mengurangi ukurannya. Torpedo akan mengurangi

ukuran pemain yang ditabrak dengan ukuran torpedo saat itu kemudian mentransfer ukuran tersebut ke pemain yang menembak. Jika pemain menembakan torpedo saat berukuran kurang dari 10, pemain tersebut akan ter-*self-destruct*.

Tabel 6. Mapping Persoalan *TorpedoAttack*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan posisi target-target penembakan torpedo dan pilihan tidak memilih target.
Himpunan Solusi	Posisi target yang dipilih, atau tidak memiliki target.
Fungsi Solusi	Memeriksa apakah target yang dipilih berupa posisi valid atau tidak memilih target.
Fungsi Seleksi	Pilihlah target *enemy* yang berada cukup dekat dengan bot (minimal dalam radius 150).
Fungsi Kelayakan	Memeriksa apakah bot memiliki *torpedo salvo* tersedia dan memeriksa apakah ukuran bot aman untuk melakukan penembakan.
Fungsi Objektif	Menembak *enemy* terdekat agar penembakan tepat sasaran dan *enemy* sulit mengelak.

3.1.6 Mapping Persoalan *UseShield*

Salah satu sub permasalahan yang ada pada permainan *Galaxio* adalah persoalan mengaktifkan perisai/ *shield*. Perisai/ *shield* adalah salah satu mekanisme bertahan yang dapat digunakan untuk menangkis dan memantulkan torpedo dari musuh. Torpedo yang terpantul oleh perisai akan bergerak ke arah yang berlawanan, sedangkan pemain akan terdorong sesuai arah tembakan torpedo. Perisai aktif selama 20 tick dan akan mengonsumsi ukuran pemain sebesar 20, serta memiliki *cooldown* selama 20 tick juga.

Tabel 7. Mapping Persoalan *UseShield*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Pilihan untuk menggunakan shield atau tidak pada sebuah tick.

Himpunan Solusi	Salah satu pilihan untuk menggunakan shield atau tidak pada sebuah tick.
Fungsi Solusi	Menjalankan aksi menggunakan perisai dengan tepat (memiliki perisai dan terdapat torpedo yang mengarah ke *bot*).
Fungsi Seleksi	Menggunakan shield ketika terdapat torpedo pada jarak dekat dari bot (jarak <= 80)
Fungsi Kelayakan	Memeriksa apakah *bot* memiliki perisai, ukuran bot aman untuk menggunakan perisai, dan perisai tidak aktif di bot.
Fungsi Objektif	Mencari waktu penggunaan shield yang paling bagus tanpa menimbulkan kerugian akibat pengurangan ukuran *bot.*

3.1.7 Mapping Persoalan *Run*

Mekanisme pertahanan lain yang dapat dilakukan adalah *run*, atau berlari. Berlari yang dimaksudkan disini adalah menghindar dari ancaman, seperti musuh yang lebih besar, proyektil teleporter, ataupun proyektil supernova, yang tidak bisa dihindari dengan menggunakan perisai.

Tabel 8. Mapping Persoalan *Run*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Permutasi dari segala kemungkinan arah pemain untuk bergerak maju.
Himpunan Solusi	Arah gerakan pemain dan aksi untuk maju.
Fungsi Solusi	Menjalankan bot ke arah yang menjauhi ancaman
Fungsi Seleksi	Memilih arah yang menjauhi ancaman yang terdekat dengan sudut 90 untuk ancaman berupa tembakan atau 180 derajat berupa musuh
Fungsi Kelayakan	Memeriksa apakah ada ancaman berupa musuh yang lebih besar, proyektil teleport, atau proyektil supernova yang berdada pada jarak 150 dari bot.
Fungsi Objektif	Mencari arah lari yang tepat untuk dapat

	menghindari ancaman tepat waktu.

3.1.8 Mapping Persoalan *PickUpSupernova*

Tabel 9. Mapping Persoalan *PickUpSupernova*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan posisi *pick up* dan pilihan untuk tidak mem-*pick up.*
Himpunan Solusi	Pilihan posisi *pick up* atau pilihan untuk tidak mem-*pick up.*
Fungsi Solusi	Memeriksa apakah posisi yang dipilih merupakan posisi valid pada peta dan layak untuk *game state* saat ini atau berupa pilihan tidak mem-*pick up*.
Fungsi Seleksi	Jika supernova tersedia, pilih posisi *pick up* dengan jarak <= 200 dari posisi bot. Jika tidak, pilih untuk tidak mem-*pick up*.
Fungsi Kelayakan	Jika memilih posisi pick-up, memeriksa apakah terdapat supernova tersedia.
Fungsi Objektif	Mem-*pick up* supernova dengan jarak cukup dekat sesaat setelah supernova tersedia.

3.1.9 Mapping Persoalan *FireSupernova*

Tabel 10. Mapping Persoalan *FireSupernova*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan posisi target-target penembakan supernova.
Himpunan Solusi	Posisi target penembakan.
Fungsi Solusi	Memeriksa apakah posisi yang dipilih merupakan posisi valid pada peta dan layak untuk *game state* saat ini.
Fungsi Seleksi	Pilihlah target *enemy* yang terbesar.
Fungsi Kelayakan	Memeriksa apakah supernova yang sudah di-*pick up* tersedia.
Fungsi Objektif	Memaksimalkan efek supernova dengan menembak ke target berukuran maksimum.

3.1.10 Mapping Persoalan *DetonateSupernova*

Tabel 11. Mapping Persoalan *DetonateSupernova*

Nama Elemen	Definisi Elemen
Himpunan Kandidat	Himpunan pilihan meledakan supernova atau tidak.
Himpunan Solusi	Pilihan untuk meledakan supernova saat ini atau tidak.
Fungsi Solusi	Memeriksa apakah pilihan berupa aksi meledakan supernova atau tidak yang valid dan layak.
Fungsi Seleksi	Pilih aksi meledakan jika supernova berada cukup jauh dari bot (jarak >= 170) dan cukup dekat dengan *enemy* terbesar (jarak <= 70) atau supernova berada pada jarak dekat dari *boundary map* (jarak <= 50).
Fungsi Kelayakan	Memeriksa apakah supernova yang sudah ditembak tersedia.
Fungsi Objektif	Memaksimalkan efek supernova dengan meledakannya saat sudah berada cukup dekat dengan target berukuran maksimum dan menghindari efek sia-sia jika supernova sudah akan keluar dari *boundary map*.

3.2 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas

3.2.1 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *Choose State*

Tabel 12. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *Choose State*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Memilih *state* yang paling menguntungkan dengan memilih *priority score t*ertinggi, *priority score* masing-masing *state* dihitung secara internal pada kelas *state* tersebut.	Efisien dan efektif karena urutan prioritas menjadi dinamis dan paling relevan dengan *state game* saat ini.
Memilih state yang paling murah secara *cost*.	Tidak efektif dan efisien karena state dengan *cost* paling murah belum tentu paling menguntungkan. Contoh: bisa saja *cost* mahal tetapi

	setelah melakukan aksi akan mendapatkan keuntungan yang sangat besar.
Memilih state dengan urutan prioritas tertentu yang tetap.	Tidak efektif dan efisien karena prioritas bisa jadi tidak relevan dengan *state game* saat ini.

3.2.2 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *GatherFood*

Tabel 13. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *GatherFood*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Langsung pergi ke area yang paling ramai dengan food.	Strategi ini tidak efektif karena apabila pada awal permainan bot langsung mencari area kaya akan makanan, ukuran bot dapat tertinggal jauh dengan bot lain.
Hanya memakan food yang memiliki jarak paling dekat dengan bot.	Strategi ini kurang efektif karena apabila selalu memakan food yang paling dekat, bot dapat masuk ke area dengan sedikit makanan.
Memakan makanan dengan jarak terdekat apabila ukuran bot masih kecil atau terdapat food dengan jarak yang dekat dengan bot dan pergi ke area dengan banyak food apabila tidak terdapat food dengan jarak sedang dengan bot.	Strategi ini efektif karena bot tidak akan tertinggal ukuran dengan bot lain pada awal permainan serta bot kemungkinan besar selalu berada di daerah dengan makanan yang banyak.

3.2.3 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *FireTeleport*

Tabel 14. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *FireTeleport*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Menembakan teleporter untuk teleportasi ke lokasi dengan food yang banyak.	Strategi ini tidak efektif karena pada dasarnya food tersebar hampir merata di seluruh peta, sehingga tidak dibutuhkan teleporter untuk mencapai tempat yang jauh.

Menembakan teleporter untuk berlindung dari tabrakan enemy, teleporter projectile, dan supernova projectile.	Strategi ini kurang efektif karena tabrakan enemy, teleporter projectile, dan supernova projectile butuh aksi penghindaran yang cepat, sedangkan menembakan teleporter butuh waktu yang cukup lama sebelum dapat digunakan.
Menembakan teleporter ke arah *enemy* yang terkecil dan lebih kecil dari ukuran bot - biaya penembakan pada arah tanpa *offset*.	Strategi ini cukup efektif karena dapat mengonsumsi *enemy* yang kecil sampai habis (mengurangi jumlah player) setelah melakukan teleportasi. Pemilihan *enemy* terkecil juga membantu mempertahanan status *enemy* agar tetap lebih kecil dari bot ketika teleporter siap digunakan. Tetapi, ukuran *enemy* yang kecil juga menyebabkan kecepatan *enemy* yang tinggi, sehingga ada kemungkinan player berhasil kabur sebelum teleporter siap digunakan.
Menembakan teleporter ke arah enemy yang terkecil dan lebih kecil dari ukuran bot - biaya penembakan pada arah dengan *offset*.	Strategi dengan tambahan *offset* ini lebih efektif karena dapat memperkecil kemungkinan *enemy* kabur sebelum teleporter siap dipakai dengan memperhitungkan arah *enemy* saat ini dan posisi kuadrannya.
Menembakan teleporter ke arah *enemy* yang terbesar dan lebih kecil dari ukuran bot - biaya penembakan.	Strategi ini cukup efektif karena dapat mengonsumsi ukuran *enemy* yang cukup besar tetapi lebih kecil dari bot (menambah ukuran dengan nilai yang cukup besar) setelah melakukan teleportasi. *Enemy* berukuran besar juga memiliki kecepatan yang lebih rendah sehingga kemungkinan kabur lebih rendah. Tetapi, pemilihan *enemy* terbesar berisiko jika sebelum teleporter siap digunakan ukuran *enemy* menjadi lebih besar dari ukuran bot.

3.2.4 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *Teleport*

Tabel 15. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *Teleport*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Teleport ketika teleporter mencapai target awal.	Tidak efektif dan efisien karena target awal belum tentu masih relevan untuk state *game saat* ini. Tidak efisien secara memori karena dibutuhkan penyimpanan target awal.
Teleport ketika teleporter berada di sekitar *enemy* yang lebih kecil dari ukuran bot saat ini.	Efektif dan efisien karena target di-*update* berdasarkan yang paling menguntungkan pada *game state* saat ini. Efisien secara memori karena tidak dibutuhkan penyimpanan target awal.

3.2.5 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *TorpedoAttack*

Tabel 16. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *TorpedoAttack*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Torpedo ditembak ke arah *enemy* yang jauh.	Tidak efektif dan efisien karena besar kemungkinan meleset.
Torpedo ditembak ke arah *enemy* yang dekat.	Efektif dan efisien karena kemungkinan meleset kecil dan *enemy* tidak dapat kabur.
Torpedo ditembak ke arah *enemy* yang kecil.	Cukup efektif dalam membunuh *enemy* karena *enemy* kecil kemungkinan besar akan mati setelah tertembak. Tetapi, *enemy* kecil sangat mudah kabur sehingga torpedo kemungkinan besar meleset dan menjadi tidak efektif (sia-sia).

3.2.6 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *UseShield*

Tabel 17. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *UseShield*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Menggunakan shield ketika banyak (lebih dari sama dengan 3) torpedo pada jarak < 80	Kurang efektif karena pada pengujian, shield sering kali telat untuk diaktifkan sehingga bot terkena banyak hit.
Menggunakan shield ketika ada (minimal 1) torpedo yang mengarah ke bot pada jarak dekat (jarak < 80)	Kurang efektif, karena *cost* menggunakan shield lebih besar ketimbang terkena *hit* dari 1 torpedo.
Menggunakan shield ketika ada (minimal 1) torpedo yang mengarah ke bot pada jarak < 150	Masih kurang efektif, karena jika torpedo hanya 1, *cost* yang digunakan untuk mengaktifkan shield
Menggunakan shield ketika banyak (lebih dari sama dengan 3) torpedo pada jarak < 150	Efektif, karena *cost* untuk menggunakan shield lebih kecil ketimbang terkena hit oleh 3 torpedo. Pada jarak ini, deteksi juga lebih akurat untuk mendeteksi torpedo yang beruntun.
Mendeteksi torpedo yang mengarah langsung ke bot.	Tidak efektif, karena bot terus bergerak, sehingga torpedo tidak akan terdeteksi jika hanya melihat torpedo yang mengarah langsung ke bot.
Mendeteksi torpedo yang mengarah ke bot dengan toleransi derajat yang sebanding dengan ukuran bot.	Efektif, karena bot dapat lebih baik mendeteksi torpedo yang mengarah kepada bot, sesuai dengan ukurannya.

3.2.7 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *Run*

Tabel 18. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *Run*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Berlari ketika terdapat torpedo mengarah ke bot	Kurang efektif, karena torpedo sudah bisa ditangani dengan menggunakan shield.
Menghindari ancaman dengan berlari 180 derajat dari arah *heading* ke ancaman	Tidak efektif untuk semua jenis ancaman, terutama untuk ancaman yang tipenya tembakan. Jika bot hanya lari menjauh, proyektil

	tembakan akan tetap dapat mengenai bot karena bot hanya bergerak lurus.
Menghindari ancaman dengan berlari 90 derajat dari arah ancaman datang	Belum efektif untuk semua jenis ancaman, terutama untuk ancaman berupa musuh yang lebih besar.
Menghindari ancaman dengan berlari 90 derajat untuk ancaman tembakan dan 180 derajat untuk ancaman musuh	Efektif, karena dengan demikian, setiap ancaman dapat dihindari dengan optimum.
Berlari menembus gas cloud dan boundary	Tidak efektif, karena tingginya *cost* yang harus dibayar ketika melewati gas cloud dan boundary.
Berlari dengan menghindari gas cloud dan boundary	Kurang efektif, akan tetapi masih menjadi alternatif yang lebih baik dari sebelumnya. Pada kasus tertentu, bot bisa jadi berada dalam kondisi terpojok dan hanya bergerak bolak-balik antara musuh dengan *gas cloud* / *boundary*.
Berlari dengan menggunakan afterburner	Cukup efektif, akan tetapi tidak efisien karena *cost* afterburner yang tinggi. Selain itu, menghentikan afterburner juga menjadi persoalan yang sulit diimplementasikan.

3.2.8 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *PickUpSupernova*

Tabel 19. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *PickUpSupernova*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Mem-*pick up* supernova dalam jarak apapun.	Kurang efektif, karena kemungkinan besar pemain lain yang lebih dekat akan berhasil mem-*pick up* terlebih dahulu.
Mem-*pick up* supernova dalam radius jarak tertentu yang cukup dekat.	Cukup efektif, dengan mem-*pick up* supernova hanya dalam radius jarak tertentu yang cukup dekat, bot dapat fokus melakukan aksi lain saat supernova berada terlalu jauh (kemungkinan akan berhasil diambil pemain lain).

3.2.9 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan *FireSupernova*

Tabel 20. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *FireSupernova*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Menembakan supernova pada lokasi dengan jumlah *enemy* yang paling banyak.	Kurang efisien dari segi komputasi karena membutuhkan perhitungan yang lebih rumit untuk menentukan posisi eksak lokasi target. Selain itu, pergerakan sekelompok *enemy* juga cenderung lebih dinamis sehingga relevansi efektivitas supernova setelah sampai pada target kemungkinan besar berkurang.
Menembakan supernova pada lokasi *enemy* terbesar.	Cukup efisien dari segi komputasi karena sederhana. Pergerakan dan perubahan ukuran satu *enemy* juga cenderung tidak sedinamis pergerakan dan perubahan ukuran sekelompok *enemy*.
Menembakan supernova pada lokasi dengan akumulasi ukuran *enemy* terbesar.	Kurang efisien dari segi komputasi karena membutuhkan perhitungan yang lebih rumit untuk menentukan posisi eksak lokasi target. Selain itu, pergerakan dan perubahan ukuran sekelompok *enemy* juga cenderung lebih dinamis sehingga relevansi efektivitas supernova setelah sampai pada target kemungkinan besar berkurang.

3.2.10 Eksplorasi Alternatif Solusi serta Analisis Efisiensi dan Efektivitas untuk Persoalan DetonateSupernova

Tabel 21. Alternatif Solusi serta Analisis Efisiensi dan Efektivitas Persoalan *DetonateSupernova*

Alternatif Solusi	Analisis Efisiensi dan Efektivitas
Detonate ketika supernova mencapai posisi target awal.	Tidak efektif dan efisien karena target awal belum tentu masih relevan untuk state *game saat* ini. Tidak efisien secara memori karena dibutuhkan penyimpanan target awal.
Detonate ketika teleporter berada cukup jauh dari bot, berada di sekitar *enemy* terbesar, atau akan keluar dari *boundary map*.	Efektif dan efisien karena target di-*update* berdasarkan yang paling menguntungkan pada *game state* saat ini. Efisien secara memori karena tidak dibutuhkan penyimpanan target awal. Selain itu, strategi ini juga efektif untuk menghindari risiko terkena efek supernova dan kesia-siaan supernova (keluar dari *boundary map)*

3.3 Strategi *Greedy* yang Dipilih dan Pertimbangan Pemilihan

3.3.1 Strategi Greedy yang Dipilih untuk Persoalan *Choose State*

Strategi *greedy* yang dipilih untuk sub persoalan *Choose State* adalah memilih *state* dengan skor prioritas tertinggi dengan perhitungan skor dilakukan secara internal pada masing-masing *state*. Perhitungan skor pada masing-masing state dilakukan sesuai strategi greedy yang diimplementasikan pada state tersebut. Hal ini mempertimbangkan efisiensi dan efektifitas strategi karena dapat memilih prioritas sesuai urutan yang dinamis dan paling relevan dengan *state game* saat ini.

3.3.2 Strategi Greedy yang Dipilih untuk Persoalan *GatherFood*

Strategi greedy yang dipilih untuk sub persoalan GatherFood adalah mencari makanan sebanyak-banyaknya sambil menuju area yang memiliki banyak food serta menghindari objek yang dapat merugikan bot. Secara umum state gatherfood memiliki 4 aksi yaitu mengambil makanan terdekat (getClosestFood), pergi ke area dengan banyak makanan (goToNewArea), menghindari boundary (dodgeBoundary), dan menghindari gas clouds (dodgeGasCloud).

Pengambilan makanan terdekat dilakukan apabila ukuran bot memiliki ukuran yang masih kecil atau terdapat food dengan jarak yang sqngat dekat

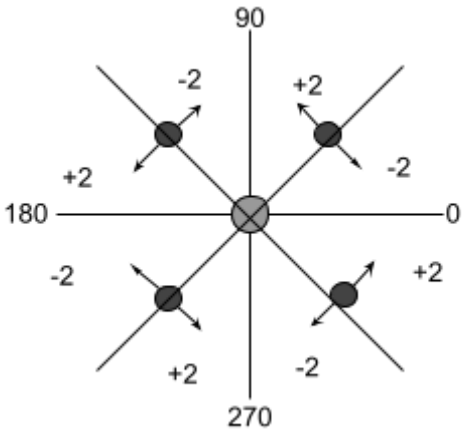
dengan bot. Hal ini dilakukan dengan pertimbangan bahwa apabila bot masih kecil bot harus fokus terlebih dahulu untuk memperbesar ukurannya serta apabila terdapat jarak food dengan jarak dekat yang sangat dengan bot tidak masalah untuk memakan food tersebut terlebih dahulu karena tidak butuh waktu yang lama untuk mencapai food tersebut.

Pergi ke area dengan banyak makanan dilakukan apabila ukuran bot sudah mulai besar atau makanan di sekitar bot mulai sedikit. Hal ini dilakukan dengan pertimbangan apabila ukuran bot sudah cukup besar bot dapat mencangkup area yang lebih besar pula sehingga tidak perlu menghampiri food satu persatu. Pergi ke area baru diimplementasikan dengan membagi area di sekitar bot menjadi 6 kuadran, bot akan menuju area yang memiliki food paling banyak dan paling sedikit memiliki rintangan.

Menghindari boundary dan menghindari gas clouds dilakukan apabila bot memiliki jarak yang dekat dengan boundary maupun gascloud. Hal ini dilakukan dengan mengubah arah bot menjadi +180 derajat arah bot dengan rintangan. Penghindaran boundary dan gasclouds ini dilakukan agar bot tidak bersentuhan dengan gasclouds dan boundary yang dapat megurangi ukuran bot.

3.3.3 Strategi Greedy yang Dipilih untuk Persoalan *FireTeleport*

Strategi *greedy* yang dipilih untuk sub persoalan *FireTeleport* adalah menembakan teleporter ke arah *enemy* yang terkecil dan lebih kecil dari ukuran bot - biaya penembakan dengan *offset*. *Offset* yang digunakan dihitung dari posisi target di salah satu dari 4 kuadran relatif terhadap *heading* bot, *offset* kemudian diberi nilai +2 atau -2, tergantung arah target pada kuadran tersebut.



Gambar 1. Ilustrasi perhitungan *heading offset* berdasarkan kuadran dan *heading* target saat ini.

Hal ini mempertimbangkan efektivitas keuntungan karena akan dapat mengonsumsi *enemy* yang kecil sampai habis (mengurangi jumlah player)

setelah melakukan teleportasi. Strategi ini juga membantu mempertahanan status *enemy* agar tetap lebih kecil dari bot ketika teleporter siap digunakan. Meskipun ukuran *enemy* yang kecil juga menyebabkan kecepatan player yang tinggi sehingga memiliki potensi kabur yang lebih besar, hal ini dapat diatasi dengan pemilihan waktu teleportasi yang menyesuaikan saat ketika *enemy* kecil berada di sekitar teleporter. Harapannya, *enemy* yang kabur masih berada dalam arah tembak *teleporter*. Pemilihan strategi ini, juga menghindari risiko jika sebelum *teleporter* siap digunakan ukuran *enemy* menjadi lebih besar dari ukuran bot (penembakan teleporter sia-sia). Tambahan *offset* juga membuat strategi lebih efektif karena dapat memperkecil kemungkinan *enemy* kabur sebelum teleporter siap dipakai dengan memperhitungkan arah *enemy* saat ini dan posisi kuadrannya.

3.3.4 Strategi Greedy yang Dipilih untuk Persoalan *Teleport*

Strategi *greedy* yang dipilih untuk sub persoalan *Teleport* adalah melakukan aksi *Teleport* ketika teleporter berada di sekitar player yang lebih kecil dari ukuran bot saat ini. Hal ini mempertimbangkan efektivitas dan efisiensi karena target di-*update* berdasarkan yang paling menguntungkan pada *game state* saat ini. Strategi ini juga efisien secara memori karena tidak dibutuhkan penyimpanan target awal. Strategi ini juga menghindari risiko melakukan aksi *Teleport* saat teleporter berada di area berbahaya (terdapat *enemy* yang lebih besar).

3.3.5 Strategi Greedy yang Dipilih untuk Persoalan *TorpedoAttack*

Strategi *greedy* yang dipilih untuk sub persoalan *TorpedoAttack* adalah menembak ke arah musuh yang dekat. Dari himpunan musuh yang dekat tersebut, dipilih yang terdekat dari bot untuk menjadi target penembakan. Hal ini mempertimbangkan efektivitas ketepatan target. Dengan memilih musuh yang dekat, kemungkinan tembakan meleset menjadi kecil.

3.3.6 Strategi Greedy yang Dipilih untuk Persoalan *UseShield*

Strategi *greedy* yang dipilih untuk sub persoalan *UseShield* adalah menggunakan *shield* ketika terdapat banyak torpedo (lebih dari 3) yang mengarah ke bot. Indikasi torpedo yang mengarah ke bot didapatkan dengan

membandingkan sudut antara bot dan torpedo dengan sudut datang torpedo itu sendiri. Derajat tersebut kemudian diperhitungkan berdasarkan toleransi sudut tertentu (dihitung dari ukuran bot dengan rasio tertentu) untuk menentukan apakah torpedo tersebut mengarah ke bot atau tidak. Jika iya, maka skor prioritas state akan ditinggikan sehingga aksi *UseShield* dapat dijalankan.

3.3.7 Strategi Greedy yang Dipilih untuk Persoalan *Run*

Strategi *greedy* yang dipilih untuk sub persoalan *Run* adalah menghindar dari ancaman pada jarak tertentu. Ancaman yang dimaksud adalah musuh yang lebih besar, proyektil torpedo, ataupun proyektil supernova yang mengarah ke bot. Kemudian, dilihat ancaman mana yang paling berbahaya berdasarkan jaraknya dengan bot. Bot kemudian akan melakukan aksi menghindar/ kabur dengan berjalan lurus (dengan command FORWARD) ke arah yang tepat. Arah tersebut adalah 180 derajat dari arah ke musuh yang lebih besar untuk ancaman berupa musuh, dan 90 derajat dari arah tembakan untuk ancaman berupa tembakan.

3.3.8 Strategi Greedy yang Dipilih untuk Persoalan *PickUpSupernova*

Strategi *greedy* yang dipilih untuk sub persoalan *PickUpSupernova* adalah mem-*pick up* supernova dalam radius jarak tertentu yang cukup dekat. Hal ini bertujuan agar bot dapat fokus melakukan aksi lain saat supernova berada terlalu jauh yang mana kemungkinan akan berhasil diambil pemain lain. Atau dengan kata lain, strategi ini meminimalisasi kemungkinan *pick up* supernova yang sia-sia.

3.3.9 Strategi Greedy yang Dipilih untuk Persoalan *FireSupernova*

Strategi *greedy* yang dipilih untuk sub persoalan *FireSupernova* adalah menembak ke arah lokasi enemy terbesar. Hal ini mepertimbangkan efisiensi dari segi komputasi. Selain itu, strategi ini juga efektif karena pergerakan dan perubahan ukuran satu *enemy* cenderung tidak sedinamis pergerakan dan perubahan ukuran sekelompok *enemy*.

3.3.10 Strategi Greedy yang Dipilih untuk Persoalan DetonateSupernova

Strategi *greedy* yang dipilih untuk sub persoalan *DetonateSupernova* adalah melakukan aksi *Detonate* ketika supernova berada cukup jauh dari bot, berada di sekitar *enemy* terbesar, atau akan keluar dari *boundary map*. Hal ini mempertimbangkan efektivitas dan efisiensi karena target di-*update* berdasarkan yang paling menguntungkan pada *game state* saat ini. Strategi ini juga efisien secara memori karena tidak dibutuhkan penyimpanan target awal. Strategi ini juga efektif untuk menghindari risiko terkena efek supernova dan kesia-siaan supernova (keluar dari *boundary map*).

BAB 4
IMPLEMENTASI DAN PENGUJIAN

4.1 Pseudocode Implementasi Algoritma Greedy pada Program Bot

4.1.1 *Pseudocode BotBrain*

```
{penggunaan package, import file & library}
package BotModels

import Models.*
import Services.*

import BotModels.States.*

public class BotBrain
{KAMUS LOKAL}
      public static BotService botService { Menerima game state
dan bot dari server }
      private static BotState currentState{ State aktif }
      private static PlayerAction currentPlayerAction { Aksi
untuk dikirim ke server }

      {Aksesor}
      public static GetGameState()-> GameState
     { Aksesor Game State terkini }
      -> botService.getGameState()

      public static GetBot()-> GameObject
     { Aksesor Data Bot terkini }
      -> botService.getBot()
      }

      {Mesin State}
      private static BotState GatherFood <- new GatherFood()
      private static BotState UseShield <- new UseShield()
      private static BotState Run <- new Run()
      private static BotState TorpedoAttack <- new
TorpedoAttack()
      private static BotState FireTeleport <- new FireTeleport()
      private static BotState Teleport <- new Teleport()
      private static BotState PickUpSupernova <- new
PickUpSupernova()
      private static BotState FireSupernova <- new
FireSupernova()
      private static BotState DetonateSupernova <- new
DetonateSupernova()

      private static BotState[] states <- {GatherFood,
FireTeleport, Teleport , Run, UseShield, TorpedoAttack,
PickUpSupernova, FireSupernova, DetonateSupernova}

      public static PlayerAction GetBotAction()
     { Mengembalikan aksi yang akan dieksekusi bot }
           currentState <- GetBestState()

           currentPlayerAction <- currentState.GetPlayerAction()
```

```
            -> currentPlayerAction


        private static BotState GetBestState()
      { Mengembalikan state terbaik saat ini }
            BotState.gameState <- GetGameState()
            BotState.bot <- GetBot()

            real highestPriority <- (-99)
            BotState bestState <- null


             i traversal [0..states.size()-1]
                  if (states[i].GetPriorityScore() >
      highestPriority) then
                        highestPriority <-
      state.GetPriorityScore()
                        bestState <- state
            -> bestState
```

4.1.2 *Pseudocode BotState*

```
{penggunaan package, import file & library}
package BotModels

import java.util.List
import java.util.stream.Collectors

import Enums.*
import Models.*

public abstract class BotState
      public static GameState gameState
      public static GameObject bot
      protected static boolean teleporterFired = false
      protected static int teleporterAngle = -1
      protected static boolean supernovaFired = false

      { ABSTRACT METHOD (MUST BE IMPLEMENTED BY EACH STATE) }
      abstract public real calculatePriorityScore()
      abstract public PlayerAction calculatePlayerAction()

      { ACCESSOR }
      public real GetPriorityScore()
      -> this.calculatePriorityScore()

      public PlayerAction GetPlayerAction()
      -> this.calculatePlayerAction()

      { SOME UTILITY FUNCTION }
      protected getDistance(input Position position1, Position
position2) -> real
      { Mengembalikan jarak antara dua posisi }
          real triangleX = Math.abs(position1.x - position2.x)
      real triangleY = Math.abs(position1.y - position2.y)
      -> Math.sqrt(triangleX * triangleX + triangleY * triangleY)

      protected getDistanceBetween(input GameObject object1,
GameObject object2) -> real
```

```
      { Mengembalikan jarak antara dua game object }
      -> getDistance(object1.getPosition(),
object2.getPosition())

      protected getDistanceToBot(input GameObject object) ->
real
      { Mengembalikan jarak antara bot dan game object lain }
      -> getDistance(bot.getPosition(), object.getPosition())

      protected getHeading(input Position position) -> int
      { Mengembalikan heading antara bot dengan posisi lain }
      real direction = toDegrees(Math.atan2(position.y -
bot.getPosition().y, position.x - bot.getPosition().x))
      -> (direction + 360) % 360

      protected getHeading2(input Position position1, Position
position2) -> int
      { Mengembalikan heading dari posisi 1 ke posisi 2 }
      var direction = toDegrees(Math.atan2(position2.y -
position1.y, position2.x - position1.x))
      -> (direction + 360) % 360

      protected getHeadingBetween(input GameObject otherObject)
-> int
      { Mengembalikan heading antara bot dan game object lain }
      -> getHeading(otherObject.getPosition())

      protected toDegrees(input real v) -> int
      { Mengubah radian ke derajat }
      -> (int) (v * (180 / Math.PI))

      protected getGameObjectsByType(input ObjectTypes type) ->
List<GameObject>
      { Mengembalikan list game object berdasarkan tipe }
      -> gameState.getGameObjects().stream().filter(item ->
item.getGameObjectType() == type).collect(Collectors.toList())

      protected getPlayers(input ObjectTypes type) ->
List<GameObject>
      { Mengembalikan list game object berdasarkan tipe }
      -> gameState.getPlayerGameObjects().stream().
collect(Collectors.toList())

      protected getGameObjectsByType(input List<GameObject>
gameObjects, ObjectTypes type) -> List<GameObject>
      { Mengembalikan list game object berdasarkan tipe dari list
game object }
      -> gameObjects.stream().filter(item ->
item.getGameObjectType() == type).collect(Collectors.toList())

      protected getGameObjectsAtArea(input Position position,
int radius) -> List<GameObject>
      { Mengembalikan list game object yang berada di area radius
dari posisi tertentu }
      -> gameState.getGameObjects().stream()
            .filter(item -> getDistance(position,
item.getPosition()) <= radius).collect(Collectors.toList())

      protected getPlayersAtArea(input Position position, int
radius) -> List<GameObject>
```

```
      { Mengembalikan list game object yang berada di area radius
dari posisi tertentu }
      -> gameState.getPlayerGameObjects().stream()
            .filter(item -> getDistance(position,
item.getPosition()) <= radius + item.getSize() + bot.getSize() &&
item.getId() != bot.getId()).collect(Collectors.toList())

      protected getGameObjectsAtBotArea(input int radius) ->
List<GameObject>
      { Mengembalikan list game object yang berada di area radius
dari posisi bot }
      -> getGameObjectsAtArea(bot.getPosition(), radius)

      protected getPlayersAtBotArea(input int radius) ->
List<GameObject>
      { Mengembalikan list game object yang berada di area radius
dari posisi bot }
      -> getPlayersAtArea(bot.getPosition(), radius)

      protected isObjectHeadingTo(input GameObject obj, Position
position, int degreeTolerance) -> boolean
      { Mengembalikan true jika objek tertentu mengarah ke posisi
tertentu, dengan toleransi +- degreeTolerance dari posisi }
      int objHeading = obj.currentHeading

      -> Math.abs((objHeading - getHeading2(obj.position,
position)) % 360) <= degreeTolerance

      protected isObjectHeadingToBot(input GameObject obj, int
degreeTolerance) -> boolean
      { Mengembalikan true jika heading bot ke game object
tertentu }
      -> isObjectHeadingTo(obj, bot.getPosition(),
degreeTolerance)
```

4.1.3 *Pseudocode GatherFood*

```
constant VERY_CLOSE_DISTANCE : integer = 32
constant CLOSE_DISTANCE : integer = 50
constant FAR_DISTANCE : integer = 200

constant BASE_SCORE : real = 100

function calculatePriorityScore() -> real
     return BASE_SCORE

function calculatePlayerAction() -> PlayerAction
     //makanan di world sudah habis -> menghindari obstacles
     if ( len(ListMakanan) == 0) then
          if ( len(ListGasCloud)==0 and distanceToGasCloud <
          ukuranBot) then
               playerAction = dodgeGasCloud()
          if (distanceToBoundary() < VERY_CLOSE_DISTANCE) then
               playerAction = dodgeBoundary()
          return playerAction

     //makanan masih ada -> gatherFood
     if(ukuranBot <=15 or foodDensity(VERY_CLOSE_DISTANCE)>=1)
```

```
     then
          playerAction = goToClosestFood()
     else if (foodDensity(CLOSE_DISTANCE >= 5) then
          playerAction = goToClosestFood()
     else
          playerAction = goToNewArea()

     // menghindari obstacles
     if ( len(ListGasCloud)==0 and distanceToGasCloud <
     ukuranBot) then
          playerAction = dodgeGasClous()
     if (distanceToBoundary() < VERY_CLOSE_DISTANCE) then
          playerAction = dodgeBoundary()
     return playerAction


function foodDensityInRange(integer distance) -> real
     // mengembalikan intensitas food pada area disekitar bot
     dengan jarak distance
     return food.size() + 1.25*superFood.size()

function getGameObjectByQuadrant(real maxQuadrant, real
nQuadrant, integer distance) -> List<GameObject>

     List<GameObject> object =
     getGameObjectsAtBotArea(distance).stream().filter(x ->
     (getHeadingBetween(x) >= (360/maxQuadrant*(nQuadrant-1)))
     &&(getHeadingBetween(x)<(360/maxQuadrant*(nQuadrant)))).coll
     ect(Collectors.toList());

     return object;

function calculateDensity(List<GameObject> objects)-> int
//Menghitung skor intensitas makanan dan rintangan dari suatu
list GameObject
     int food = len(getGameObjectByType(FOOD))
     int superfood = len(getGameObjectByType(SUPERFOOD))
     int gascloud = len(getGameObjectByType(GASCLOUD))
     int torpedo = len(getGameObjectByType(TORPEDOSALCO))
     int supernova = len(getGameObjectByType(SUPERNOVAPICKUP))

     return food*10 + superFood*15 + gasCloud * (-30) +
     torpedo*(-20) + supernova*20

function foodHeatMap(int distance)->integer
//Mengembalikan heading yang menuju ke area dengan intensitas
makanan dan rintangan paling strategis
     List<innteger> density
     i traversal [1..6]
          density[i-1]=calculateDensity(getGameObjectByQuadrant(6
          ,i,distance)))

     idx_max = 0

     i traversal[1..5]
          if (density[idx_max]<density[i]) then
                idx_max = i;

     return ((idx_max)*60)+30
```

```
function distanceToBoundary() -> real
// mengembalikan jarak bot dengan map boundary
     return radiusMap - getDistance(bot,centerPoint)

function getGasCloud() -> List<GameObject>
//mengembalikan list semua gascloud yang ada pada world terurut
membesar berdasar jarak dengan bot
     List<GameObject> gasCloud =
     getGameObjectsByType(ObjectTypes.GASCLOUD).stream().sorted(C
     omparator.comparing(x->
     getDistanceToBot(x))).collect(Collectors.toList())

     return gasCloud

function distanceToGasCloud() -> real
// mengembalikan distance gascloud yang terdekat dengan bot
     if (len(getGasCloud)==0) then
          return 999;
     else
          return getDistanceToBot(getGasCloud[0])

function getHeadingToGasCloud() -> integer
//mengembalikan heading bot ke gascloud terdekat
     if (len(getGasCloud)==0) then
          return currentHeading;
     else
          return getHeadingBetween(getGasCloud[0])

function isSuperFoodActive() -> boolean
      if ((bot.getEffect()>=8 && bot.getEffect() < 16))then
           return true
      else if (bot.getEffect()>=24) then
           if (bot.getEffect()-16 >= 8 && bot.getEffect()-16<16)then
                return true

      return false;


function getDistanceToBotWithHeading(GameObject x) -> double
//Mengembalikan jarak antara food terdekat dengan juga
mempertimbangkan heading

     int headToFood = getHeading(x.position)
     int botHeading = bot.getCurrHeading()
     double actualDist = getDistanceToBot(x)

     if (Math.abs((headToFood - botHeading) % 360) <= 15) then
         return actualDist;
     else
         return actualDist + 50

function goToClosestFood()-> playerAction
//Mengembalikan aksi FORWARD dengan heading menuju food terdekat
     List <GameObject> food = getGameObjectsByType(FOOD)
     sorted food by getDistanceToBotWithHeading()

     List<GameObject>superfood=getGameObjectsByType(SUPERFOOD)
     sorted superfood by getDistanceToBotWithHeading()

     if (not isEmpty(food)) then
```

```
            if (not isEmpty(superfood)) then
            //utamakan superfood apabila superfood tidak aktif
                  if (getDistaceWithHeading(food[0]) >=
                  getDistanceWithHeading(superfood[0]) && not
                  isSuperFoodActive) then
                        playeraction = (FORWARD,
                        getHeadingBetween(superfood[0])
                  else
                        playeraction = (FORWARD,
                        getHeadingBetween(food[0])
            else
                  playeraction = (FORWARD,
                  getHeadingBetween(food[0])
      else
            playeraction = goToNewArea()

      return playeraction


function goToNewArea() -> playerAction
//Mengembalikan aksi FORWARD dengan heading menuju area paling
strategis
      playerAction = (FORWARD, foodHeatMap(FAR_DISTANCE))
      return playerAction

function dodgeBoundary() -> playerAction
//Mengembalikan aksi FORWARD dengan heading menuju pusat map
      playerAction=(FORWARD,(getHeading(gameState.getWorld().getCe
      nterPoint()))
      return playerAction

function dodgeGasCloud() -> playerAction
// Mengembalikan aksi FORWARD dengan heading untuk menghindari
gas cloud
      playerAction.action=(FORWARD(getHeadingToGasCloud()+180)%360
      )
      return playerAction;
```

4.1.4 *Pseudocode FireTeleport*

```
{penggunaan package, import file & library}
package BotModels.States

import BotModels.*
import Enums.*
import Models.*

import java.util.*
import java.util.stream.*

public class FireTeleport extends BotState

    {Konstanta besar untuk menghitung nilai minimum}
    constant integer LARGE_NUM <- 1000000

    {METODE ABSTRAK}
```

```
public calculatePriorityScore()-> real

    {Mengecek apakah bot sudah pernah menembak
    teleporter dan apakah bot memiliki teleporter
    yang tersedia.}
   if (BotState.teleporterFired = True OR
       bot.getTeleporterCount() = 0 OR
       BotState.teleporterAngle != -1)then
       {Jika tidak, maka prioritas = 0}
       -> 0
   else
       {Jika ya, maka menghitung prioritas
        berdasarkan ukuran pemain}
       -> sizeSaveToFire()

public calculatePlayerAction() -> PlayerAction

     {Mendapatkan daftar pemain musuh yang lebih
     kecil dari ukuran bot dan berada di sekitar
     bot}
    List<GameObject> enemySmallList = enemySmall()

    if (enemySmallList.size() != 0) then
      {Jika ada pemain musuh yang lebih kecil dari
      bot, maka mencari pemain musuh yang terkecil
      untuk dijadikan target teleporter}
      GameObject target <-
                    enemySmallest(enemySmallList)
      {Menembak teleporter ke arah target}
        -> fireTeleport(target)
     else
        {Jika tidak ada pemain musuh yang lebih kecil
        dari bot, maka bot akan STOP}
        PlayerAction playerAction <-
                                    new PlayerAction()

        playerAction.action <- PlayerActions.STOP
        playerAction.heading <- bot.getCurrHeading()

        -> playerAction

{METODE PEMBANTU}
private enemySmall() -> List<GameObject>
    {Mendapatkan daftar pemain musuh yang lebih kecil
    dari ukuran bot - 20 dan berada di sekitar bot}

    {Jarak antara bot dan tepi world dikurangi offset
    5}
    integer radius <-
                gameState.getWorld().getRadius() - 5

    List<GameObject> enemySmallList <-
               getPlayersAtBotArea(radius)
               .stream()
               .filter(x -> x.getId() != bot.getId()
               AND x.getSize() <= bot.getSize() - 20)
               .collect(Collectors.toList())

    -> enemySmallList
```

```
private enemySmallest(List<GameObject>enemySmallList)
                                        -> GameObject
    {Mencari pemain musuh terkecil dari daftar pemain
    musuh yang diberikan}

    integer minsize <- LARGE_NUM, size
    integer smallestidx <- 0

    i traversal [0..enemySmallList.size()]
        {iterasi elemen list untuk mendapatkan musuh
        terkecil}
        size <- enemySmallList[i].getSize();
        if (size < minsize) then
            minsize <- size;
            smallestidx <- i

    -> enemySmallList[smallestidx]

private calcHeadingOffset(GameObject target)->integer
    {Menerima parameter target, yaitu GameObject
    musuh yang menjadi target tembakan teleporter,
    kemudian menghitung perbedaan arah (offset)
    antara heading bot dengan heading target}

    integer heading <- getHeadingBetween(target)
    integer offset <- 0

    {Jika heading target berada di salah satu dari 4
    kuadran relatif terhadap heading bot, offset
    diberi nilai +2 atau -2, tergantung arah
    target.}

    if(heading >= 0 AND heading < 90) then
        integer theading <- target.getCurrHeading()
        if(theading >= 45 AND theading < 225) then
             offset <- 2
        else offset <- -2

    else if(heading >= 90 AND heading < 180) then
        integer theading <- target.getCurrHeading()
        if(theading >= 135 AND theading < 315) then
             offset <- 2
        else offset <- -2

    else if(heading >= 180 AND heading < 270) then
        integer theading <- target.getCurrHeading()
        if(theading >= 225 AND theading < 45) then
             offset <- 2
        else offset <- -2

    else if(heading >= 270 AND heading < 360){
        integer theading <- target.getCurrHeading()
        if(theading >= 315 AND theading <= 135)
             offset <- 2
        else offset <- -2

   {Mengembalikan nilai heading bot ditambah offset
   sebagai arah tembakan teleporter}
    -> heading + offset
```

```
    private sizeSaveToFire() -> real
        {Menghitung skor prioritas suatu aksi teleporter
        dengan membandingkan ukuran bot dengan ukuran
        musuh terkecil yang dapat ditembak dengan
        teleporter, serta jarak bot dengan musuh terkecil
        tersebut}

        integer botSize <- bot.getSize()
        List<GameObject> enemySmallList <- enemySmall()

        if(enemySmallList.size() = 0)then
           {Jika tidak ada musuh yang dapat ditembak,
           nilai prioritas diberi nilai 0}
            -> 0
        else
           {Jika ada musuh yang dapat ditembak, nilai
           prioritas dihitung sebagai (ukuran bot /
           ukuran musuh terkecil) dikali 250 dikali (250
           / jarak bot dengan musuh terkecil)}
            GameObject smallestEnemy <-
                        enemySmallest(enemySmallList)
            integer smallestEnemySize <-
                        smallestEnemy.getSize()

            -> (botSize/smallestEnemySize) * 250 *
               (250/getDistanceToBot(smallestEnemy))

    private fireTeleport(GameObject target)->
                                          PlayerAction
        {Menerima parameter target, yaitu GameObject
        musuh yang menjadi target tembakan teleporter

        {Membuat objek playerAction dari kelas
        PlayerAction}
        PlayerAction playerAction <- new PlayerAction()

        {Memanggil calcHeadingOffset(target) untuk
        mendapatkan arah tembakan}
        integer shootDirection <-
                             calcHeadingOffset(target)

        {Mengeset playerAction.action dan
        playerAction.heading}
        playerAction.action <- PlayerActions.FIRETELEPORT
        playerAction.heading <- shootDirection

        {Mengeset status teleporter}
        BotState.teleporterAngle <- shootDirection
        BotState.teleporterFired <- True

        {Mengembalikan objek playerAction}
        -> playerAction
```

4.1.5 *Pseudocode Teleport*

```
{penggunaan package, import file & library}
package BotModels.States;

import BotModels.*;
```

```
import Enums.*;
import Models.*;

import java.util.*;
import java.util.stream.Collectors;

public class Teleport extends BotState

    {METODE ABSTRAK}
    public calculatePriorityScore() -> float
        {Metode menghitung dan mengembalikan skor
        prioritas}

        if (BotState.teleporterFired = False) then
            {Jika teleporter belum digunakan prioritas 0}
            -> 0
        else
            {jika teleporter telah digunakan, metode
            decideTeleportScore() dipanggil untuk
            memutuskan apakah teleporter harus digunakan
            atau tidak}
            -> decideTeleportScore()

    public calculatePlayerAction() -> PlayerAction
        -> teleport()

    {METODE PEMBANTU}
    private decideTeleportScore() -> integer
       {Memutuskan apakah teleporter harus digunakan atau
       tidak}

       {Memeriksa apakah terdapat teleporter bot aktif}
        List<GameObject> myTeleporter <-
           getGameObjectsByType(ObjectTypes.TELEPORTER)
           .stream()
           .filter(item ->
                   Math.abs((item.getCurrHeading() -
                   BotState.teleporterAngle) % 360) <=
                   15).collect(Collectors.toList());

        if (myTeleporter.size() = 0) then
            {Jika tidak, teleporterFired diatur ke false
            dan skor prioritas menjadi 0}

            BotState.teleporterFired <- False
            -> 0
        else
            {Jika ya, cari target yang berada di sekitar
            teleporter}

            List<GameObject> targetAroundTeleporter <-
            getPlayersAtArea(myTeleporter[0].position,
                             bot.getSize() + 100)
            .stream()
            .filter(e -> e.getId() !=
                    bot.getId())
                    .collect(Collectors.toList())
            if (targetAroundTeleporter.size() = 0) then
                {Jika tidak ada target yang berada di
                sekitar teleporter, skor prioritas
```

```
                menjadi 0}

                -> 0

            else
                {Jika ada target yang berada di sekitar
                teleporter, peeriksa apakah target
                tersebut lebih besar dari ukuran bot}

                i traversal[0..
                          targetAroundTeleporter.size()]
                  if (targetAroundTeleporter[i].getSize()
                      >= bot.getSize()) then
                      {Jika ya, skor prioritas menjadi 0}

                      -> 0
                i traversal[0..
                          targetAroundTeleporter.size()]
                  if (getDistanceToBot
                     (targetAroundTeleporter[i])
                      <= bot.getSize() + 100) then
                      {Jika tidak dan target berada dalam
                      jarak 100 dari bot, skor prioritas
                      menjadi 900, bot harus melakukan
                      teleport}

                      -> 900
                -> 0

    private teleport() -> PlayerAction
        PlayerAction playerAction <- new PlayerAction()

        {Mengeset playerAction.action dan
        playerAction.heading}
        playerAction.action <- PlayerActions.TELEPORT
        playerAction.heading <- bot.getCurrHeading()

        {Mengeset status teleporter}
        BotState.teleporterAngle <- -1
        BotState.teleporterFired <- False

        {Mengembalikan objek playerAction}
        -> playerAction
```

4.1.6 *Pseudocode TorpedoAttack*

```
{penggunaan package, import file & library}
package BotModels.States

import BotModels.*
import Enums.*
import Models.*

import java.util.*

public class TorpedoAttack extends BotState
    constant integer VERY_CLOSE_DISTANCE <- 50
    constant integer CLOSE_DISTANCE <- 150
    constant integer LARGE_NUM <- 1000000
```

```
{METODE ABSTRAK}
public calculatePriorityScore() -> real
    if (bot.getTorpedoSalvoCount() = 0 OR
        sizeSaveToAttack() = 0) then
        {Memeriksa apakah bot memiliki torpedo atau
        apakah ukuran bot terlalu kecil untuk
        menyerang musuh. Jika salah satunya benar,
        metode ini mengembalikan nilai 0.}
        -> 0

    else
        {Jika tidak, mencari musuh yang berada dalam
        jarak dekat}

        List<GameObject> enemyInRangeListClose <-
         enemyInRange(CLOSE_DISTANCE + bot.getSize())

        if (enemyInRangeListClose.size() = 0) then
           {Jika tidak ada musuh yang terdeteksi
           dalam dekat, mengembalikan nilai 0}

            -> 0

        else
           {Jika ada musuh yang terdeteksi, memeriksa
           apakah ada musuh yang sangat dekat dengan
           bot}

            List<GameObject>
                  enemyInRangeListVeryClose <-
                  enemyInRange(VERY_CLOSE_DISTANCE +
                               bot.getSize())
            if (enemyInRangeListVeryClose.size()= 0)
                then
                {Jika tidak, menghitung skor
                prioritas berdasarkan jarak antara
                bot dan musuh terdekat. Semakin besar
                jarak semakin rendah prioritas,
                prioritas ini dikalikan dengan
                ukuran musuh}

                GameObject closestEnemy <-
                           getObjClosestEnemy(
                               enemyInRangeListClose)
                real dist <-
                           getDistClosestEnemy(
                                       closestEnemy)
                real prio <-
                           (CLOSE_DISTANCE/(dist -
                            closestEnemy.getSize()))
                            * sizeSaveToAttack() *
                            200)
                -> prio

            else
                {Jika terdapat musuh yang sangat
                dekat, menghitung skor prioritas yang
                lebih tinggi daripada musuh yang
                hanya berada dalam jarak dekat. Skor
```

```
                prioritas yang dihasilkan juga akan
                dikalikan dengan ukuran musuh}

                GameObject closestEnemy <-
                          getObjClosestEnemy(
                          enemyInRangeListVeryClose)
                real dist <-
                          getDistClosestEnemy(
                          getObjClosestEnemy(
                         enemyInRangeListVeryClose))
                real prio <-
                          (VERY_CLOSE_DISTANCE/(dist
                          - closestEnemy.getSize()))
                          * sizeSaveToAttack() *
                          200)
                -> prio

public calculatePlayerAction() -> PlayerAction

    {Mendapatkan daftar pemain berada di jarak dekat
    dengam bot}
    List<GameObject> enemyInRangeList <-
                          enemyInRange(bot.getSize()
                          + CLOSE_DISTANCE)
    if(enemyInRangeList.size() != 0) then
       {Jika ada pemain musuh dalam jarak dekat, maka
       mencari pemain musuh yang terdekat untuk
       dijadikan target torpedo}

        GameObject enemy <-
                          getObjClosestEnemy(
                         enemyInRangeList)
        -> attackTorpedo(enemy)

    else
        {Jika tidak ada pemain musuh yang dekat
        dari bot, maka bot akan STOP}
        PlayerAction playerAction <- new
                                     PlayerAction()
        playerAction.action <- PlayerActions.STOP
        playerAction.heading <- 0
        -> playerAction

{METODE PEMBANTU}
private enemyInRange(integer radius) ->
                                    List<GameObject>
    {Mencari musuh yang berada dalam jangkauan bot
    dalam radius tertentu, mengembalikan list musuh
    yang berada dalam jangkauan}

    List<GameObject> enemyInRangeList <-
                        getPlayersAtBotArea(radius)
    -> enemyInRangeList

private getObjClosestEnemy(List<GameObject>
                          enemyInRangeList) ->
                                        GameObject
    {Mencari dan mengembalikan musuh terdekat dari
    daftar musuh yang diberikan}
```

```
        real mindist <- LARGE_NUM, dist
        ineger closestidx <- 0;
        i traversal [0..enemyInRangeList.size()]
            dist <- getDistanceToBot(
                    enemyInRangeList.get(i))
            if (dist < mindist) then
                mindist <- dist
                closestidx <- i
        -> enemyInRangeList[closestidx]

    private getDistClosestEnemy(GameObject closestEnemy)
                                                    -> real
        {Menghitung jarak antara bot dan musuh terdekat
        yang diberikan}

        -> getDistanceToBot(closestEnemy)

    private sizeSaveToAttack() -> integer
       {Menentukan seberapa besar ukuran bot agar dapat
       menyerang musuh dengan torpedo. Mengembalikan
       nilai 0 jika bot terlalu kecil untuk menyerang
       , dan meningkat seiring dengan bertambahnya
        ukuran bot}

        integer botSize <- bot.getSize()
        if (botSize < 15) then -> 0
        if (botSize < 30) then -> 1
        if (botSize < 45) then -> 2
        else -> 3

    private attackTorpedo(GameObject enemy) ->
                                            PlayerAction

        {Mengembalikan tindakan serangan torpedo yang
        harus dilakukan oleh bot}

        {Mengeset playerAction.action dan
        playerAction.heading}
        PlayerAction playerAction <- new PlayerAction()

        {Mengeset status teleporter}
        playerAction.action <-
                            PlayerActions.FIRETORPEDOES
        playerAction.heading <- getHeadingBetween(enemy)

        {Mengembalikan objek playerAction}
        -> playerAction
```

4.1.7 *Pseudocode UseShield*

```
package BotModels.States

import BotModels.*
import Enums.*
import Models.*

import java.util.*
import java.util.stream.*
```

```
public class UseShield extends BotState
      { KAMUS LOKAL }
      { CONSTANTS }
      private final int CLOSE_DISTANCE <- 150

      { ABSTRACT METHOD }
      public calculatePriorityScore() -> real
      { check if bot is in danger of torpedo heading to it }
            if (bot.getSize() < 30 || bot.getEffect() >= 16)
      then
                  -> 0
            int torpedoCount <-
      getTorpedosInRange(CLOSE_DISTANCE + bot.getSize()).size()
            real priorityScore <- bot.shieldCount * torpedoCount
      * 40
            -> priorityScore

      public calculatePlayerAction()-> PlayerAction
      { activate shield }
            PlayerAction playerAction <- new PlayerAction()
            playerAction.action <- PlayerActions.ACTIVATESHIELD

            int headToCenter <- getHeading(gameState.getWorld().
      getCenterPoint())
            playerAction.heading <- headToCenter

            -> playerAction


      { HELPER METHODS }
      private getTorpedosInRange(int distance) ->
List<GameObject>
      { Mengembalikan list torpedo yang mengarah ke bot }
            List<GameObject> torpedoes <- getGameObjectsByType
            (getGameObjectsAtBotArea(distance), ObjectTypes.
            TORPEDOSALVO).stream().filter(x ->
            isObjectHeadingToBot(x, (int) (bot.getSize()/2.5)))
            .collect(Collectors.toList())
```

4.1.8 *Pseudocode Run*

```
package BotModels.States

import BotModels.*
import Enums.*
import Models.*

import java.util.*
import java.util.stream.*

public class Run extends BotState
    { KAMUS LOKAL }
    { CONSTANTS }
      private final int MEDIUM_DISTANCE <- 150
      private final int DISTANCE_TO_TOLERANCE_RATIO <- 10

      private final int RUN_ANGLE <- 180
```

```
      { ABSTRACT METHOD }
      public calculatePriorityScore() -> real
      { check if bot is in danger }
      int biggerEnemyCount <-
getBiggerEnemiesInRange(MEDIUM_DISTANCE + bot.getSize()).size()
      int teleportProjectileCount <-
getTeleporterInRange(MEDIUM_DISTANCE + bot.getSize()).size()
      int supernovaProjectileCount <- getSupernovaProjInRange
(MEDIUM_DISTANCE + bot.getSize()).size()


      real priorityScore <-
            biggerEnemyCount * 50 +
            teleportProjectileCount * 150 +
            supernovaProjectileCount * 150

      -> priorityScore


      public calculatePlayerAction() -> PlayerAction
      PlayerAction playerAction <- new PlayerAction()

      { if in gas cloud or boundary }
      if (!getGasCloud().isEmpty() &&  bot.getSize() >
distanceToGasCloud() + getGasCloud().get(0).getSize()) then
            -> dodgeGasCloud()

      if (distanceToBoundary()< 20+bot.getSize()) then
            -> dodgeBoundary()

      { set action to run }
      playerAction.action <- PlayerActions.FORWARD

      List<GameObject> biggerEnemiesClose <-
getBiggerEnemiesInRange(MEDIUM_DISTANCE)
      GameObject closestEnemy <- biggerEnemiesClose.size() > 0 ?

biggerEnemiesClose.stream().sorted(Comparator.comparing(x ->
getDistanceToBot(x))).collect(Collectors.toList()).get(0) : null

      List<GameObject> teleportProjectileClose <-
getTeleporterInRange(MEDIUM_DISTANCE)
      GameObject closestTeleportProjectile <-
teleportProjectileClose.size() > 0 ?

teleportProjectileClose.stream().sorted(Comparator.comparing(x ->
getDistanceToBot(x))).collect(Collectors.toList()).get(0) : null

      List<GameObject> supernovaProjectileClose <-
getSupernovaProjInRange(MEDIUM_DISTANCE)
      GameObject closestSupernovaProjectile <-
supernovaProjectileClose.size() > 0 ?

supernovaProjectileClose.stream().sorted(Comparator.comparing(x
-> getDistanceToBot(x))).collect(Collectors.toList()).get(0) :
null

      if (closestEnemy != null) then
            { run away from bigger enemy }
```

```
            playerAction.heading <-
(getHeadingBetween(closestEnemy) + RUN_ANGLE) % 360
      else if (closestTeleportProjectile != null) then
            { run away from teleporter projectile }
            playerAction.heading <-
(getHeadingBetween(closestTeleportProjectile) + RUN_ANGLE - 90) %
360
      else if (closestSupernovaProjectile != null) then
            { run away from supernova projectile }
           playerAction.heading <-
(getHeadingBetween(closestSupernovaProjectile) + RUN_ANGLE - 90)
% 360
      else
            playerAction.heading <-  bot.getCurrHeading()


      -> playerAction
      }

      { HELPER METHODS }
      private getBiggerEnemiesInRange(int distance) ->
List<GameObject>
      { return list of bigger enemies within distance that is
heading towards bot }
      List<GameObject> biggerEnemies <-
getPlayersAtArea(bot.getPosition(), distance)
            .stream().filter(x -> x.getId() != bot.getId() &&
x.getSize() > bot.getSize() && isObjectHeadingTo(x,
bot.getPosition(), distance/DISTANCE_TO_TOLERANCE_RATIO))
            .collect(Collectors.toList())
      -> biggerEnemies

      private getTeleporterInRange(int distance) ->
List<GameObject>
      { return list of teleporter projectiles within distance
that is heading towards bot }
      List<GameObject> teleportProjectiles <-
getGameObjectsByType(getGameObjectsAtBotArea(distance),
ObjectTypes.TELEPORTER).
            stream().filter(x -> isObjectHeadingToBot(x,
distance/DISTANCE_TO_TOLERANCE_RATIO)).collect(Collectors.toList(
))
      -> teleportProjectiles

      private getSupernovaProjInRange(int distance) ->
List<GameObject>
      { return list of supernova projectiles within distance that
is heading towards bot }
      List<GameObject> supernovaProjectiles <-
getGameObjectsByType(getGameObjectsAtBotArea(distance),
ObjectTypes.SUPERNOVABOMB).
            stream().filter(x -> isObjectHeadingToBot(x,
distance/DISTANCE_TO_TOLERANCE_RATIO)).collect(Collectors.toList(
))
      -> supernovaProjectiles

      private dodgeBoundary() -> PlayerAction
      { return player action to dodge the boundary }
      PlayerAction playerAction <- new PlayerAction()
      playerAction.action <- PlayerActions.FORWARD
```

```
      playerAction.heading <-
(getHeading(gameState.getWorld().getCenterPoint()) + 60) % 360

      -> playerAction

      private dodgeGasCloud() -> PlayerAction
      { return player action to dodge the gas cloud }
      PlayerAction playerAction <- new PlayerAction()
      playerAction.action <- PlayerActions.FORWARD
      playerAction.heading <- (getHeadingToGasCloud() + 180) %
360
      -> playerAction

      private distanceToBoundary() -> real
      { return distance to the boundary of the world }
      -> gameState.getWorld().getRadius() -
getDistance(gameState.getWorld().getCenterPoint(),
bot.getPosition())

      private getGasCloud() -> List<GameObject>
      { return list of gas cloud sorted by distance to bot }
      List<GameObject> gasCloud <-
getGameObjectsByType(ObjectTypes.GASCLOUD)
            .stream().sorted(Comparator
            .comparing(x-> getDistanceToBot(x)))
            .collect(Collectors.toList())
          -> gasCloud

      private distanceToGasCloud() -> real
      { return distance to the closest gas cloud }
      if (getGasCloud().isEmpty()) then
            -> 999
      else
            -> getDistanceToBot(getGasCloud().get(0))

      private getHeadingToGasCloud() -> int
      { return heading to the closest gas cloud }
      if (getGasCloud().isEmpty()) then
            -> bot.getCurrHeading()
      else
            -> getHeadingBetween(getGasCloud().get(0))
```

4.1.9 *Pseudocode PickUpSupernova*

```
fuction calculatePriorityScore() -> real
// Implementasi perhitungan skor prioritas untuk state
PickUpSuperNova apabila jarak bot dan supernova  <= 250 maka
ambil supernova
      if (supernoveDistance()<=250-bot.getSize() and
      supernoveDistance()!=-99) then
            return 2000
      else
            return 0

function calculatePlayerAction()-> playerAction
//Implementasi pemilihan aksi bot untuk state PickUpSupernova
```

```
      playerAction = pickUpSupernova()
      return playerAction

function supernoveDistance()-> real
// Mengembalikan jarak supernova ke bot, -99 bila pickup supernova
tidak ada
       if (getGameObjectsByType(SUPERNOVAPICKUP).isEmpty())then
            return -99
       else
            return getDistanceToBot(getGameObjectsByType
            (ObjectTypes.SUPERNOVAPICKUP).get(0))

fuction pickUpSupernova()-> playerAction
//Mengembalikan aksi FORWARD dengan heading mengarah ke supernova
pick up
      PlayerAction playerAction = new PlayerAction();
      playerAction.action = PlayerActions.FORWARD;
      playerAction.heading =getHeadingBetween(
      getGameObjectsByType(ObjectTypes.SUPERNOVAPICKUP).get(0))
      return playerAction
```

4.1.10 *Pseudocode FireSupernova*

```
function calculatePriorityScore()-> real
// Implementasi perhitungan skor prioritas untuk state
FireSupernova, apabila player memiliki supernova pickup makan
dstate akan dilakukan
     if (bot.getSupernovaCount()==1) then
          return 1000
     return 0

function calculatePlayerAction()-> playerAction
//Implementasi pemilihan aksi bot untuk state FireSupernova
     playerAction = fireSupernova()
     return playerAction

function getTargetHeading()-> integer
//Mengembalikan heading pada musuh yang paling besar
     List <GameObject> enemy =
     gameState.getPlayerGameObjects().stream().filter(y->
     y.getId() != bot.getId()).sorted(Comparator.comparing(x->
     x.getSize())).collect(Collectors.toList())

     GameObject biggestEnemy = enemy.get(enemy.size()-1)
     return getHeadingBetween(biggestEnemy)

function fireSupernova()-> PlayerAction
//Mengembalikan aksi FIRESUPERNOVA dengan heading mengarah ke
musuh yang memiliki ukuran paling besar
    playerAction.action = PlayerActions.FIRESUPERNOVA
    playerAction.heading = getTargetHeading()
    BotState.supernovaFired = true
    return playerAction
```

4.1.11 *Pseudocode DetonateSupernova*

```
function calculatePriorityScore()-> real
// Implementasi perhitungan skor prioritas untuk state
DetonateSupernova, state terjadi apabila terdapat supernova bomb
yang diledakkan sendiri oleh bom dan telah siap untuk diledakkan
yaitu apabila telah mendekati target dan aman dari musuh serta
teleport jangan sampai keluar dari map
      if (isBombExist() && BotState.supernovaFired )then
           if (isReadytoDetonate())then
               return 2000
           else if (distanceSupernovaToBoundary()<=50)then
               return 2000
           else
               return 0

      else
            return 0

function calculatePlayerAction()-> PlayerAction
// Implementasi pemilihan aksi bot untuk state DetonateSupernova
      playerAction = detonateSupernova()
      return playerAction

function isBombExist()-> boolean
// Mengembalikan true apabila terdapat supernova bomb pada world
      return !getGameObjectsByType(SUPERNOVABOMB.isEmpty()


function distanceSupernovaToBoundary()->real
// Mengembalikan jarak supernovabomb ke boundary
     supernovaBomb=getGameObjectsByType(SUPERNOVABOMB)[0]  return
     gameState.getWorld().getRadius()
     -getDistance(gameState.getWorld().getCenterPoint(),
     supernovaBomb.getPosition())

function detonateSupernova()-> PlayerAction
// Mengembalikan aksi DETONATESUPERNOVA apabila supernova bomb
telah sampai pada sasaran dan berada pada jarak tertentu dari bot
     playerAction.action = PlayerActions.DETONATESUPERNOVA;
     playerAction.heading = 0
     BotState.supernovaFired = false
     return playerAction
```

4.2 Struktur Data yang Digunakan pada Program Bot

4.2.1 Struktur Data pada *BotBrain*

Kelas *BotBrain* merupakan kelas yang berfungsi sebagai pengambil keputusan dalam pemilihan *state*, yang ditentukan berdasarkan perhitungan skor prioritas pada *state* permainan terkini.

Tabel 22. Atribut dan Metode Kelas *BotBrain*

Attribut	**Penjelasan**
public static BotService botService	Referensi ke BotService yang menerima data dari server
private static BotState currentState	State Bot saat ini
private static PlayerAction currentPlayerAction	Aksi pemain saat ini
private static BotState GatherFood	Instans dari state *GatherFood*
private static BotState UseShield	Instans dari state *UseShield*
private static BotState Run	Instans dari state *Run*
private static BotState TorpedoAttack	Instans dari state *TorpedoAttack*
private static BotState FireTeleport	Instans dari state *FireTeleport*
private static BotState Teleport	Instans dari state *Teleport*
private static BotState[] states	List dari state bot
Metode	**Penjelasan**
public static GameState GetGameState()	Aksesor GameState
public static GameObject GetBot()	Aksesor ke GameObject Bot
public static PlayerAction GetBotAction()	Mengembalikan aksi pemain yang akan dilakukan (untuk dikirim ke server)
private static BotState GetBestState()	Memilih state terbaik berdasarkan perhitungan skor prioritas dari masing-masing state

4.2.2 Struktur Data pada *BotState*

Kelas *BotState* adalah kelas abstrak untuk *state-state* yang ada pada bot. Kelas BotState ini memiliki sejumlah atribut statik yang dapat diakses oleh setiap state, dan juga *method-method* standar yang digunakan dalam pengambilan informasi dari *game state*. Fungsi abstrak yang harus diimplementasikan adalah perhitungan skor prioritas dan pemilihan aksi.

Tabel 23. Atribut dan Metode Kelas *BotState*

Attribut	**Penjelasan**
public static GameState gameState	State dari game yang diperbarui setiap tick
public static GameObject bot	GameObject bot dari server
protected static boolean teleporterFired	Kondisi penembakan teleport (sudah ditembak/belum)
protected static int teleporterAngle	Sudut teleporter yang ditembakkan
Metode	**Penjelasan**
abstract public float calculatePriorityScore()	Fungsi abstrak untuk perhitungan skor prioritas masing-masing state
abstract public PlayerAction calculatePlayerAction()	Fungsi abstrak untuk pemilihan aksi masing-masing state
public float GetPriorityScore()	Aksesor skor prioritas
public PlayerAction GetPlayerAction()	Aksesor aksi dari state
protected double getDistance(Position position1, Position position2)	Mengembalikan jarak antara 2 posisi
protected double getDistanceBetween(GameObject object1, GameObject object2)	Mengembalikan jarak antara 2 GameObject
protected double getDistanceToBot(GameObject object)	Mengembalikan jarak antara GameObject dengan bot
protected int getHeading(Position position)	Menghitung arah sudut bot terhadap suatu posisi
protected int getHeading2(Position position1, Position position2)	Menghitung arah sudut suatu posisi terhadap posisi lainnya
protected int getHeadingBetween(GameObject	Menghitung arah sudut bot terhadap GameObject lain

otherObject)	
protected int toDegrees(double v)	Konversi radian ke derajat
protected List<GameObject> getGameObjectsByType(ObjectTypes type)	Mengembalikan list GameObject di game dengan tipe tertentu
protected List<GameObject> getPlayers(ObjectTypes type)	Mengembalikan list Player di game
protected List<GameObject> getGameObjectsByType(List<GameObject> gameObjects, ObjectTypes type)	Mengembalikan list GameObject dengan tipe tertentu dari list GameObject lain
protected List<GameObject> getGameObjectsAtArea(Position position, int radius)	Mengembalikan list GameObject pada suatu posisi dalam radius tertentu
protected List<GameObject> getPlayersAtArea(Position position, int radius)	Mengembalikan list Player pada suatu posisi dalam radius tertentu
protected List<GameObject> getGameObjectsAtBotArea(int radius)	Mengembalikan list GameObject pada area bot dalam radius tertentu
protected List<GameObject> getPlayersAtBotArea(int radius)	Mengembalikan list Player pada area bot dalam radius tertentu
protected boolean isObjectHeadingTo(GameObject obj, Position position, int degreeTolerance)	Mengecek apakah suatu GameObject mengarah ke posisi tertentu
protected boolean isObjectHeadingToBot(GameObject obj, int degreeTolerance)	Mengecek apakah suatu GameObject mengarah ke bot

4.2.3 Struktur Data pada *GatherFood*

Tabel 24. Atribut dan Metode Kelas *GatherFood*

Attribut	Penjelasan
private final int VERY_CLOSE_DISTANCE	Konstanta untuk ukuran jarak sangat dekat
private final int CLOSE_DISTANCE	Konstanta untuk ukuran jarak dekat
private final int FAR_DISTANCE	Konstanta untuk ukuran jarak jauh

private final float BASE_SCORE	Konstanta untuk skor prioritas default
Metode	**Penjelasan**
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state GatherFood
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state Teleport
private PlayerAction goToClosestFood()	Mengembalikan aksi FORWARD dengan heading menuju food terdekat
private PlayerAction goToNewArea()	Mengembalikan aksi FORWARD dengan heading menuju area paling strategis
private PlayerAction dodgeBoundary()	Mengembalikan aksi FORWARD dengan heading menuju pusat map
private PlayerAction dodgeGasCloud()	Mengembalikan aksi FORWARD dengan heading untuk menghindari gas cloud
private double foodDensityInRange(int distance)	Mengembalikan skor intensitas makanan di area sekitar bot dengan jarak tertentu
private List<GameObject> getGameObjectsByQuadran(float maxQuadrant, float nQuadrant, int distance)	Mengembalikan list objek-objek yang berada pada kuadran ke nQuadrant dari area yang terbagi menjadi maxQuadrant area sejauh distance dengan pusat bot
private int calculateDensity(List<GameObject> objects){	Menghitung skor intensitas makanan dan rintangan dari suatu list GameObject
private int foodHeatMap(int distance)	Mengembalikan heading yang menuju ke area dengan intensitas makanan dan rintangan paling strategis
private double distanceToBoundary()	Mengembalikan jarak bot ke boundary
private List<GameObject> getGasCloud()	Mengembalikan semua objek gas cloud yang ada di map terurut secara

	membesar berdasarkan jarak dengan bot
private double distanceToGasCloud()	Mengembalikan jarak gas cloud terdekat dari bot
private int getHeadingToGasCloud()	Mengembalikan heading antara bot dan gas cloud terdekat
private boolean isSuperFoodActive()	Mengembalikan true apabila superfood sedang aktif, false jika tidak
private double getDistanceToBotWithHeading(GameObject x)	Mengembalikan jarak antara food terdekat dengan juga mempertimbangkan heading

4.2.4 Struktur Data pada *FireTeleport*

Kelas *FireTeleport* adalah kelas yang berfungsi dalam melakukan aksi menembak teleporter.

Tabel 25. Atribut dan Metode Kelas *FireTeleport*

Attribut	**Penjelasan**
private final int LARGE_NUM	Konstanta untuk nilai besar
Metode	**Penjelasan**
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state FireTeleport
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state FireTeleport
private List<GameObject> enemySmall()	Mengembalikan list *enemy* yang berukuran <= ukuran bot - 20
private GameObject enemySmallest(List<GameObject> enemySmallList)	Mengembalikan *enemy* terkecil dari list enemy berukuran kecil
private int calcHeadingOffset(GameObject target)	Mengembalikan heading offset berdasarkan letak *enemy* dan arah geraknya saat ini.
private float sizeSaveToFire()	Mengembalikan nilai untuk ukuran bot tertentu dengan menghitung rasionya terhadap *enemy* target
private PlayerAction	Mengembalikam aksi

fireTeleport(GameObject target	FIRETELEPORT dengan *heading* sesuai hasil kakulasi dan men-*set* status teleporter.

4.2.5 Struktur Data pada *Teleport*

Kelas *Teleport* adalah kelas yang berfungsi dalam melakukan aksi teleportasi.

Tabel 18. Atribut dan Metode Kelas *Teleport*

Metode	Penjelasan
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state Teleport
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state Teleport
private int decideTeleportScore()	Mengembalikan hasil perhitungan skala prioritas aksi Teleport dan perhitungan kelayakan aksi yang menunjukan pemilihan keputusan aksi.
private PlayerAction teleport()	Mengembalikan aksi TELEPORT dan melakukan reset status teleporter.

4.2.6 Struktur Data pada *TorpedoAttack*

Kelas *TorpedoAttack* adalah kelas yang berfungsi dalam melakukan aksi menembakan torpedo.

Tabel 26. Atribut dan Metode Kelas *TorpedoAttack*

Attribut	Penjelasan
private final int VERY_CLOSE_DISTANCE	Konstanta untuk ukuran jarak sangat dekat
private final int CLOSE_DISTANCE	Konstanta untuk ukuran jarak dekat
private final int LARGE_NUM	Konstanta untuk nilai besar
Metode	**Penjelasan**
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state TorpedoAttack
public PlayerAction	Implementasi pemilihan aksi bot

calculatePlayerAction()	untuk state TorpedoAttack
private List<GameObject> enemyInRange(int radius)	Mengembalikan list *enemy* yang berada pada radius tertentu
private GameObject getObjClosestEnemy(List<GameObject> enemyInRangeList)	Mengembalikan *enemy* terdekat dengan bot dari list enemy pada radius tertentu
private double getDistClosestEnemy(GameObject closestEnemy)	Mengembalikan jarak *enemy* terdekat dari bot.
private int sizeSaveToAttack()	Mengembalikan nilai untuk ukuran bot tertentu dalam rentang 0 (ukuran < 15) sampai 3 (ukuran >= 45)
private PlayerAction attackTorpedo(GameObject enemy)	Mengembalikan aksi FIRETORPEDOES dengan *heading* ke arah *enemy*

4.2.7 Struktur Data pada *UseShield*

Kelas *UseShield* adalah kelas yang berfungsi dalam melakukan aksi menggunakan perisai.

Tabel 27. Atribut dan Metode Kelas *UseShield*

Attribut	**Penjelasan**
private final int CLOSE_DISTANCE	Konstanta untuk ukuran jarak dekat
Metode	**Penjelasan**
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state UseShield
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state UseShield
private List<GameObject> getTorpedosInRange(int distance	Mengembalikan list torpedo yang mengarah ke bot

4.2.8 Struktur Data pada *Run*

Kelas *Run* adalah kelas yang berfungsi dalam melakukan aksi menggunakan berlari atau menghindar dari ancaman.

Tabel 28. Atribut dan Metode Kelas *Run*

Attribut	Penjelasan
private final int MEDIUM_DISTANCE	Konstanta untuk ukuran jarak menengah
private final int DISTANCE_TO_TOLERANCE_RATIO	Konstanta untuk perhitungan rasio jarak ancaman terhadap toleransi arah datangnya
private final int RUN_ANGLE	Konstanta untuk sudut lari
Metode	**Penjelasan**
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state Run
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state Run
private List<GameObject> getBiggerEnemiesInRange(int distance)	Mengembalikan list musuh yang lebih besar dari bot pada jangkauan tertentu
private List<GameObject> getTeleporterInRange(int distance)	Mengembalikan list proyektil teleporter yang mengarah ke bot pada jangkauan tertentu
private List<GameObject> getSupernovaProjInRange(int distance)	Mengembalikan list proyektil supernova yang mengarah ke bot pada jangkauan tertentu
private PlayerAction dodgeBoundary()	Mengembalikan aksi menghindari World Boundary
private PlayerAction dodgeGasCloud()	Mengembalikan aksi menghindari Gas Cloud
private double distanceToBoundary()	Mengembalikan jarak bot ke World Boundary
private List<GameObject> getGasCloud()	Mengembalikan list Gas Cloud terurut dari yang paling dekat dengan bot
private double distanceToGasCloud()	Mengembalikan jarak bot ke Gas Cloud
private int getHeadingToGasCloud()	Mengembalikan sudut bot mengarah ke Gas Cloud

4.2.9 Struktur Data pada PickUp*Supernova*

Tabel 29. Atribut dan Metode Kelas *PickUpSupernova*

Metode	Penjelasan
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state PickUpSuperNova
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state PickUpSupernova
private PlayerAction pickUpSupernova()	Mengembalikan aksi FORWARD dengan heading mengarah ke supernova pick up
double supernoveDistance()	Mengembalikan jarak supernova ke bot, -99 bila pickup supernova tidak ada

4.2.10 Struktur Data pada *FireSupernova*

Tabel 30. Atribut dan Metode Kelas *FireSupernova*

Metode	Penjelasan
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state FireSupernova
public PlayerAction calculatePlayerAction()	Implementasi pemilihan aksi bot untuk state FireSupernova
private PlayerAction fireSupernova()	Mengembalikan aksi FIRESUPERNOVA dengan heading mengarah ke musuh yang memiliki ukuran paling besar
private int getTargetHeading()	Mengembalikan heading pada musuh yang paling besar

4.2.11 Struktur Data pada *DetonateSupernova*

Tabel 31. Atribut dan Metode Kelas *DetonateSupernova*

Metode	Penjelasan
public float calculatePriorityScore()	Implementasi perhitungan skor prioritas untuk state DetonateSupernova
public PlayerAction	Implementasi pemilihan aksi bot

calculatePlayerAction()	untuk state DetonateSupernova
private PlayerAction detonateSupernova()	Mengembalikan aksi DETONATESUPERNOVA apabila supernova bomb telah sampai pada sasaran dan berada pada jarak tertentu dari bot
private boolean isReadytoDetonate()	Mengembalikan true apabila supernova telah sampai pada musuh dengan ukuran terbesar dan berada pada jarak yang jauh dari bot
private boolean isBombExist()	Mengembalikan true apabila terdapat supernova bomb pada world
private double distanceSupernovaToBoundary()	Mengembalikan jarak supernovabomb ke boundary

4.3 Analisis Desain Solusi pada Setiap Pengujian

4.2.1 Analisis Desain Solusi Persoalan *Choose State*

Berdasarkan hasil pengujian, desain solusi persoalan *Choose State* sudah cukup efektif dalam memilih *state*. Pada hasil pengujian terdapat dominasi dari beberapa *state* yang lebih sering dipilih dibandingkan yang lain. Akan tetapi, hal ini sudah sesuai dengan optimisasi yang diinginkan. Meskipun begitu, sebelumnya dibutuhkan iterasi *trial and error* untuk mengimplementasikan fungsi menghitung *priority score* pada masing-masing *state*. Beberapa konstanta yang digunakan juga harus diatur berulang kali sampai mendapatkan angka yang optimal. Sehingga, proses implementasi menjadi sedikit sulit dan kurang sistematis.

4.2.2 Analisis Desain Solusi Persoalan *GatherFood*

Berdasarkan hasil pengujian, desain solusi persoalan GatherFood sudah cukup efektif walau masih terdapat beberapa kekurangan. Apabila terdapat 2 food dengan jarak yang sama bot dapat menjadi kebingungan dan berputar sehingga dapat membuang waktu walaupun hal ini jarang terjadi. Selain itu apabila disekitar bot terdapat banyak gas cloud disekeliling bot bot dapat berputar-putar sehingga tidak dapat menghindari gas cloud. Akan tetapi, secara keseluruhan solusi persoalan GatherFood telah dapat menjalankan fungsinya dengan baik yaitu mengumpulkan makanan sebanyak-banyaknya agar bot bertambah besar.

4.2.3 Analisis Desain Solusi Persoalan *FireTeleport*

Berdasarkan hasil pengujian dengan 3 bot *reference* dan 1 bot rancangan, desain solusi *FireTeleport* membuat aksi *FireTeleport* jarang dilakukan. Hal ini karena, tidak terdapat *enemy* yang cukup kecil sehingga pada perhitungan *priority score*, yang salah satunya menggunakan rasio ukuran bot dengan *enemy*, hasil yang didapat kecil. Sedangkan, terdapat *state* lain dengan *priority score* yang besar. Hal ini dapat pula disebabkan karena pemain dalam pertandingan tidak cukup banyak. Akan tetapi, hal ini tidak terlalu menimbulkan masalah karena urgensi teleportasi yang memang tidak terlalu besar dibandingkan *state-state* bertahan hidup dan menyerang jarak dekat. Disamping itu, perhitungan *offset* heading cukup efektif dalam meningkatkan ketepatan sasaran. Hal ini dapat dilihat dari aksi teleportasi pada gambar ….

```
Fire Teleport Priority Score: 98.52053State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 98.52053State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 98.52053State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 97.608604State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 96.57466State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 96.57466State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 95.56173State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 95.56173State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 94.30442State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 94.30442State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 94.30442State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.258865State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.258865State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.258865State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.258865State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.36119State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.36119State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.36119State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.5934State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.5934State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.82629State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.82629State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 93.82629State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.91537State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.91537State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.91537State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.02068State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.02068State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.02068State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.02068State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 92.02068State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 91.14186State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 90.32386State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 90.32386State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 90.32386State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 89.35422State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 89.35422State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 88.56414State : BotModels.States.GatherFood@3514a4c0
```

Gambar 2. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *FireTeleport* lebih kecil dari *priority score state* lain

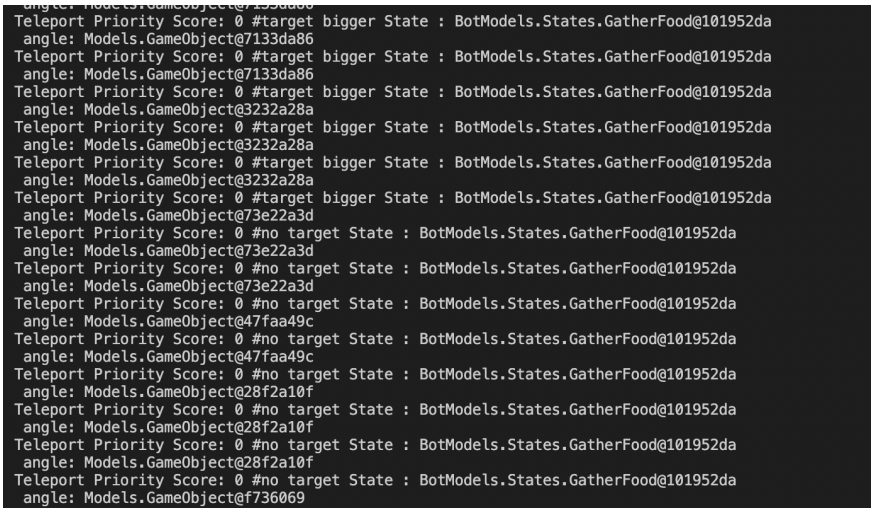
```
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
Fire Teleport Priority Score: 0.0State : BotModels.States.GatherFood@3514a4c0
```

Gambar 3. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *FireTeleport* sama dengan 0 untuk kasus ukuran bot terlalu kecil.

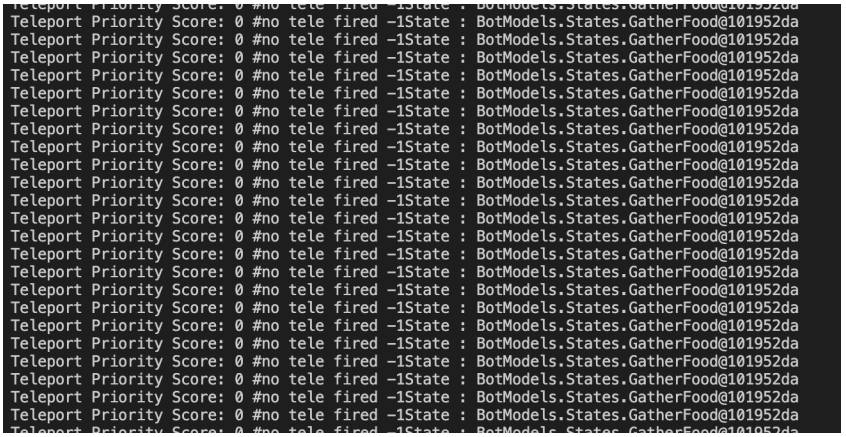
Gambar 4. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *FireTeleport* cukup besar sehingga aksi *FireTeleport* dilakukan.

4.2.4 Analisis Desain Solusi Persoalan *Teleport*

Berdasarkan hasil pengujian dengan 3 bot *reference* dan 1 bot rancangan, desain solusi *Teleport* membuat teleportasi sangat jarang dilakukan. Hal ini karena, penembakan teleporter yang sudah cukup jarang, ditambah dinamisnya perubahan status ukuran dan posisi *enemy*. Selain itu, terdapat pula masalah seperti hilangnya teleporter sesaat setelah ditembakan. Padahal, menurut aturan game, teleporter tidak akan menabrak apapun.



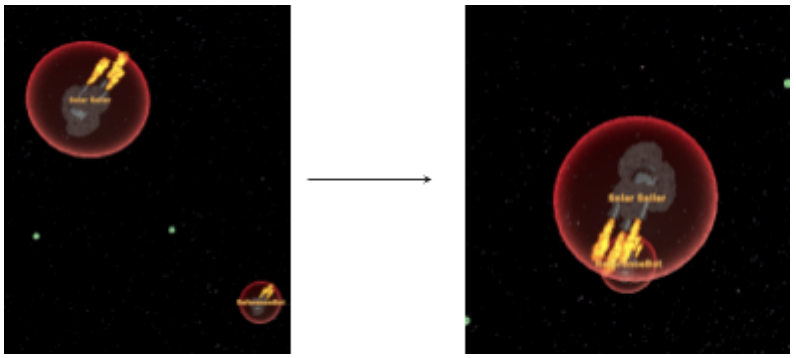
Gambar 5. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *Teleport* 0 karena semua target berukuran besar atau tidak ada target.



Gambar 6. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *Teleport* 0 karena tidak tersedia teleporter yang sudah ditembakan.

Gambar 7. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *Teleport* 0 karena teleporter tidak ditemukan, sesaat setelah penembakan.

Gambar 8. Hasil pengujian yang menunjukan kasus ketika *priority score* untuk *Teleport* 900 sehingga aksi *Teleport* dilakukan.



Gambar 9. Hasil tangkapan layar *visualiser* pada pengujian yang menunjukan saat aksi *Teleport* dilakukan.

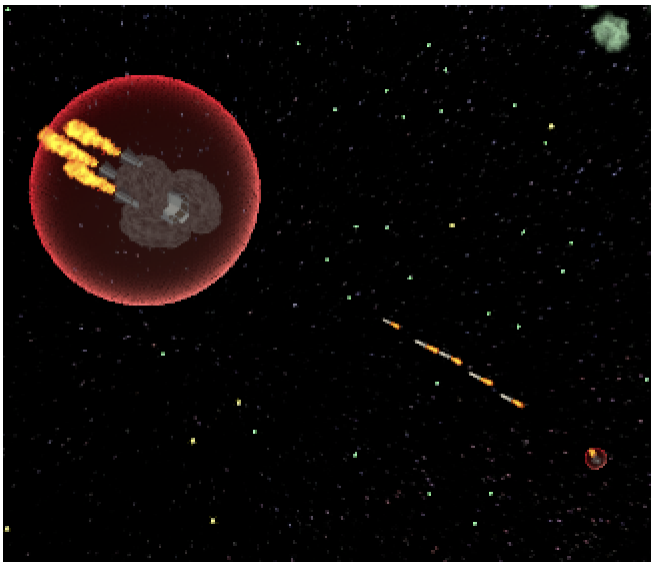
4.2.5 Analisis Desain Solusi Persoalan *TorpedoAttack*

Berdasarkan hasil pengujian dengan 3 bot *reference* dan 1 bot rancangan, desain solusi *TorpedoAttack* sudah cukup efektif dalam membuat serangan yang tepat sasaran dan menguntungkan. Perhitungan *priority score* membuat aksi

FireTorpedoes sering dilakukan. Hal ini karena jarak dekat minimum yang ditetapkan cukup besar. Akan tetapi, hal ini tidak menimbulkan masalah karena setelah menembak dengan tepat sasaran bot dapat mengonsumsi ukuran *enemy* yang ditembaknya. Meskipun begitu, untuk *enemy* bot yang memiliki kemampuan mendeteksi *torpedo salvo projectile* kemudian dapat mengaktifkan shield atau menghindarinya, desain solusi *TorpedoAttack* ini akan berkurang efektifitasnya.

```
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.TorpedoAttack@76c3e77a
lState : BotModels.States.GatherFood@101952da

lState : BotModels.States.GatherFood@101952da

lState : BotModels.States.GatherFood@101952da

lState : BotModels.States.GatherFood@101952da

lState : BotModels.States.GatherFood@101952da

lState : BotModels.States.GatherFood@101952da
```

Gambar 10. Hasil pengujian yang menunjukkan seringnya aksi *FireTorpedoes* dilakukan.



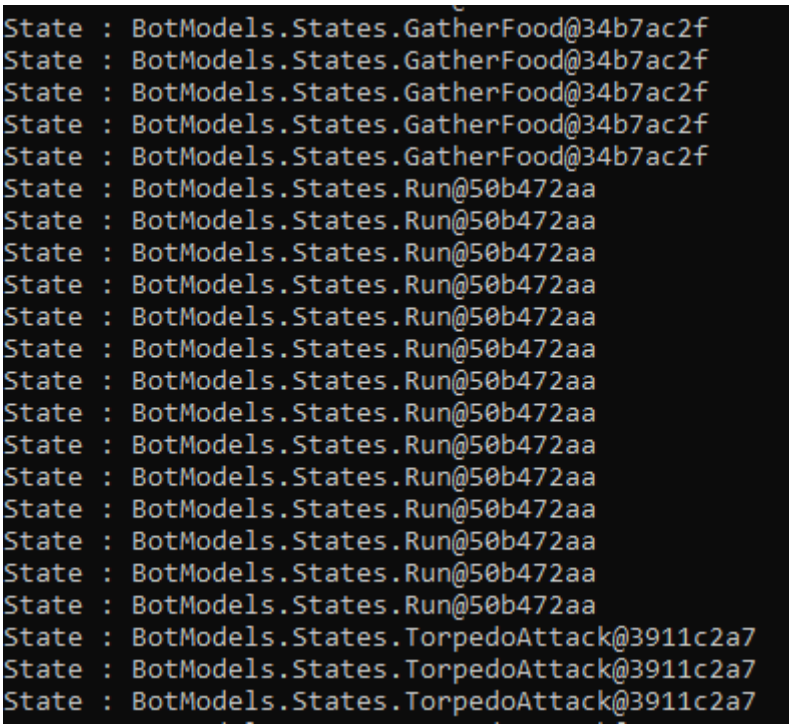
Gambar 11. Hasil tangkapan layar pada pengujian yang menunjukkan ketepatan sasaran penembakan torpedo.

4.2.6 Analisis Desain Solusi Persoalan *UseShield*

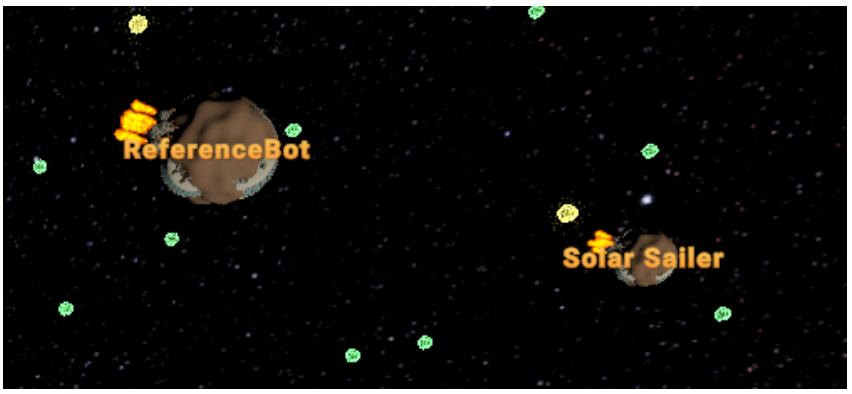
Dari hasil percobaan dengan 2 bot reference, 1 bot lain, dan 1 bot rancangan, setelah beberapa kali pengujian untuk mendapatkan angka-angka yang tepat, penggunaan shield sudah cukup baik dan tepat waktu untuk mencegah *hit* dari torpedo. Penggunaan shield yang tepat ini memberikan waktu bagi bot untuk melakukan serangan balik, dan jika musuh tidak memiliki mekanisme shield, maka bot rancangan akan memiliki keuntungan yang besar.

4.2.7 Analisis Desain Solusi Persoalan *Run*

Untuk sub permasalahan *Run*, hasil pengujian beberapa kali sudah tepat mengevaluasi waktu untuk berlari atau menghindar dari musuh. Dalam pengujian, aksi ini cukup jarang dilakukan, karena musuh yang diuji belum memiliki kemampuan untuk menembak teleport ataupun supernova. Akan tetapi, untuk menghindar dalam keadaan musuh lebih besar yang mengejar dapat berjalan dengan aman. Akan tetapi, pada beberapa *edge case,* yaitu ketika bot terpojok antara musuh dengan obstacle (*gas cloud*/ *boundary*), bot hanya berputar-putar akibat kebingungan untuk memilih terkena hit *gas cloud/ boundary* atau termakan musuh.



Gambar 12. Hasil pengujian yang menunjukkan aksi *Run* dijalankan.



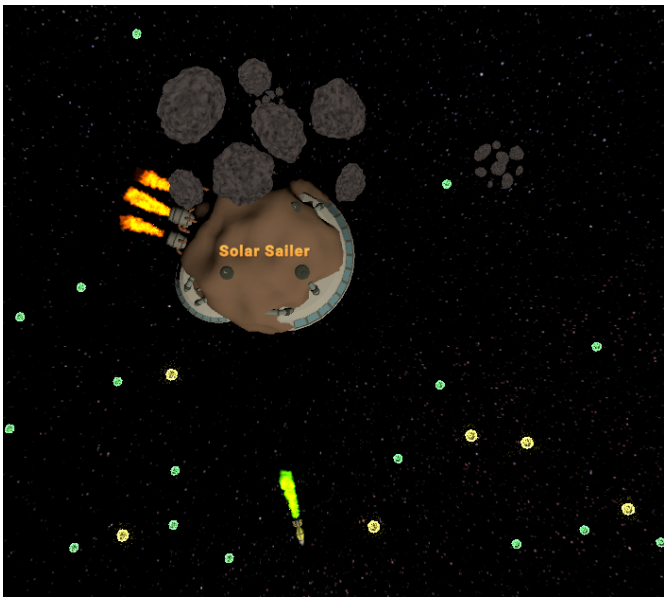
Gambar 13. Hasil tangkapan layar saat aksi *Run*.

4.2.8 Analisis Desain Solusi Pick Up *Supernova*

Berdasarkan hasil pengujian, desain solusi persoalan PickUpSupernova telah menjalankan tugasnya dengan baik. Supernova berhasil di pick up apabila jarak supernova degan bot dekat.

4.2.9 Analisis Desain Solusi Fire *Supernova*

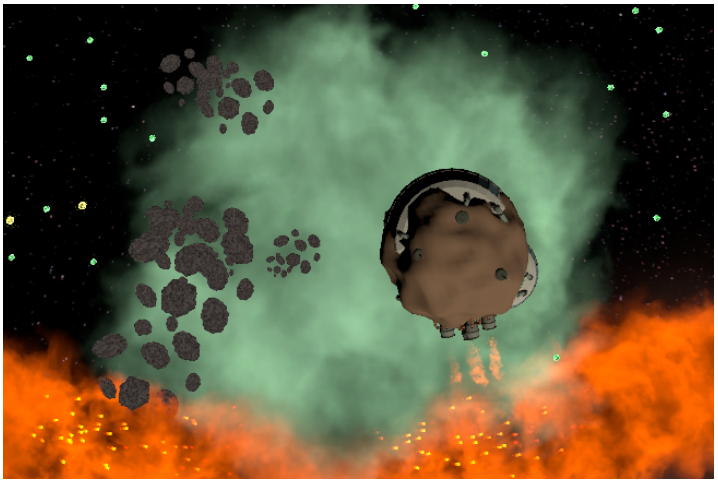
Berdasarkan hasil pengujian, desain solusi persoalan FireSupernova sudah efektif. Bot telah dapat menembakan supernova tepat sasaran yaitu musuh yang pada saat itu memiliki ukuran paling besar.



Gambar 14. Hasil tangkapan layar saat fire supernova.

4.2.10 Analisis Desain Solusi Detonate Supernova

Berdasarkan hasil pengujian, desain solusi persoalan DetonateSupernova telah menjalankan fungsi dengan baik. Supernova bomb yang di tembakkan bot berhasil diledakkan pada area musuh dan tidak mengenai diri sendiri.



Gambar 15. Hasil tangkapan layar saat detonateSupernova.

BAB 5
PENUTUP

5.1 Kesimpulan

Berdasarkan penjabaran di atas, dapat disimpulkan bahwa program bot untuk menyelesaikan persoalan *Galaxio* kami sudah cukup efektif dalam mencapai tujuan optimasi strategi greedy yaitu memaksimalkan skor pada pertandingan dan bertahan selama mungkin. Dalam menyelesaikan persoalan ini, kami melakukan dekomposisi pada persoalan Galaxio menjadi beberapa sub persoalan. Di antaranya, sub persoalan *ChooseState, GatherFood, FireTeleport, Teleport, TorpedoAttack, UseShield, Run, PickUpSupernova, FireSupernova,* dan *DetonateSupernova*.

Untuk sub persoalan *ChooseState*, strategi yang dipilih adalah memilih *state* dengan skor prioritas tertinggi pada *state game* saat ini. Setiap skor prioritas dihitung secara internal pada masing-masing *state*. Sedangkan, untuk sub persoalan *GatherFood,* strategi yang dipilih adalah mencari makanan sebanyak-banyaknya sambil menuju ke area dengan banyak makanan sekaligus menghindari benda-benda yang bisa membahayakan bot. Selanjutnya, untuk sub persoalan *FireTeleport,* strategi yang dipilih adalah menembakan teleporter ke arah enemy yang terkecil dan lebih kecil dari ukuran bot - biaya penembakan dengan *offset*.

Setelah itu, untuk sub persoalan *Teleport,* strategi yang dipilih adalah melakukan aksi *Teleport* ketika teleporter berada di sekitar *enemy* yang lebih kecil dari ukuran bot saat ini. Sedangkan, untuk sub persoalan *TorpedoAttack,* strategi yang dipilih adalah menembak ke arah musuh yang dekat. Lalu, untuk sub persoalan *UseShield,* strategi yang dipilih adalah menggunakan *shield* ketika terdapat torpedo dengan jumlah tertentu yang mengarah ke bot. Kemudian, untuk sub persoalan *Run,* strategi yang dipilih adalah menghindar dari ancaman pada jarak tertentu.

Selain itu, untuk sub persoalan *PickUpSupernova,* strategi yang dipilih adalah mem-*pick up* supernova dalam radius jarak tertentu yang cukup dekat. Untuk sub persoalan *FireSupernova,* strategi yang dipilih adalah menembak ke arah lokasi enemy terbesar. Terakhir, untuk sub persoalan *DetonateSupernova,* strategi yang dipilih adalah melakukan aksi *Detonate* ketika supernova berada cukup jauh dari bot, berada di sekitar *enemy* terbesar, atau akan keluar dari *boundary map*.

5.2 Saran, Komentar, dan Refleksi

Setelah mengerjakan Tugas Besar ini, kami dapat lebih memahami implementasi strategi *greedy* pada persoalan-persoalan selain persoalan teoritis yang telah dipelajari di kelas. Kami juga menjadi lebih paham tentang cara memetakan suatu persoalan menjadi elemen-elemen algoritma *greedy*. Selain itu, kami juga menjadi lebih terlatih untuk memilih strategi greedy yang paling efektif dan efisien untuk suatu persoalan tertentu.

Di samping banyak mempelajari tentang algoritma greedy, kami juga banyak mendapatkan ilmu dari hasil eksplorasi program game. Selain itu, karena program bot menggunakan bahasa Java, kami juga menjadi lebih fasih dalam menerapkan konsep pemrograman berbasis objek. Terlebih, karena kelompok kami menerapkan konsep *inheritance* dan *abstract base class*.

Sebagai refleksi, kami ingin mengevaluasi diri agar kedepannya dapat membuat timeline pengerjaan yang lebih jelas dan rapi. Kami juga harus lebih disiplin dalam mengerjakan bagian tugas masing-masing pada tenggat waktu yang telah disepakati. Hal ini untuk menghindari pengerjaan yang terlalu dekat dengan *deadline*, khususnya pengerjaan laporan. Sebagai saran, kami ingin memberikan saran kepada pihak asisten agar dapat memberikan tenggat waktu yang lebih panjang untuk pengerjaan Tugas Besar ini. Hal ini karena, dibutuhkan waktu yang cukup banyak untuk melakukan eksplorasi cara kerja *game* sebelum memulai proses perancangan strategi *greedy*. Terlebih, game Galaxio memiliki beberapa *bug* serta memiliki starter-bot yang kurang kompatibel. Terakhir, kami juga menyarankan diadakannya asistensi selama pengerjaan tugas agar dapat lebih membantu pengerjaan.

DAFTAR PUSTAKA

https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/building-a-bot.md
https://github.com/EntelectChallenge/2021-Galaxio/blob/develop/starter-bots/README.md
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf