LAPORAN HASIL TUGAS BESAR 1

Pemanfaatan Algoritma Greedy dalam Aplikasi Permainan “Worms”

DIAJUKAN UNTUK MEMENUHI TUGAS MATA KULIAH

IF2211 - Strategi Algoritma

SEMESTER II TAHUN 2020/2021

Disusun oleh :

Kelompok 60 : send hlp pls

Faris Aziz	13519065
Maximillian Lukman	13519153
Richard Rivaldo	13519185

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

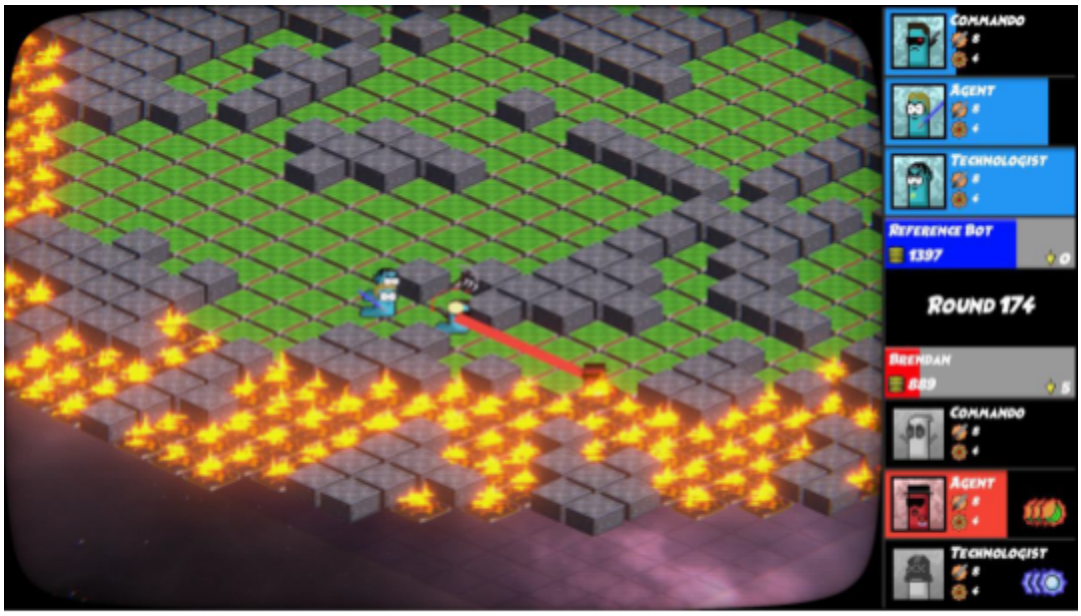
2021

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

1.1 Deskripsi Tugas

Worms adalah sebuah turned-based game yang memerlukan strategi untuk memenangkannya. Setiap pemain akan memiliki 3 worms dengan perannya masing-masing. Pemain dinyatakan menang jika ia berhasil bertahan hingga akhir permainan dengan cara mengeliminasi pasukan worms lawan menggunakan strategi tertentu.



Gambar 1. Contoh tampilan permainan Worms

Pada tugas besar pertama Strategi Algoritma ini, gunakanlah sebuah game engine untuk mengimplementasikan permainan Worms. Game engine dapat diperoleh pada laman berikut:
https://github.com/EntelectChallenge/2019-Worms

Tugas mahasiswa adalah mengimplementasikan seorang “pemain” Worms, dengan menggunakan strategi greedy untuk memenangkan permainan. Untuk mengimplementasikan seorang “pemain” tersebut, mahasiswa disarankan melanjutkan program yang terdapat pada starter bot di dalam starter pack pada laman berikut ini:
(https://github.com/EntelectChallenge/2019-Worms/releases/tag/2019.3.2)

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

1. Peta permainan berukuran 33x33 cells. Terdapat 4 tipe cell, yaitu air, dirt, deep space, dan lava yang masing-masing memiliki karakteristik berbeda. Cell dapat memuat powerups yang bisa diambil oleh worms yang berada pada cell tersebut.

2. Di awal permainan, setiap pemain akan memiliki 3 pasukan worms dengan peran dan nilai health points yang berbeda, yaitu::
   a. Commando
   b. Agent
   c. Technologist

3. Pada setiap round, masing-masing pemain dapat memberikan satu buah command untuk pasukan worm mereka yang masih aktif (belum tereliminasi). Berikut jenis-jenis command yang ada pada permainan:
   a. Move
   b. Dig
   c. Shot
   d. Do Nothing
   e. Banana Bomb
   f. Snowball
   g. Select

4. Command dari kedua pemain akan dieksekusi secara bersamaan (bukan sekuensial) dan akan divalidasi terlebih dahulu. Command juga akan dieksekusi sesuai urutan prioritas tertentu.

5. Beberapa command, seperti shot dan banana bomb dapat memberikan damage pada worms target yang terkena serangan, sehingga mengurangi health pointsnya. Jika health points suatu worm sudah habis, maka worm tersebut dinyatakan tereliminasi dari permainan.

6. Permainan akan berakhir ketika salah satu pemain berhasil mengeliminasi seluruh pasukan worms lawan atau permainan sudah mencapai jumlah round maksimum (400 rounds).

Adapun peraturan yang lebih lengkap dari permainan Worms, dapat dilihat pada laman:
https://github.com/EntelectChallenge/2019-Worms/blob/develop/game-engine/game-rules.md.

1.2 Spesifikasi Tugas

Pada tugas besar kali ini, anda diminta untuk membuat sebuah bot untuk bermain permainan Worms yang telah dijelaskan sebelumnya. Untuk memulai, anda dapat mengikuti panduan singkat sebagai berikut.

1. Download latest release starter pack.zip dari tautan berikut.
   https://github.com/EntelectChallenge/2019-Worms/releases/tag/2019.3.2.
2. Untuk menjalankan permainan, kalian butuh beberapa requirement dasar sebagai berikut.
   a. Java (minimal Java 8):
      https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html
   b. IntelIJ IDEA: https://www.jetbrains.com/idea/
   c. NodeJS: https://nodejs.org/en/download/
3. Untuk menjalankan permainan, kalian dapat membuka file “run.bat” (Untuk Windows/Mac dapat buka dengan double-click, Untuk Linux dapat menjalankan command “make run”).
4. Secara default, permainan akan dilakukan diantara reference bot (default-nya berbahasa JavaScript) dan starter bot yang disediakan. Untuk mengubah hal tersebut, silahkan edit file “game-runner-config.json”. Anda juga dapat mengubah file “bot.json” untuk mengatur informasi terkait bot anda.
5. Silahkan bersenang-senang dengan memodifikasi bot yang disediakan di starter-bot. Ingat bahwa bot kalian harus menggunakan bahasa Java dan di-build

menggunakan IntelIJ. Dilarang menggunakan kode program tersebut untuk pemainnya atau kode program lain yang diunduh dari Internet. Mahasiswa harus membuat program sendiri, tetapi belajar dari program yang sudah ada tidak dilarang.

6. (Optional) Anda dapat melihat hasil permainan dengan menggunakan visualizer berikut:
   https://github.com/dlweatherhead/entelect-challenge-2019-visualiser/releases/tag/v1.0f1

Strategi greedy yang diimplementasikan tiap kelompok harus dikaitkan dengan fungsi objektif dari permainan itu sendiri, yaitu memenangkan permainan dengan cara mengeliminasi seluruh worms lawan dengan senjata dan skill yang sudah disediakan dalam permainan. Salah satu contoh pendekatan greedy yang bisa digunakan (pendekatan tak terbatas pada contoh ini saja) adalah menyerang pasukan lawan dengan senjata dengan hitpoint / damage terbesar. Buatlah strategi greedy terbaik, karena setiap "pemain" dari masing-masing kelompok akan diadu satu sama lain dalam suatu kompetisi Tubes 1 (secara daring).

Strategi greedy harus dituliskan secara eksplisit pada laporan, karena akan diperiksa pada saat demo apakah strategi yang dituliskan sesuai dengan yang diimplementasikan. Tiap kelompok dapat menggunakan kreativitas mereka dalam menyusun strategi greedy untuk memenangkan permainan. Implementasi pemain harus dapat dijalankan pada game engine yang telah disebutkan pada spesifikasi tugas besar, serta dapat dikompetisikan dengan pemain dari kelompok lain.

BAB II
LANDASAN TEORI

2.1 Algoritma Greedy

Algoritma *Greedy* adalah salah satu algoritma yang paling populer dalam memecahkan permasalahan yang membutuhkan optimisasi. Secara khusus, ada dua permasalahan optimisasi, yaitu persoalan memaksimasi (*maximization*) dan persoalan meminimasi (*minimization*).

Secara formal, Algoritma *Greedy* dapat didefinisikan sebagai algoritma yang memecahkan permasalahan langkah demi langkah (*step by step*). Untuk setiap langkah pemecahan permasalahan algoritma ini harus bisa mengambil pilihan terbaik yang ditemuinya pada saat itu tanpa perlu memerhatikan konsekuensi yang mungkin muncul pada masa depan. Setiap pemilihan ini berarti bahwa kita memilih setiap solusi yang bersifat optimum lokal. Pada akhirnya, kita berharap bahwa kita bisa mendapatkan solusi optimum global yang terdapat di antara solusi-solusi optimum lokal tersebut.

Hal ini dikarenakan solusi yang kita pilih belum tentu merupakan solusi yang optimum secara keseluruhan dan bisa jadi hanya merupakan solusi sub optimum. Penyebabnya adalah algoritma jenis ini tidak bekerja secara menyeluruh terhadap semua kemungkinan solusi yang ada. Selain itu, juga terdapat beberapa fungsi seleksi yang berbeda pada saat pemilihan solusi, sehingga kita harus bisa memilih fungsi yang tepat untuk mendapatkan solusi optimal.

Berikut merupakan elemen-elemen yang menggambarkan sebuah Algoritma *Greedy*.

1. Himpunan Kandidat (C)
   Himpunan yang berisi kandidat yang akan dipilih pada setiap langkah, misalnya simpul sisi di dalam graf, job, task, koin, benda, karakter.
2. Himpunan Solusi (S)
   Himpunan yang berisi kandidat yang sudah dipilih.

3. Fungsi Solusi
   Fungsi yang digunakan untuk menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi paling optimal.

4. Fungsi Seleksi (*Selection Function*)
   Fungsi yang digunakan untuk memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.

5. Fungsi kelayakan (*Feasibility*)
   Memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak).

6. Fungsi Obyektif
   Fungsi yang digunakan untuk memaksimumkan atau meminimumkan solusi yang kita miliki.

2.2 Pemanfaatan *Game Engine*

Game Engine, seperti namanya adalah sebuah mesin yang dirancang secara khusus untuk menjalankan program atau aplikasi permainan. Pada permainan Worms, *game engine* yang digunakan dibuat dengan Unity dan sudah tersedia dari *starter pack* yang berasal dari Entelect, penyelenggara *challenge* ini. Sebagai tambahan, *game engine* ini juga dilengkapi dengan sebuah *visualizer* yang bisa diunduh secara terpisah untuk menampilkan program dengan grafis layaknya permainan komputer biasa.

Dalam berinteraksi dengan program yang kita bangun, *game engine* ini menggunakan file-file berformat .json untuk mengambil maupun menyimpan *state* permainan dan atribut-atribut penting lainnya dalam permainan. Misalnya, pada `game-config.json`, *developer* telah mendefinisikan atribut-atribut yang akan dipakai di dalam permainan dan *game engine* menjadikan file ini sebagai *source file* permainan. Pada direktori `match-logs` yang telah diisi dengan hasil *run* program, kita juga bisa melihat ada file `GlobalState.json` yang menyimpan perubahan dan kondisi permainan untuk setiap ronde permainan.

Game engine yang disediakan oleh Entelect telah memiliki konfigurasi *default* permainan yang disimpan dalam file dengan format tersebut. Artinya, kita tidak perlu lagi mengatur *settings* yang ada dalam permainan tersebut dan bisa langsung

menjalankan permainan yang ada. Dalam hal ini, *bot* dasar permainan sudah ada melalui konfigurasi tersebut dan tidak perlu ditambahkan dengan sendirinya.

Namun, kita tetap bisa melakukan perubahan pada bagian-bagian tertentu dalam permainan dengan memodifikasi file-file .json ini. Misalnya, untuk mengubah nama *bot* atau nama pemain yang kita inginkan sebagai *in-game name* kita. Perubahan ini dapat kita aplikasikan dengan mengubah atribut `nickName` di dalam file bot.json yang ada pada direktori Java maupun bahasa lainnya. Di file ini kita juga bisa mengganti direktori, nama, atau bahasa yang digunakan dalam pembuatan *bot worms.*

Ketika kita ingin melakukan *testing* atau *sparring* dengan *bot* baru yang bukan Reference Bot, kita bisa menambahkan *bot* ini dengan mengganti atribut `player-a` dan `player-b` yang ada di dalam permainan dengan direktori *bot* yang ingin dipertandingkan. Kita juga bisa mengganti target direktori yang digunakan untuk menyimpan *state* permainan tiap rondenya.

Awalnya, program *bot* yang kita miliki belum mempunyai atribut permainan yang lengkap. Atribut-atribut ini misalnya adalah belum terdefinisinya beberapa *command* khusus seperti Banana Bombs, Snowball, dan Select, juga termasuk atribut-atribut yang digunakan untuk mengecek keberhasilan *command* ini. *Method-method* yang sudah ada juga masih bisa diperbaiki menjadi lebih baik.

Oleh karena itu, sebelum melakukan pengimplementasian lebih lanjut, penulis melengkapi bagian-bagian yang hilang ini terlebih dahulu. Dalam bahasa Java, bagian ini dibuat dengan paradigma berorientasi objek, sehingga didefinisikan kelas-kelas baru untuk menampung atribut tersebut. Kita tidak bisa langsung membuat atribut baru, misalnya `count` untuk menggantikan `countSnowball` yang ada di dalam permainan. Hal ini akan mengakibatkan atribut tersebut menjadi tidak terdefinisi.

Maka, program harus mengambil atribut-atribut yang ada di dalam file .json yang digunakan *game engine*. Hal ini dilakukan dengan menggunakan anotasi `SerializedName` pada JavaJSONObject untuk mengambil atribut-atribut pada file .json tersebut dan memasukkannya pada atribut kelas tertentu tanpa perlu diinisialisasi. Dengan demikian, program yang kita bangun sekarang bisa

berinteraksi dengan *state game engine* dan permainan tersebut dengan perantara file .json ini.

Untuk memudahkan *build* program, dapat digunakan Intellij IDEA. Intellij IDEA telah menyediakan Maven Toolbox dan kita bisa memanfaatkan fitur `Install` di dalam `Lifecycle` untuk menghasilkan file .jar. File ini akan digunakan dalam proses *running* permainan. Dengan melakukan konfigurasi yang telah disebutkan sebelumnya, kita bisa melakukan *running* dengan menjalankan program `run.bat`. Program ini akan menjalankan *game engine* secara otomatis di dalam Command Prompt atau Terminal. Permainan ini akan ditampilkan dalam format *text-based*.

BAB III
PEMANFAATAN STRATEGI GREEDY

3.1 Pemetaan Algoritma *Greedy* pada Permasalahan Permainan *Worms*

a. Pemetaan Umum Permasalahan Permainan

Berikut merupakan pemetaan permasalahan permainan *Worms* secara umum menjadi elemen-elemen penyusun Algoritma *Greedy*.

Elemen Algoritma *Greedy*	Pemetaan pada Permainan *Worms*
Himpunan Kandidat (C)	Perintah-perintah yang mungkin dijalankan oleh *worm* di dalam permainan. Perintah yang dimaksud dapat merupakan perintah untuk menembak lawan, melakukan pergerakan biasa, menghancurkan *dirt*, mengeluarkan perintah *select* untuk memilih *worm* secara khusus, atau sama sekali tidak melakukan apa-apa.
Himpunan Solusi (S)	Perintah terbaik yang dipilih sesuai dengan kondisi setiap ronde permainan selama permainan berlangsung.
Fungsi Solusi	Memeriksa bahwa perintah yang dipilih adalah perintah yang terdefinisi atau tidak di dalam permainan. Jika tidak, tentunya perintah ini bukan solusi yang valid untuk dijalankan oleh *game engine*.
Fungsi Seleksi (*Selection Function*)	Pemilihan perintah yang sesuai dengan prioritas strategi yang digunakan dalam *bot* yang telah dibangun dalam implementasi permainan *Worms*. Pembangunan prioritas dalam bahasa pemrograman bisa dilakukan dengan membuat fungsi penyeleksian yang terurut sesuai dengan prioritas strategi yang dimiliki.
Fungsi Kelayakan (*Feasibility*)	Memeriksa bahwa semua kondisi yang diperlukan untuk mengeksekusi perintah yang dipilih sudah terpenuhi atau belum. Validasi dilakukan terhadap komponen-komponen permainan, misalnya jumlah items yang masih dimiliki oleh *worm* pemain, jarak sebuah *worm* lawan yang masih dalam jangkauan *worm*

	pemain, atau jenis *cell* yang ada di dalam *map* maupun valid tidaknya koordinat yang ingin dituju oleh *worm* pemain saat melakukan pergerakan.
Fungsi Objektif	Memenangkan permainan *worms*, baik dengan membunuh semua *worm* lawan, maupun dengan mengumpulkan poin yang sebanyak-banyaknya dengan harapan bahwa ketika mencapai ronde maksimum, total poin yang kita miliki lebih banyak dibandingkan poin lawan sehingga kita memenangkan permainan.

b. Pemetaan Pergerakan ke dalam Algoritma *Greedy*

Pergerakan merupakan salah satu aspek paling penting dalam permainan *Worms*, karena dengan pergerakan yang baik kita bisa mendapatkan keuntungan berupa poin maupun posisi yang lebih baik dibandingkan dengan lawan. Berikut merupakan pemetaan pergerakan yang ada di dalam permainan *Worms* ke dalam algoritma *greedy*.

Elemen Algoritma *Greedy*	Pemetaan pada Permainan *Worms*
Himpunan Kandidat (C)	Perintah-perintah yang berkaitan dengan pergerakan dalam permainan *Worms*. Perintah tersebut termasuk *move*, *dig*, ataupun *do nothing*.
Himpunan Solusi (S)	Perintah terbaik yang dipilih sesuai dengan kondisi peta di sekitar *worms* pada setiap ronde permainan selama permainan berlangsung dan *worms* yang dimiliki masih hidup.
Fungsi Solusi	Memeriksa bahwa perintah yang dipilih adalah perintah yang terdefinisi atau tidak di dalam permainan. Jika tidak, tentunya perintah ini bukan solusi yang valid untuk dijalankan oleh *game engine*.
Fungsi Seleksi (*Selection Function*)	Pemilihan perintah yang sesuai dengan prioritas strategi yang digunakan dalam *bot* yang telah dibangun dalam implementasi permainan *Worms*. Pembangunan prioritas dalam bahasa pemrograman bisa dilakukan dengan membuat fungsi penyeleksian yang terurut sesuai dengan

	prioritas strategi yang dimiliki.
Fungsi Kelayakan (*Feasibility*)	Memeriksa bahwa semua kondisi yang diperlukan untuk mengeksekusi perintah yang dipilih sudah terpenuhi atau belum. Validasi dilakukan terhadap validitas dari koordinat, tipe *cell*, maupun *occupied* atau tidaknya suatu *cell* yang ingin dituju oleh *worm* pemain.
Fungsi Objektif	Memaksimalkan poin dan kebutuhan strategis yang bisa didapatkan *worm* pemain ketika melakukan pergerakan. Dalam hal ini, juga harus diminimalisir kemungkinan untuk *worms* pemain tidak melakukan apa-apa (karena tidak menghasilkan poin dan merugikan) serta mengurangi bahaya yang mungkin ditemui saat bergerak.

c. Pemetaan Penyerangan ke dalam Algoritma *Greedy*

Strategi menyerang juga merupakan salah satu aspek paling penting dalam permainan *Worms*, karena strategi ini menjadi fundamental bagi *worm* pemain ketika bertemu dengan *worm* lawan sehingga *worm* pemain bisa mengalahkan *worm* musuh. Berikut merupakan pemetaan penyerangan yang ada di dalam permainan *Worms* ke dalam algoritma *greedy*.

Elemen Algoritma *Greedy*	Pemetaan pada Permainan *Worms*
Himpunan Kandidat (C)	Perintah-perintah yang berkaitan dengan penyerangan dalam permainan *Worms*. Perintah tersebut termasuk *shoot* dan *banana bomb* atau *snowball*, serta *select command* yang bisa digunakan dalam kondisi darurat.
Himpunan Solusi (S)	Perintah terbaik yang dipilih sesuai dengan kondisi *worm* lawan yang ada di sekitar *worms* pemain pada setiap ronde permainan selama permainan berlangsung dan *worms* kedua pihak masih hidup dan bisa menyerang.
Fungsi Solusi	Memeriksa bahwa perintah yang dipilih adalah perintah yang terdefinisi atau tidak di dalam permainan. Jika tidak, tentunya perintah ini bukan solusi yang valid untuk dijalankan oleh

	game engine.
Fungsi Seleksi (*Selection Function*)	Pemilihan perintah yang sesuai dengan prioritas strategi yang digunakan dalam *bot* yang telah dibangun dalam implementasi permainan *Worms*. Pembangunan prioritas dalam bahasa pemrograman bisa dilakukan dengan membuat fungsi penyeleksian yang terurut sesuai dengan prioritas strategi yang dimiliki.
Fungsi Kelayakan (*Feasibility*)	Memeriksa bahwa semua kondisi yang diperlukan untuk mengeksekusi perintah yang dipilih sudah terpenuhi atau belum. Validasi dilakukan terhadap validitas dari *range* tembakan, garis tembakan, keberadaan musuh, maupun jumlah *special attack* yang dimiliki oleh *worm* pemain serta kompatibilitas *worm* dalam menggunakan *special attack* tersebut.
Fungsi Objektif	Memaksimalkan output *damage* dan poin yang bisa didapatkan *worm* pemain ketika melakukan penyerangan. Dalam hal ini, juga harus dimaksimalkan efek-efek yang bisa didapat dari *special attack*, seperti *frozen duration* dan *impact damage*. Selain itu, dengan *select command* juga bisa diminimalkan bahaya yang bisa didapat *worm* pemain ketika diserang oleh musuh sehingga tidak diserang terus-menerus.

d. Pemetaan *Endgame* ke dalam Algoritma *Greedy*

Endgame merupakan fase saat permainan mencapai tingkat akhir, yaitu dalam permainan *worms* ditandai dengan area gerak yang semakin sempit dan *round* yang sudah bernilai ratusan. *Endgame* menjadi ronde-ronde yang sangat menentukan keberhasilan *worms* pemain dalam memenangkan permainan. Berikut merupakan pemetaan strategi *endgame* yang ada di dalam permainan *Worms* ke dalam algoritma *greedy*.

Elemen Algoritma *Greedy*	Pemetaan pada Permainan *Worms*
Himpunan Kandidat (C)	Perintah-perintah yang berkaitan dengan *endgame* dalam permainan *Worms*. Perintah tersebut termasuk perintah-perintah pergerakan dan penyerangan yang ada di dalam *Worms*.

Himpunan Solusi (S)	Perintah terbaik yang dipilih sesuai dengan kondisi *worm* pemain maupun *worm* lawan yang masih hidup serta kondisi peta yang mulai mengecil dan menyebabkan area pergerakan lebih sedikit.
Fungsi Solusi	Memeriksa bahwa perintah yang dipilih adalah perintah yang terdefinisi atau tidak di dalam permainan. Jika tidak, tentunya perintah ini bukan solusi yang valid untuk dijalankan oleh *game engine*.
Fungsi Seleksi (*Selection Function*)	Pemilihan perintah yang sesuai dengan prioritas strategi yang digunakan dalam *bot* yang telah dibangun dalam implementasi permainan *Worms*. Pembangunan prioritas dalam bahasa pemrograman bisa dilakukan dengan membuat fungsi penyeleksian yang terurut sesuai dengan prioritas strategi yang dimiliki.
Fungsi Kelayakan (*Feasibility*)	Memeriksa bahwa semua kondisi yang diperlukan untuk mengeksekusi perintah yang dipilih sudah terpenuhi atau belum. Validasi dilakukan terhadap validitas dari *special attack* yang tersisa, posisi *worm* pemain dan lawan, jumlah *health* dari pemain dan lawan, serta koordinat dan tipe *block* dalam peta yang menjadi bagian dari pergerakan *worm*.
Fungsi Objektif	Jika kita sudah berada dalam posisi yang unggul dalam artian memiliki *health* yang lebih tinggi dibanding lawan, kita bisa mencoba untuk menyerang dan membunuh *worm* lawan sehingga kita bisa menjadi pemenang permainan. Jika tidak, kita bisa mencoba kabur dari serangan lawan untuk memaksimalkan poin yang didapat dari pergerakan dan memungkinkan terjadinya *comeback* meskipun darah *worm* yang tersisa tinggal 1%.

3.2 Eksplorasi Alternatif Strategi *Greedy* dalam Permainan *Worms*

Mekanisme permainan *Worms* yang cukup banyak dan kompleks membuat banyak sekali strategi yang bisa diimplementasikan untuk mencapai tujuan dari permainan ini, yaitu memenangkan permainan. Oleh karena itu, eksplorasi strategi ini

akan dibagi ke dalam kelompok-kelompok strategi untuk memudahkan klasifikasi strategi *Worms*. Berikut merupakan alternatif-alternatif yang bisa diimplementasikan.

a. Strategi Pergerakan *Worms*

Di dalam *Worms* banyak sekali strategi pergerakan yang bisa kita implementasikan. Setiap strategi pergerakan juga bisa dipakai tergantung dari kondisi permainan yang terus berubah-ubah selama permainan berlangsung.

Salah satu strategi pergerakan yang mungkin untuk dipakai adalah pergerakan *worm* mengikuti salah satu *worm* teman yang menjadi target *following*. Target *worm* yang bisa dipilih adalah salah satu teman terdekat dari setiap *worm* yang dipilih oleh *game engine* untuk setiap ronde, atau ditetapkan secara pasti dari awal hingga akhir permainan *worm* mana yang harus dipilih.

Selain itu, *worm* pemain bisa memilih untuk mencoba dalam mendekati *power up* yang paling dekat di sekitarnya. Hal ini bertujuan supaya *worm* kita bisa mendapatkan *power up* yang ada di sekitarnya sehingga kita juga bisa menambah poin dan mengembalikan *health* yang mungkin sudah berkurang.

Ketika bertemu dengan *cell* bertipe tertentu, seperti *dirt* dan *lava* maupun *deep space*, harus diberikan perintah pergerakan yang tepat dalam menangani setiap jenisnya. Ketika bertemu dengan *dirt*, kita bisa menggali dan menghancurkan *dirt* ini dengan tujuan untuk mendapatkan poin yang lebih banyak. Namun, ketika kita bertemu dengan *block* bertipe *lava* atau *deep space* yang merugikan *worm* kita, kita bisa mencari *block* lain yang ada di sekitarnya atau bahkan bergerak ke arah yang sebaliknya sehingga kita tidak menuju ke blok tersebut. Karena *lava* bergerak dari pinggiran peta menuju ke tengah peta dan menyisakan beberapa area kecil, kita bisa memerintahkan *bot* kita untuk bergerak dengan orientasi menuju ke tengah peta.

b. Strategi Menyerang

Ketika menyerang, kita tentunya menginginkan *damage output* yang semaksimal mungkin. Dalam hal tersebut kita juga bisa mendapatkan poin yang lebih besar dari *damage* tersebut maupun dengan membunuh musuh. Oleh karena itu, kita harus memaksimalkan penggunaan *resources* yang dimiliki setiap *worm*.

Misalnya, untuk *worm* yang memiliki atribut khusus seperti *Banana Bombs* dan *Snowball*, *worm* ini harus memaksimalkan penggunaan atribut tersebut dengan cara memakai atribut tersebut terlebih dahulu. Setelah habis, barulah *worm* tersebut bisa menggunakan serangan biasa.

Untuk *Banana Bombs*, karena atribut ini bisa menghancurkan *dirt* juga, maka mungkin saja kita memilih untuk menggunakan atribut ini untuk menghancurkan

dirt dalam jumlah yang banyak (misalnya 10 *dirt*). Untuk *Snowball*, kita bisa memaksimalkan *freeze duration* dari *worm* yang terkena efek serangan ini dan menggunakan tembakan biasa sebelum melakukan *snowball* lagi. Dengan demikian, kita bisa memaksimalkan *damage output* dan poin yang bisa kita dapatkan.

Selain itu, kita juga bisa melakukan perburuan untuk mencari dan menyerang lawan yang jaraknya terdekat dari *worm* kita. Dengan demikian, *worm* akan terus mencari dan bergerak ke arah musuh tersebut untuk menyerang dan membunuhnya.

c. Strategi *Survival*

Selain menyerang pada gilirannya, jika *worm* yang kita miliki dalam bahaya atau serangan *worm* musuh, padahal *game engine* sedang tidak memilih *worm* tersebut, kita bisa menggunakan perintah *select* sebagai mekanisme *self-defense worm* tersebut. Dengan perintah ini, kita bisa memilih *worm* yang awalnya tidak terpilih untuk menyerang *worm* lawan, sehingga selain kita bisa bertahan hidup, kita juga bisa memberikan perlawanan untuk *worm* tersebut. Artinya, meskipun *worm* tersebut mati, tetapi ia sudah menggunakan semua *resources* yang dimilikinya serta menghasilkan *damage* sebesar-besarnya sehingga membantu *worm* lain yang masih hidup.

Selain itu, dalam kondisi hanya satu *worm* kita yang masih hidup, kita bisa memutuskan beberapa strategi yang bisa jadi menjadi penentu kemenangan kita. Misalnya, ketika dalam situasi 1v1, kita bisa mencoba untuk mematikan lawan apabila kita memiliki *health* yang lebih banyak dibandingkan musuh. Dengan demikian, kita bisa langsung memenangkan permainan.

Selain itu, kita juga bisa mencoba untuk kabur apabila ternyata *worm* kita tersisa satu tetapi *worm* lawan masih banyak yang hidup. Dalam hal ini, tentunya *worm* lawan akan mencoba untuk mengejar dan membunuh kita. Oleh karena itu, ketika darah yang dimiliki *worm* lawan terdekat yang mengejar kita tinggal sedikit, kita bisa mematikan *worm* tersebut untuk mendapatkan poin lebih. Jika tidak, tentunya kabur merupakan satu-satunya pilihan yang harus dipilih.

d. Strategi *Team-Killing*

Ketika *worm* yang kita miliki sudah menggunakan semua *resources* yang dia miliki dan ternyata *health point* yang dimilikinya sudah rendah, kita bisa membunuh *worm* tersebut dengan cara menyerangnya secara sengaja. Dengan demikian, kita bisa mencegah lawan untuk mendapatkan poin yang berasal dari pembunuhan *worm* tersebut.

e. Strategi Pengejaran

Pada fase *endgame*, tidak sedikit *bot* yang didesain untuk kabur ketika dalam kondisi yang tidak menguntungkan. Ketika kita memerintahkan *bot* yang kita miliki untuk menembak, besar kemungkinan tembakan kita akan gagal. Oleh karena itu, salah satu strategi yang bisa digunakan dalam mengatasi hal ini adalah dengan mencoba untuk bergerak ke dalam posisi *worm* lawan. Hal ini akan mengakibatkan kedua *worm* mendapatkan *pushback damage* dan tentunya kita akan menang apabila *health worm* kita lebih besar.

Dalam permainan *Worms* ini kita bisa menggabung-gabungkan setiap strategi dan menggunakannya dalam kondisi tertentu saja yang telah kita definisikan. Meskipun demikian, setiap strategi tentunya memiliki *trade-off* nya masing-masing dengan strategi lainnya. Oleh karena itu, strategi-strategi ini harus bisa saling melengkapi satu sama lain sehingga hasil yang didapatkan bisa menjadi maksimal.

3.3 Analisis Efisiensi dan Efektivitas Strategi *Greedy*

Sebenarnya, dalam permainan *Worms* ini, kompleksitas waktu algoritma tidak terlalu penting, karena *game engine* telah didesain untuk menerima perintah dari dua pemain dan menjalankannya secara bersamaan. Meskipun demikian, efisiensi dan efektivitas dari algoritma ini perlu dianalisis sehingga menghasilkan algoritma yang lebih mangkus dan lebih baik lagi. Berikut merupakan analisis efisiensi dan efektivitas dari setiap strategi di atas (sebelum dikombinasikan).

3.3.1. Analisis Efisiensi

Untuk strategi *following*, kita juga harus mengecek kondisi *Commando Worm* dari semua *worm* yang kita miliki. Hal ini dilakukan untuk memastikan bahwa *Commando Worm* masih hidup dan bisa diikuti oleh *worm* lain yang sedang dipilih oleh *game engine*. Dengan demikian, strategi *following* ini bisa dikatakan memiliki kompleksitas O(n).

Namun, untuk strategi mencari *power up*, kita harus mengecek dan mencari posisi *cell* yang mengandung *power up* dari sekian banyak *cell* pada peta permainan. Selain itu, kita juga perlu menentukan *cell power up* yang paling dekat dengan *worm* yang sedang aktif. Oleh karena itu, kompleksitas waktu untuk strategi pergerakan yang mencari *power up* ini jauh lebih tinggi, yaitu diperkirakan mendekati $O(n^2)$.

Untuk strategi pergerakan *worms* secara umum, perlu dilakukan pemeriksaan terhadap semua *cell* yang mungkin berada di sekitar *worm* pemain. Hal ini mau tidak mau harus dilakukan supaya *worm* ini bisa mendeteksi dan menentukan ke arah dan blok mana dia bisa dan harus pergi. Selain itu, karena

ada kemungkinan bahwa blok tersebut telah ditempati oleh teman atau musuh, *worm* ini juga harus mengecek hal tersebut. Oleh karena itu, perkiraan kompleksitas waktu untuk algoritma pergerakan ini adalah $O(n^2)$, yaitu dengan melakukan perulangan atau *looping* untuk setiap *surrounding cells* di dekat *worm* kita.

Untuk strategi menyerang, kita harus selalu mengecek keberadaan musuh dalam jangkauan serangan yang kita miliki. Selain itu, kita juga harus mencari posisi musuh tersebut secara tepat supaya tembakan yang kita lakukan tidak meleset. Oleh karena itu, strategi penyerangan umumnya memang diharuskan untuk memiliki kompleksitas yang sangat tinggi, bahkan bisa jadi melebihi $O(n^2)$. Strategi ini akan menjadi lebih kompleks lagi ketika kita ingin mengecek *resources special attack* yang *worm* kita miliki sehingga kita memaksimalkannya.

Untuk strategi berburu, tentunya kita akan mencari posisi lawan yang terdekat dan kemudian melakukan mekanisme pengecekan serangan seperti yang telah dilakukan di atas. Oleh karena itu, strategi ini juga memiliki kompleksitas yang sangat tinggi, yaitu $O(n^2)$. Hal serupa juga dapat dikatakan untuk strategi pengejaran, bedanya pada strategi ini kita harus mencari posisi *worm* lawan secara presisi sehingga kita bisa memberikan *pushback damage*. Meskipun demikian, kompleksitas waktu yang dimiliki strategi ini masih berada di kisaran kompleksitas waktu $O(n^2)$.

Strategi *survival* mengharuskan kita untuk mengecek beberapa struktur data secara iteratif sampai permainan berakhir. Struktur data yang perlu dicek terkait dengan keberadaan musuh, posisi *worm* kita yang tersisa, serta kondisi dan tipe blok yang ada di sekitarnya. Selain itu, terkadang kita juga harus mengecek kondisi *health* yang dimiliki oleh semua *worm* lawan yang masih hidup. Oleh karena itu, strategi ini bisa dikatakan sebagai strategi yang memerlukan kemampuan komputasi yang paling tinggi dengan kompleksitas waktu $O(n^3)$.

3.3.2. Analisis Efektivitas

Analisis efektivitas algoritma ini kami tinjau baik saat semua alternatif strategi digabungkan atau ditinjau secara masing-masing supaya analisis yang dihasilkan bisa lebih komprehensif dan sistematis.

Di setiap kombinasi strategi yang divariasikan, strategi pergerakan dan penyerangan merupakan fundamental strategi *worms*. Strategi pergerakan dan penyerangan yang dilakukan dengan mekanisme di atas terbukti sudah cukup untuk mengalahkan *reference bot* yang disediakan Entelect. Hal ini dikarenakan

bot sudah didesain untuk menghasilkan *damage output* yang sebesar-besarnya, baik melalui utilitas *skill* maupun serangan biasa.

Namun, ketika strategi divariasikan dengan *team killing*, *bot* yang kami bangun cenderung mengalami kekalahan terhadap *reference bot*. Hal ini dikarenakan *bot* ini dibunuh begitu saja oleh *worm* lainnya. Padahal, tentunya bisa saja *worm* ini justru menjadi penentu kemenangan permainan. Oleh karena itu, strategi ini tidak lagi kami pilih untuk digunakan sebagai strategi utama.

Pada akhirnya, kami mengombinasikan keempat strategi lainnya sebagai strategi utama yang digunakan dalam membangun *bot* kami. Agoritma *greedy* yang kami terapkan cukup efektif dalam hal mencari *power up*, mencari musuh ataupun melakukan optimalisasi poin yang didapat dari *digging* dan *move* secara terus-menerus dengan meminimalisir *do nothing*.

Tentunya, *best case* dari pengujian strategi ini didapatkan ketika *worm* lawan ternyata berkumpul bersama-sama dalam jarak tembak dan *impact* yang mungkin *worm* kami berikan dengan menggunakan *special skill*. Dengan demikian, *damage* dan poin yang kami hasilkan menjadi sangat besar dan memudahkan *worm* kami dalam memenangkan pertandingan.

Namun, *worst case* yang sering kami temui adalah ketika *worm* kami belum berhasil *regroup* satu sama lain atau ketika mencari *power up*. Karena *worm* lawan juga memiliki strategi yang mirip, maka hampir dapat dipastikan bahwa *worm* ini akan bertemu dan seringkali *worm* dengan darah yang lebih sedikit seperti *Agent* dan *Technologist* dikalahkan terlebih dahulu sebelum mencapai potensi utilitas terbaiknya.

Meskipun demikian, ketika memasuki ronde akhir, *worm* kami masih memiliki kesempatan untuk menang melalui strategi *survival*. Ketika mencapai ronde 400, *worm* kami sering memenangkan permainan dengan skor yang lebih tinggi. Padahal, *health* yang dimilikinya bisa saja sangat rendah dibandingkan milik lawan. Dengan terus bergerak dan melakukan *digging*, *worm* bisa mendapatkan poin yang lebih maksimal, ketimbang melakukan *missed attack*. Selain itu, *best case* lain juga masih dapat terjadi apabila ternyata *worm* lawan terdekat yang melakukan *hunting* memiliki *health* yang tinggal sedikit dan *worm* kami berhasil membunuhnya, mengakibatkan penambahan poin yang lebih banyak.

Secara keseluruhan, kombinasi strategi ini cukup efektif dalam melakukan tugasnya masing-masing untuk mencapai tujuan akhir yang sama, yaitu mendapatkan poin sebesar-besarnya atau mematikan lawan. Meskipun demikian, strategi *survival* yang digunakan cukup riskan dalam hal kesalahan komputasi

seperti *Stack Overflow* karena kebutuhan komputasi yang tinggi seperti yang telah disebutkan sebelumnya.

3.4 Pemilihan Algoritma *Greedy*

Pada akhirnya, kami mencoba untuk menggabungkan strategi-strategi di atas. Selain itu, kami juga mencoba mengganti-ganti urutan prioritas perintah yang bisa dikombinasikan. Hasil yang paling optimal kami dapatkan dan pada akhirnya kami putuskan untuk dipakai adalah dengan menggunakan prioritas seperti urutan berikut: strategi penyerangan dengan memaksimalkan *select command*, strategi *following Commando Worm*, strategi mencari *power up*, strategi *survival* dengan urutan 1 v 1 dan 1 v Many, strategi penyerangan biasa, dan strategi *following* sebagai strategi *default*.

Algoritma greedy yang kami pilih untuk diimplementasikan adalah algoritma yang memaksimalkan poin setiap ronde melalui susunan prioritas perintah yang diberikan kepada setiap *worm*. Kami juga memaksimalkan skor yang didapat dari strategi penyerangan dengan memaksimalkan pemanfaatan utilitas yang berasal dari *special attack* yang dimiliki *worm* kami.

Selain itu, algoritma *greedy* kami desain untuk menghindari munculnya respon *do nothing* ataupun *invalid command* yang bisa muncul akibat beberapa hal. Hal tersebut misalnya adalah *target occupied* atau *no command given*. Hal ini kami lakukan sebagai upaya untuk terus mengumpulkan poin pada setiap rondenya tanpa berhenti.

Strategi yang demikian dibuat dengan memprioritaskan pergerakan sebagai strategi *default* dari *worm* kami. Kami menjadikan pergerakan sebagai strategi fundamental dalam mengumpulkan poin sebanyak-banyaknya karena strategi ini secara rata-rata memiliki *case* yang paling optimal untuk dijadikan sebagai strategi *default*.

Urutan prioritas terakhir yang kami tentukan berasal dari proses *trial* dan *error* yang kami ujikan pada *reference bot*. Hasilnya, prioritas yang disusun demikian menghasilkan *winrate* tertinggi dibandingkan variasi-variasi lainnya. Berdasarkan pengujian yang telah kami lakukan juga, *worm* yang telah kami bangun umumnya hanya kalah ketika mendapatkan *worst case* yang telah disebutkan di atas, baik terbunuhnya *worm* dengan *special attack* ataupun gagalnya usaha *worm* untuk kabur, serta diakibatkan oleh kemungkinan terjadinya kesalahan komputasi yang telah disebutkan sebelumnya.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma *Greedy*

Implementasi algoritma greedy pada program kami terdapat pada bagian file bot.java. Di dalam file tersebut terdapat sebuah fungsi pemanggilan yang utama yaitu run.

4.1.1. Public Command run

```
function run() → Command
{Fungsi utama untuk menentukan point maksimal berdasarkan strategi yang telah
dipersiapkan }

Kamus Lokal
enemyWorm, enemyBananaSnowball, enemy, enemySpecial, aliveenemy : Worm
PowerUp : Position
toPowerUp : Direction

Algoritma

{Get enemy worms in range}
enemyWorm ← getClosestEnemyInWeaponRange()
enemyBananaSnowball ← getClosestEnemyInSpecialAttackRange()

{Use All Select Command Chances to Attack or Self-Defense}
if(selectionAvailable) then
    traversal aliveWorms[0..2]
    if(health > 0) then
        enemy ← getClosestEnemyInWeaponRange()
        enemySpecial ← getClosestEnemyInSpecialAttackRange()
        if(enemy not null or enemySpecial not null) then
            UseSelectCommandToAttack()

{Get and Follow Commando Worm if it is still alive}
if(commandoWorm not null) then
    followCommando()

{Check if power up is still available and not taken yet}
if(powerUpsStillAvailable) then
```

```
        PowerUp ← getPowerUpCell()
        toPowerUp ← resolveDirection(currentWormPosition, PowerUp)
        if(enemyWorm not null or enemyBananaSnowball not null) then
            → AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)
        → digAndMove(currentWorm, toPowerUp)

    {Check if 1v1 condition or not}
    if(onevone) then
        if(enemyBananaSnowball not null and countBanana > 0) then
            → AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)
        else if(enemyBananaSnowball not null and countSnowball > 0) then
            → AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)

        iterate opponentWorms[0..2]
            if(health > 0) then
                aliveenemy ← opponentWorms[i]
        if(enemyWormHealth > myWormHealth) then
            → lonewolf()

    {Check if 1v more than 1 enemies}
    if(Alone) then
        if(enemyBananaSnowball not null and countBanana > 0) then
            AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)
        else if(enemyBananaSnowball not null and countSnowball > 0) then
            AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)

        aliveenemy ← getClosestEnemies()
        if(enemyWormHealth > myWormHealth) then
            → lonewolf()
        else if(enemyWorm not null and enemyBananaSnowball not null and
        SpecialAttacksAvailable) then
            → AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)
        → lonewolf()

    {Available to attack}
    if(enemyWorm not null and enemyBananaSnowball not null and
    SpecialAttacksAvailable) then
        → AttackCommand(currentWorm, enemyWorm, enemyBananaSnowball)

    → following()
```

4.1.2. Public Command AttackCommand

```
function AttackCommand(current : MyWorm , enemyWorm : Worm,
enemyBananaSnowball : worm, id : int) → Command
{ Fungsi untuk memaksimalkan attack strategy precondition untuk memanggil fungsi
ini adalah terdapat musuh di jarak tembak }

Kamus Lokal
distCWtoEBS : float
Direction : Direction
foundFrozen : boolean

Algoritma
distCWtoEBS ← 0
Direction ← null

if (enemyWorm ≠ null) then
     distCWtoEBS ← euclideanDistance(current.position.x, current.position.y,
enemyWorm.position.x, enemyWorm.position.y)
    Direction ← resolveDirection(current.position, enemyWorm.position)

if  (enemyBananaSnowball ≠ null and current.id = 2 and current.bananas.count > 0)
then
            if (id≠0) then
           →  SelectCommand(id, null, enemyBananaSnowball.position.x,
enemyBananaSnowball.position.y,
                  false, true, false, false, false);
        else
           →  BananaBomb(enemyBananaSnowball.position.x,
enemyBananaSnowball.position.y);
else if (enemyBananaSnowball ≠ null and current.id = 3 and current.snowballs.count
> 0) then
         {Maximizing Utility}
         {Worm = Technologist}
         {Can Use Snowball}

         {Detect if any enemy's worm is frozen}
         foundFrozen ←  false;
         for (enemies in opponent.worms) do
           if (enemies.health>0 and enemies.notFrozen ≠ 0)
              foundFrozen ← true
```

```
        if (enemyWorm ≠ null and foundFrozen)
            {Maximizing Freeze Duration between each snowball used}
            {One or more enemy's worm is frozen}
            {Do ordinary shoot before using another snowball}
            if (id≠0)
                → SelectCommand(id, direction, -1, -1, false, false, false,
                    false, true)
            else
                return new ShootCommand(direction);
        else if (foundFrozen)
            return lonewolf()
        else
            {No enemy frozen -> If missed snowball or before finding any enemies}
            { Use snowball to freeze enemies}
            if (id≠0)
                → SelectCommand(id, null, enemyBananaSnowball.position.x,
enemyBananaSnowball.position.y,
                    true, false, false, false, false)
            else
                → Snowball(enemyBananaSnowball.position.x,
enemyBananaSnowball.position.y);

else if (enemyWorm ≠ null and distCWtoEBS ≤ current.weapon.range) then
        if (id≠0) then
            → SelectCommand(id, direction, -1, -1, false, false, false, false, true);
        else
            → ShootCommand(direction);

→ DoNothingCommand
```

4.1.3. Public Command lonewolf

```
function lonewolf() → Command
{ Fungsi ini strategi untuk kabur dari musuh hingga bertahan sampai titik darah
penghabisan }

Kamus Lokal
enemyLines, CellDirection : Direction

Algoritma
{ mendapatkan musuh yang berada di jangkauan tembak current worm }
```

```
enemyLines ← resolveDirection(currentWorm.position,
getClosestEnemies(currentWorm).position)
{ mendapatkan arah dari cell yang diacak }
CellDirection ← getRandomMap()

{ apabila cell berada di area tembak target maka random ulang hingga ketemu area
yg bukan area tembak target }
While (CellDirection = enemyLines or (((CellDirection.x*-1)=enemyLines.x) and
((CellDirection.y*-1)=enemyLines.y))) do
        CellDirection ← getRandomMap()

→ digAndMove(currentWorm, CellDirection)
```

4.1.4. Public Command following

```
function following() → Command
{ Fungsi ini bertujuan untuk memeriksa apakah Worm bertipe Commando masih
hidup apabila hidup maka semua Worm akan menuju ke Commando jika Commando
sudah tiada maka tiap worm akan menuju worm lainnya }

Kamus Lokal
commando, enemies, friends : Worm
commandoDirection, friendDirection, huntDirection : Direction

Algoritma
{ menemukan Commando, jika tidak maka nilai menjadi null }
commando ← findCommando()
{ menemukan enemy yang terdekat }
enemies ← getClosestEnemies(currentWorm)
if (commando ≠ null and currentWorm.id ≠ 1) then
        commandoDirection ← resolveDirection(currentWorm.position,
commando.position)
        → digAndMove(currentWorm, commandoDirection)
else if  (commando = null and currentWorm.id ≠ 1) then
      friends ← getClosestFriend(currentWorm)
      friendDirection ← resolveDirection(currentWorm.position, friends.position)
      → digAndMove(currentWorm, friendDirection)
else if (Alone()) then
     → lonewolf()

huntDirection ← resolveDirection(currentWorm.position, enemies.position)
→ digAndMove(currentWorm, huntDirection)
```

4.1.5. Public Command digAndMove

```
function digAndMove(currentWorm : Worm , direction : Dir) → Command
{ Fungsi ini bertujuan untuk memaksimalkan score saat ingin bergerak di map }

Kamus Lokal
newX, newY : int
Block, blockNotDS, block1 : Cell
AllSurroundingBlocks[] : List of Cell
ran : Position
random : Direction

Algoritma
newX ←  currentWorm.position.x + dir.x;
newY ←  currentWorm.position.y + dir.y;

AllSurroundingBlocks[] ← getSurroundingCells(currentWorm.position.x,
currentWorm.position.y)
cellIdx ← random.nextInt(AllSurroundingBlocks.size())

Block ← AllSurroundingBlocks[cellIdx]

for Blocks in AllSurroundingBlocks do
       If (Blocks.x=newX and Blocks.y = newY) then
            block ← Blocks

if (block.type=CellType.LAVA) then
     → reverseAtLava(currentWorm, dir)
else if (block.type = CellType.DIRT) then
     → DigCommand(block.x, block.y)
else if (block.type=CellType.AIR) then
     → MoveCommand(block.x, block.y)
else if (block.type =CellType.DEEP_SPACE) then
     blockNotDS ← AllSurroundingBlocks[cellIdx]
     if (blockNotDS.type ≠ CellType.DEEP_SPACE) then
            if(blockNotDS.type ≠ CellType.DIRT or blockNotDS.type =
CellType.AIR) then
              →  MoveCommand(blockNotDS.x, blockNotDS.y)
           else if(blockNotDS.type = CellType.DIRT) then
              →  DigCommand(blockNotDS.x, blockNotDS.y)
block1 ←  AllSurroundingBlocks[cellIdx]
```

```
ran ← new Position(block1.x, block1.y)
random ← resolveDirection(currentWorm.position, ran)
→ digAndMove(currentWorm, random)
```

4.1.6 Private Boolean Alone

```
function Alone() → Boolean
{ Fungsi ini bertujuan untuk mengecek apakah worm kita tinggal satu }

Kamus Lokal
count: int

Algoritma
count ← 0
iterate myWorms[0..2]
    if(health > 0) then
        count ← count + 1
→ count = 1
```

4.1.7 Private Boolean onevone

```
function onevone() → Boolean
{ Fungsi ini bertujuan untuk mengecek apakah worm kita sedang 1v1 dengan worm
lawan}

Kamus Lokal
count: int

Algoritma
count ← 0
iterate enemyWorms[0..2]
    if(health > 0) then
        count ← count + 1
→ count = 1 and Alone()
```

4.1.8 Private Worm findCommando

```
function findCommando() → Worm
{ Fungsi ini bertujuan untuk mengambil data Commando }

Kamus Lokal
```

```
count: int

Algoritma
iterate myWorms[0..2]
    if(Id = IdCommando and CommandoHealth > 0) then
        → myWorms[i]
```

4.1.9 Private Worm getClosestFriend dan getClosestEnemy

```
function getClosest() → Worm
{ Fungsi ini bertujuan untuk mengambil data worm lawan ataupun musuh yang
terdekat dengan currentWorm}

Kamus Lokal
range, minRange: float
worm: Worm

Algoritma
worm ← null
iterate myWorms[0..2] {atau iterate opponentWorms[0...2]
    range ← euclideanDistance(currentWorm, target)
    if(range < minRange) then
        minRange ← range
        worm ← myWorms[i]
→ worm
```

4.1.10 Private Position getPowerUpCell

```
function getPowerUpCell() → Position
{ Fungsi ini bertujuan untuk mengambil data cell power up}

Kamus Lokal
MapList: List

Algoritma
Cells ← filterPowerUp(MapList)
→ getClosest() {Mengambil cell terdekat}
```

4.1.11 Private Command ReverseAtLava

```
function ReverseAtLava() → Command
```

{ Fungsi ini bertujuan untuk mengarahkan *worm* ke arah sebaliknya jika bertemu dengan *lava*}

Kamus Lokal

Algoritma
→ digAndMoveToOppositeofLava()

4.2 Struktur Data Program *Worms*

Pada permainan *"Worms"* ini, struktur data yang digunakan berbasis pada *class*/kelas. Meskipun sudah terdapat beberapa kelas yang diberikan oleh *template* permainan dari *Entelect*, dalam proses pembuatan kami menambahkan beberapa kelas untuk melengkapi beberapa implementasi seperti skill dari *worm*, dll. Algoritma greedy yang kami kembangkan secara umum membagi kelas-kelas tersebut menjadi lima bagian, yaitu : *Command, Entities, Enums, Bot,* dan *Main*.

a. Bagian Command

Kelas-kelas pada bagian ini adalah kelas yang berguna untuk menyimpan aksi gerakan atau kemampuan yang dapat digunakan oleh *worm*.

i. BananaBomb

Kelas untuk menghasilkan *new* Command yaitu menggunakan kemampuan melempar bom pisang ke posisi (x,y).

ii. DigCommand

Kelas untuk menghasilkan *new* Command yang mengakibatkan *worm* untuk menghancurkan block (x,y) di sekitar yang bertipe 'DIRT'.

iii. DoNothingCommand

Kelas untuk menghasilkan *new* Command yang mengakibatkan *worm* untuk tidak melakukan apa-apa pada ronde saat kelas tersebut dipanggil.

iv. MoveCommand

Kelas untuk menghasilkan *new* Command yang mengakibatkan *worm* untuk berpindah tempat ke block (x,y) yang kosong atau bertipe 'AIR'.

v. SelectCommand

Kelas untuk menghasilkan *new* Command yang memilih *worm* yang tidak sedang dipilih oleh *game engine*.

vi. ShootCommand

Kelas untuk menghasilkan *new* Command yang mengakibatkan *worm* menembak terhadap suatu arah yang telah ditentukan sebelumnya.

vii. Snowball

Kelas untuk menghasilkan *new* Command yaitu menggunakan kemampuan melempar bola salju ke posisi (x,y).

b. Bagian Entities

Kelas-kelas pada bagian ini berguna untuk menyimpan data entity dari permainan.

i. Bananas

Kelas untuk menyimpan atribut-atribut yang diperlukan oleh *worm* saat menggunakan kemampuan.

ii. Cell

Kelas untuk menyimpan data peta berupa koordinat (x,y), tipe sel/blok, dan *powerUp* yang ada di salah satu sel pada peta.

iii. GameState

Kelas untuk menyimpan data-data permainan seperti data-data *worms* pemain dan musuh yang masih hidup, ronde maksimal, *worm* yang sedang bergerak, dsb.

iv. MyPlayer

Kelas untuk menyimpan data-data dari pemain seperti *worms* yang tersisa, *healthpoint*, dan skor dari pemain.

v. MyWorms

Kelas untuk menyimpan data senjata dari *worms* pemain.

vi. Opponent

Kelas untuk menyimpan data-data dari musuh seperti *worms* musuh yang tersisa dan skor musuh.

vii. Position

Kelas untuk membentuk atribut koordinat x dan y.

viii. PowerUp

Kelas untuk menyimpan atribut seperti tipe *powerUp* dan *value*-nya (total HP yang didapat saat mendapat *powerUp*).

ix. Snowballs

Kelas untuk menyimpan data-data dari kemampuan *snowballs* seperti tersisa amunisi berapa, durasi efek beku, jangkauan efek beku, dan jarak tembak.

x. Weapon

Kelas untuk menyimpan data dari kemampuan menembak biasa *worms* seperti *damage* dan jarak tembaknya.

xi. Worm

Kelas untuk menyimpan data-data *worms* seperti *id, health*, posisi *worms*, dll.

c. Bagian Enums

Bagian ini berisi kelas-kelas yang menyimpan data-data seperti tipe cell/blok pada peta, arah-arah yang terdapat pada permainan, dan *powerUp* yang tersedia pada permainan.

i. CellType

Kelas untuk menyimpan enumerasi tipe-tipe sel yang terdapat pada permainan antara lain ‘AIR’, ‘DIRT, ‘DEEP_SPACE’, dan ‘LAVA’.

ii. Direction

Kelas untuk menyimpan enumerasi arah-arah yang terdapat pada permainan yaitu *N, NE, E, SE, S, SW, W, NW*.

iii. PowerUpType

Kelas untuk menyimpan enumerasi tipe-tipe dari *powerUp* yang ada pada permainan (untuk saat ini hanya *Health Pack*).

d. Kelas Bot

Kelas ini berisi pengembangan dari implementasi algoritma Greedy yang kami rancang untuk memenangkan permainan “Worms”, memanfaatkan seluruh

kelas-kelas sebelumnya. Method-method pada kelas ini rata-rata menggunakan struktur *array* dan *primitive type variable* dalam mengolah input yang diterimanya.

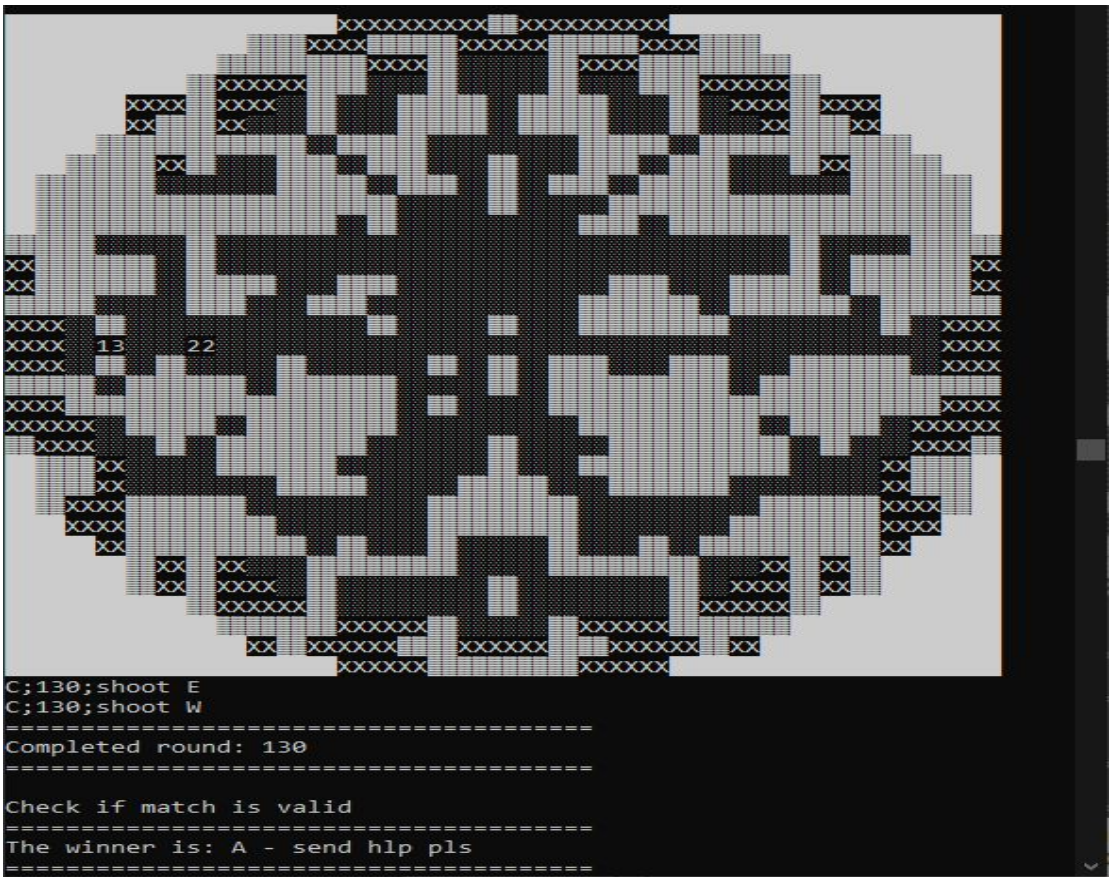
e. Kelas Main

Kelas untuk memanggil kelas *Bot* dan melaksanakan program dengan menjalankan seluruh algoritma dan *command-command* yang sudah dibentuk untuk memulai permainan.

4.3 Analisis Pengujian Algoritma *Greedy*

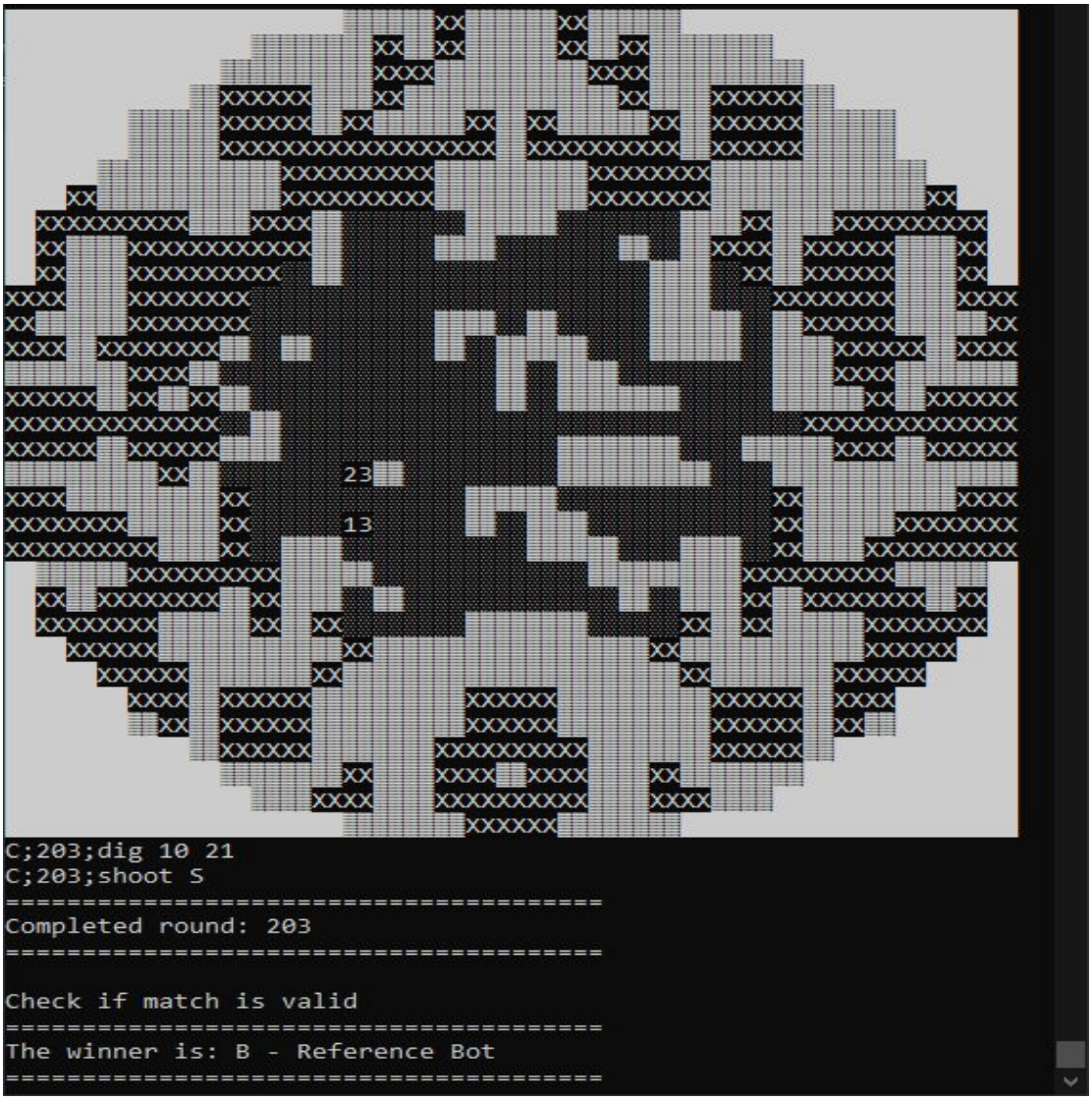
Percobaan untuk mengambil data uji ini dilakukan cukup dengan menjalankan `run.bat` pada folder starter-pack (src jika sudah diganti) yang berisi *bot* yang telah dikembangkan. Pada kesempatan kali ini, kami melakukan tiga kali pengujian terhadap *bot* yang telah kami rancang.

a. Pengujian I



Pada pengujian pertama, *bot* yang kami rancang berhasil mengalahkan *reference bot* yang disediakan oleh Entelect hanya dalam 130 ronde dimana *worm* terakhir dari pemain adalah Technologist dan *worm* terakhir dari musuh adalah Agent. Hasil akhir ini setelah dilihat pada visualizer, dikarenakan pada saat permulaan permainan, *worm* musuh terlihat berkumpul bersama-sama ditengah dan kebetulan *worm* player yang bertipe Technologist dan Agent datang belakangan sehingga skill-skill seperti Snowball dan BananaBomb dapat dimanfaatkan secara maksimal dan mengenai hampir semua *worm* musuh dan *hitpoints* yang diakibatkan juga besar sehingga musuh dapat dengan mudah dibunuh.

b. Pengujian II



Namun, pada pengujian kedua, *bot* yang kami rancang mengalami kekalahan terhadap *reference bot* dalam 203 ronde, dimana kedua pihak sama-sama tersisa Technologist. Dapat dilihat disini bahwa meskipun *worm* kami menggunakan strategi *lonewolf* pada kasus satu lawan satu, *worm* kami tetap dapat kena tembak dan akhirnya kalah. Setelah dilihat pada visualizer, benar bahwa pada saat satu lawan satu, *worm* player tidak berhasil menghindari semua tembakan musuh dengan strategi *lonewolf* sehingga ia mati dan player kehabisan *worm*. Kasus seperti ini sebenarnya adalah kasus terburuk dimana pemain dipaksa kabur-kaburan sambil menghindari tembakan yang diakibatkan dari sudah matinya *worm-worm* lain dari pemain di awal-awal permainan.

c. Pengujian III

Terakhir, pada pengujian ketiga kami menemukan kasus yang unik dimana permainan dapat bertahan hingga ronde 400 dan kedua pihak sama-sama tersisa Technologist dan pemain berhasil menghindar secara terus-menerus menggunakan strategi *lonewolf* dan pada akhirnya skor akhir yang dicapai pemain jauh lebih besar daripada milik *reference bot* sehingga pemain dinyatakan menang. Pada visualizer, pada akhir-akhir ronde 200-400 terlihat bahwa strategi *lonewolf* dimana *worm* pemain kabur-kaburan sambil menghindari tembakan mulai dijalankan, dan ternyata hingga ronde 400 pemain masih belum mati sehingga skor akhir yang dihasilkan pastinya lebih besar daripada skor akhir musuh.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Kami berhasil mengimplementasikan permainan "Worms" menggunakan algoritma *greedy* yang bisa mencapai tujuan objektif dari permainan ini, yaitu memenangkan permainan baik dengan cara mengeliminasi seluruh *worms* lawan menggunakan senjata dan *special skill* yang telah disediakan atau mendapatkan skor akhir yang lebih besar. Selain itu, kami juga menggunakan beberapa pendekatan/strategi *greedy* dalam prosesnya untuk memenangkan permainan seperti *command* untuk berjalan/mengeksplor peta, menyerang musuh dengan senjata terbaik yang dimiliki, ataupun bertahan hidup dan mengumpulkan poin sebanyak-banyaknya.

Namun, pada tugas besar 1 Strategi Algoritma kali ini kami juga menyadari bahwa algoritma *greedy* yang dibuat tidak selalu membuahkan hasil yang paling optimal pada implementasi permainan "Worms" ini. Hal ini dapat terlihat ketika kami tidak selalu menang melawan *bot* yang disediakan oleh Entelect. Hal ini dimungkinkan terjadi karena perubahan kondisi *gameState* dan *map* yang acak pada setiap kali pengujian. Selain itu, algoritma juga melakukan penyeleksian solusi secara bertahap per rondenya sehingga kurang meninjau semua kemungkinan yang ada secara keseluruhan, sesuai dengan *nature* dari algoritma jenis ini.

5.2. Saran

Pengaplikasian algoritma greedy pada permainan "Worms" ini masih sangat jauh untuk dapat dibilang sempurna sehingga saran kami pada tugas besar kali ini adalah agar pengembangan permainan "Worms" ini mungkin dapat dikembangkan kembali pada masa yang mendatang. Program dapat dikembangkan lebih jauh menggunakan pendekatan-pendekatan *greedy* yang lebih efisien dan efektif dari strategi yang telah kami implementasikan. Untuk waktu jangka panjangnya, prinsip

Intelegensi Buatan (Artificial Intelligence) mungkin dapat diterapkan pada permainan ini sehingga *bot* yang dimainkan dapat "belajar" setiap pengujiannya dan beradaptasi pada pengujian selanjutnya.

DAFTAR PUSTAKA

Entelect Forum. Entelect Worms 2019 Challenge. Diakses pada 17 Februari 2020 pukul 18.21 WIB melalui https://forum.entelect.co.za/.

Munir, R. (2020). Algoritma Greedy Bagian 1[PDF]. Institut Teknologi Bandung. Diakses pada 18 Februari 2020 pukul 11.21 WIB melalui *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf*.

Munir, R. (2020). Algoritma Greedy Bagian 2[PDF]. Institut Teknologi Bandung. Diakses pada 18 Februari 2020 pukul 11.33 WIB melalui *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf*.

Munir, R. (2020). Algoritma Greedy Bagian 3[PDF]. Institut Teknologi Bandung. Diakses pada 18 Februari 2020 pukul 11.47 WIB melalui *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag3.pdf*.