TUGAS BESAR 1

PEMANFAATAN ALGORITMA GREEDY DALAM APLIKASI PERMAINAN WORM

IF 2211

STRATEGI ALGORITMA

SEMESTER 2

Oleh

Kelompok 61 Ca-ching

Dwianditya Hanif Raharjanto	**13519046**
Michael Owen	**13519055**
M. Abdi Haryadi. H	**13519156**



PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

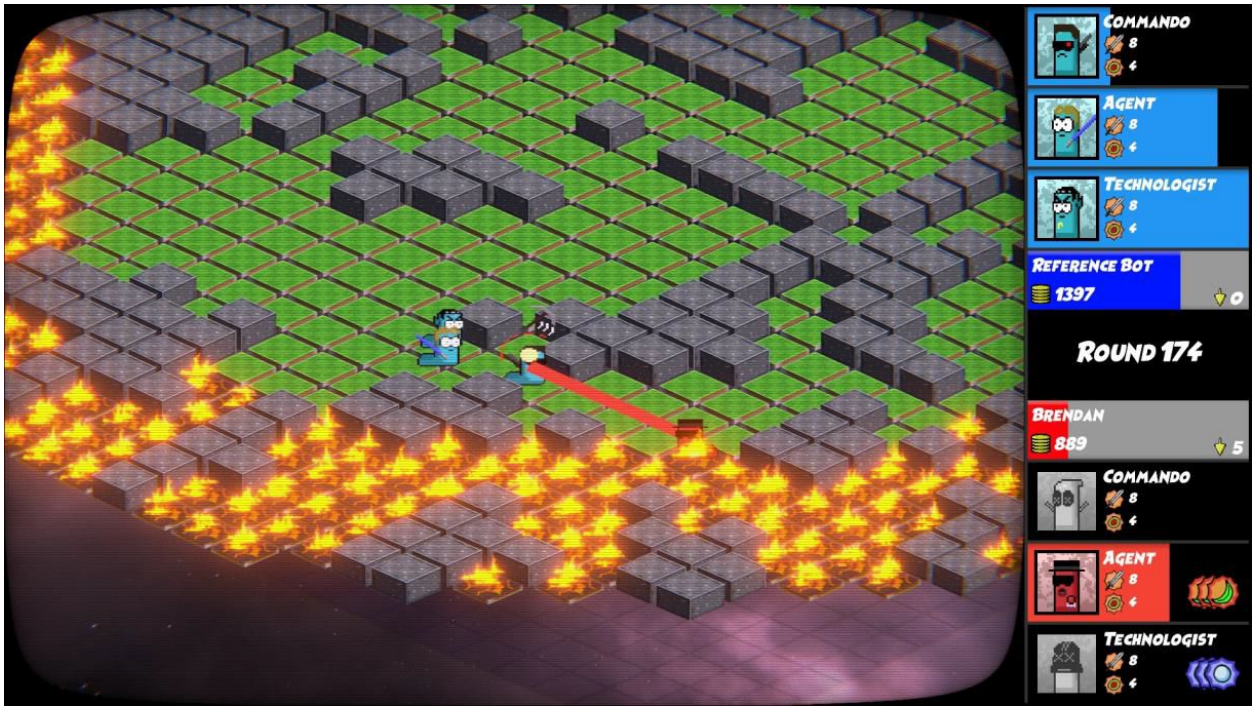
INSTITUT TEKNOLOGI BANDUNG

2020/2021

Daftar Isi

Bab 1 Deskripsi Tugas

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



Gambar 1 Contoh tampilan permainan Worms

Pada tugas besar pertama Strategi Algoritma ini, mahasiswa menggunakan sebuah *game engine* untuk mengimplementasikan permainan Worms. *Game engine* dapat diperoleh pada laman https://github.com/EntelectChallenge/2019-Worms. Tugas mahasiswa adalah mengimplementasikan seorang "pemain" Worms dengan menggunakan strategi *greedy* untuk memenangkan permainan. Untuk mengimplementasikannya, mahasiswa disarankan melanjutkan program yang terdapat pada *starter bot* pada laman https://github.com/EntelectChallenge/2019-Worms/releases/tag/2019.3.2 , berkas *starter pack*.

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 33 x 33 *cell*. Terdapat 4 tipe *cell*, yaitu *air*, *dirt*, *deep space*, dan *lava* yang masing-masing memiliki karakteristik berbeda. *Cell* dapat memuat *power-up* yang bisa diambil oleh *worm* yang berada pada *cell* tersebut.
2) Pada awal permainan, setiap pemain akan memiliki tiga pasukan *worm* 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 *worm* target yang terkena serangan, sehingga mengurangi *health point*-nya. Jika *health point* suatu *worm* sudah habis, maka *worm* tersebut dinyatakan tereliminasi dari permainan.
6) Permainan akan berakhir ketika salah satu pemain berhasil mengeliminasi seluruh pasukan *worm* lawan atau permainan sudah mencapai jumlah *round* maksimum (400 *round*).

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.

Bab 2 Landasan Teori

2.1 Algoritma *Greedy*

Pada tugas besar Strategi Algoritma yang pertama ini kami menggunakan strategi algoritma *greedy* yang secara umum dikenal sebagai metode sederhana dalam memecahkan persoalan optimasi (mencari solusi optimal). Hanya ada dua macam persoalan optimasi yaitu maksimasi dan minimasi.

Definisi dari algoritma *greedy* sendiri adalah algoritma yang memecahkan persoalan secara langkah per langkah. Pada setiap langkah kita perlu mengevaluasi setiap pilihan yang ada dan mengambil pilihan terbaik karena tidak bisa mundur ke langkah sebelumnya. Jadi setiap langkah kita mengambil solusi terbaik yang dapat diperoleh pada saat itu tanpa memikirkan konsekuensi ke depan (menggunakan prinsip "*take what you can get now"*) dan berharap bahwa pemilihan optimum lokal akan membuahkan hasil akhir berupa optimum global.

Dalam algoritma *greedy* terdapat enam elemen yang harus dipenuhi yaitu :

1) Himpunan kandidat : Berisi kandidat yang akan dipilih pada setiap langkah.
2) Himpunan solusi : Berisi kandidat yang sudah dipilih dari himpunan kandidat.
3) Fungsi solusi : Menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi.
4) Fungsi seleksi : Memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.
5) Fungsi kelayakan : Memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak).
6) Fungsi obyektif : Memaksimumkan atau meminimumkan.

Algoritma *greedy* ini tidak selalu menghasilkan solusi optimum, tetapi bisa menjadi solusi sub-optimum karena algoritma *greedy* ini tidak beroperasi secara menyeluruh terhadap semua kemungkinan solusi yang ada seperti algoritma *brute force* dan terdapat beberapa fungsi seleksi yang berbeda. Sehingga kita harus memilih fungsi yang sangat tepat jika ingin algoritma menghasilkan solusi optimal.

Jadi algoritma *greedy* ini cocok kita pakai ketika solusi terbaik mutlak tidak terlalu diperlukan, karena algoritma *greedy* menghasilkan solusi hampiran. Kemudian waktu yang dibutuhkan untuk memproses algoritma ini sangat lebih cepat daripada algoritma *brute force*. Namun, ketika solusi optimal dihasilkan, maka harus dapat dibuktikan secara matematis. Lebih mudah memperlihatkan algoritma *greedy* tidak selalu optimal dengan menunjukkan *counterexample* (dengan menunjukkan solusi yang diperoleh tidak optimal).

2.2 Pemanfaatan *Game Engine*

Untuk memanfaatkan *game engine*, hal yang perlu disiapkan pertama kali adalah memasang Java SE Development Kit (JDK) 8 dan IntelliJ IDEA. Dari laman https://github.com/EntelectChallenge/2019-Worms/releases/tag/2019.3.2 , unduh dan ekstrak berkas `starter-pack.zip`.

Pemain yang digunakan berupa *bot* yang diprogram sesuai dengan yang diinginkan pemrogram. Berkas yang akan diprogram berada pada direktori `starter-pack\starter-bots\java\src\main\java\za\co\entelect\challenge\Bot.java`. Setelah diprogram sesuai yang diinginkan, *bot* akan difinalisasi dengan mengepaknya. Hal itu dilakukan dengan mengakses tab "Add Maven Project" pada IntelliJ IDEA dengan berkas pom.xml pada direktori `starter-pack\starter-bots\java\`, lalu akses `java-sample-bot`, lalu `Lifecycle`, lalu `install`. Hasilnya berupa berkas yang berada pada folder `target` dengan nama `java-sample-bot-jar-with-dependencies.jar`.

Setelah mendapatkan hasil pembuatan *bot*, atur direktori *bot* yang digunakan dengan mengganti nilai `player-a` atau `player-b` pada `starter-pack\game-runner-config.json`. Setelah itu, jalankan `run.bat` untuk sistem operasi Windows, atau gunakan perintah `make run` untuk Linux. Pada konsol terlihat status jenis *cell* dan posisi setiap *worm* untuk setiap *round*. Pada akhir eksekusi, tercipta folder baru pada `starter-pack\match-logs\`. Unduh berkas pada laman https://github.com/dlweatherhead/entelect-challenge-2019-visualiser/releases/tag/v1.0f1 untuk memvisualisasikannya dengan lebih baik. Ekstrak berkas tersebut, dan masukkan folder dari `starter-pack\match-logs\` yang sebelumnya tercipta ke `EC2019 Final v1.0f1\Matches\`. Kemudian, jalankan `start-visualizer.bat` pada `EC2019 Final v1.0f1` dan klik nama *match* yang ingin ditampilkan.

(Informasi ini diperoleh dari `starter-pack\starter-bots\java\README.md`)

Bab 3 Pemanfaatan Strategi *Greedy*

3.1 Pemetaan Persoalan Worms Menjadi Elemen-Elemen *Greedy*

Secara umum, ada enam elemen dalam algoritma *greedy*: himpunan kandidat, himpunan solusi, fungsi solusi, fungsi seleksi, fungsi kelayakan, dan fungsi objektif (Munir, 2021). Keenam elemen ini dapat dikaitkan dengan persoalan Worms. Himpunan kandidat dalam operasi Worms adalah himpunan bagian dari seluruh aksi yang mungkin dalam Worms. Ukuran dari himpunan ini bergantung pada strategi *greedy* yang digunakan. Dalam himpunan tersebut dipilih hanya satu solusi sehingga himpunan solusi dalam hal ini dapat dikatakan "solusi"—tanpa himpunan. Hal itu terjadi karena untuk setiap *round*, perintah yang diterima hanyalah satu. Mengingat ukurannya hanya satu, fungsi solusi tidak diperlukan lagi karena sudah pasti hanya satu solusi.

Untuk memilih solusi dalam himpunan kandidat, digunakan fungsi seleksi yang implementasinya bergantung pada strategi *greedy* yang digunakan. Akan tetapi, tidak semua solusi dapat diterima meskipun perintahnya ada, misalnya *worm* berjalan ke *deep space*. Oleh karena itu, fungsi kelayakan digunakan agar solusi yang menyimpang dengan peraturan Worms atau hal-hal lain yang juga bergantung pada strategi *greedy* dieliminasi. Hal yang membuat solusinya optimal (secara lokal) adalah fungsi objektif. Fungsi ini bergantung pada strategi *greedy* juga dan bisa saja sudah dicakup dalam fungsi seleksi.

3.2 Alternatif Solusi *Greedy*

Berikut adalah beberapa alternatif solusi *greedy* yang dapat digunakan dalam permainan Worms:

1) Mode ofensif atau menyerang tanpa *defend* dengan memberikan damage sebesar mungkin pada lawan
2) Mendekati lawan tanpa melakukan penyerangan
3) Defensif pada posisi tertentu yang telah ditetapkan di peta
4) Defensif dengan cara menjauh dari lawan sejauh mungkin dan mencoba menghindari serangan lawan tanpa melakukan penyerangan.
5) Mode ofensif dengan memfokuskan serangan pada satu lawan terlebih dahulu sampai lawan kehabisan darah.
6) Menggunakan umpan agar cacing-cacing yang lain bisa menyerang secara masif.

3.3 Analisis Efisiensi

Berikut adalah analisis efisiensi untuk setiap alternatif solusi yang dipaparkan sebelumnya:

1) Secara umum, strategi ofensif cukup efisien untuk diimplementasikan karena kode utama yang diperlukan hanyalah fungsi untuk mengejar dan menyerang musuh dengan damage tertinggi.

2) Untuk strategi mendekati lawan efisiensi cukup baik karena hanya perlu mencari posisi musuh lalu bergerak ke arah tersebut.
3) Secara umum, mode defensif pada posisi tertentu cukup efisien karena hanya menginstruksikan cacing untuk bergerak ke koordinat yang telah ditentukan lalu tinggal melakukan spam serangan dari titik tersebut.
4) Strategi defensif dengan cara menjauh dari musuh sejauh mungkin dinilai sangat efisien karena kita hanya perlu mencari teman yang paling dekat dengan musuh sebagai acuan untuk bergerak kemana. Selain itu, untuk menghindar serangan musuhnya, kita menjauh dengan bergerak diagonal karena jangkauan musuh ketika kita bergerak diagonal persentase terkena serangan musuhnya lebih kecil daripada bergerak menjauh sejajar dari musuh.
5) Strategi memfokuskan serangan pada satu cacing lawan cukup efisien karena instruksi yang diberikan adalah mendekati cacing lawan yang terdekat hingga masuk jangkauan serangan dan menyerangnya tanpa mengubah target sampai cacing lawan kehabisan darah.
6) Strategi menggunakan umpan untuk menyerang dinilai sangat tidak efisien karena sangatlah kompleks. Pertama, kita harus menggerakkan cacing yang mau kita jadikan umpan tetapi posisi cacing sisanya tidak boleh terlalu jauh dari cacing umpan. Kemudian, kita juga harus memperhatikan jarak aman antara cacing umpan dengan cacing sisanya.

3.4 Analisis Efektivitas

Berikut adalah analisis efektivitas untuk setiap alternatif solusi yang dipaparkan sebelumnya:

1) Efektivitas dari strategi ofensif cukup baik jika cacing musuh bergerak secara bersamaan atau jika musuh menggunakan strategi menghindar. Namun, karena berfokus pada *damage* yang besar, serangan dapat melukai tim sendiri. Selain itu, jika cacing berada di dalam lava, tetapi musuh terus melakukan serangan balik, cacing bisa mati karena *health*-nya terus berkurang karena serangan musuh ditambah *damage* lava.
2) Strategi mendekati lawan kurang efektif karena sulit untuk menang karena mengandalkan poin saja. Selain itu, jika berada di area musuh, musuh dapat melakukan serangan selagi cacing mendekat.
3) Mode defensif pada posisi tertentu cukup efektif dalam meraih kemenangan karena cacing tim dapat menyerang cacing lawan tanpa perlu khawatir pergerakan lava. Selain itu, cacing tim yang dapat terkena serangan area hanya satu cacing saja karena posisinya telah direncanakan sedemikian agar tidak saling berada pada jangkauan serangan area.
4) Strategi defensif menghindari lawan sejauh mungkin ini kurang efektif dalam meraih kemenangan karena tidak ada perlawanan dari strategi ini. Selain itu, kemungkinan akan mengarah ke lava karena yang kita instruksikan hanyalah menghindari lawan sejauh mungkin. Namun, ada kemungkinan untuk memenangkan *game* ketika cacing lawan menyerang temannya sendiri karena cacing kita berhasil menghindar dari serangan lawan.

5) Strategi memfokuskan serangan pada satu target sampai kehabisan darah cukup efektif dalam memenangkan game ini. Seperti halnya ketika kita bergotong-royong dalam menyelesaikan suatu masalah akan cepat selesai, sama halnya seperti strategi ini akan lebih cepat membunuh lawan. Namun, strategi ini memiliki kelemahan karena dengan strategi ini, kemungkinan besar cacing tim kita berkumpul pada tempat yang berdekatan. Akibatnya, ketika lawan menggunakan *special attack*-nya, sudah pasti cacing tim kita terkena dampaknya semua sehingga tim kita lebih cepat kehabisan darahnya daripada lawan.
6) Strategi menggunakan umpan untuk menyerang ini tidak efektif dalam memenangkan pertandingan karena terlalu banyak efek sampingnya daripada keuntungannya. Ketika umpan ternyata terlalu jauh dari cacing sisanya, umpan tersebut akan menjadi sia-sia. Kemungkinan terbaiknya adalah ketika jarak antara umpan dan cacing sisanya ideal dan yang terperangkap umpan adalah semua cacing lawan.

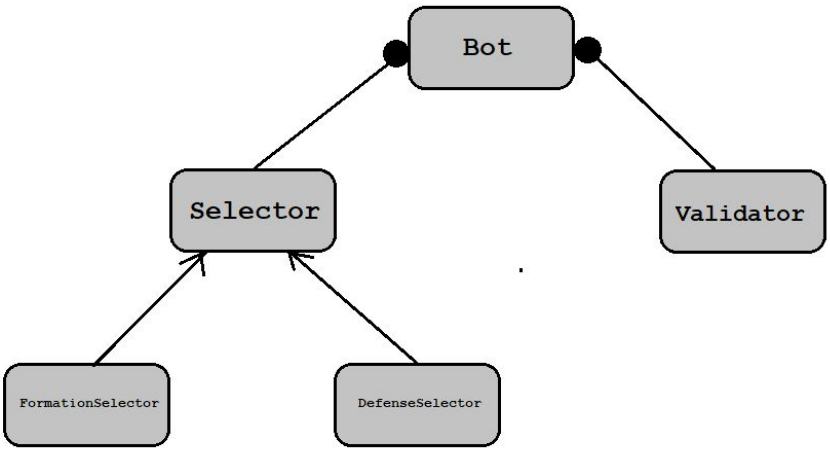
3.5 Strategi *Greedy* yang Dipilih

Strategi yang kami pilih adalah strategi defensif pada posisi tertentu. Kami memilih strategi ini karena setelah membandingkan efektifitas dan efisiensi programnya, kami rasa strategi ini cocok untuk diimplementasikan untuk memenangkan permainan. Dengan melakukan pertahanan sambil melakukan spam serangan, diharapkan musuh akan sulit mendekat dan akhirnya kehabisan *health* akibat lava dan serangan yang dilancarkan. Selain itu, dengan strategi ini, kemungkinan permainan tidak akan berakhir seri karena *safe area* akan dikuasai cacing tim kami. Akibatnya, jika ada cacing musuh yang memasuki *range* serangan, tentu saja mereka akan terkena spam serangan dan akan kehabisan *health*. Untuk efisiensi program juga sangat baik karena tidak menggunakan konsep yang terlalu rumit.

Bab 4 Implementasi dan Pengujian

4.1 Implementasi Program

Program yang kami buat terdiri dari tiga objek: `Bot`, `Selector`, dan `Validator`. Mengingat strategi yang kami gunakan adalah strategi defensif pada posisi tertentu, terdapat dua jenis yang diterapkan: `FormationSelector` sebagai selektor selama cacing belum berada di *fixed point*, dan `DefenseSelector` jika sudah berada di *fixed point*. Gambar 2 menunjukkan hubungannya. Implementasi yang kami buat disimpan dalam `src/java/src/main/java/za/co/entelect/challenge/`.



Gambar 2 Hubungan Elemen-Elemen Program

4.1.1 `Bot`

Bot adalah objek utama dalam program ini. Aksi utamanya adalah metode `run()` yang mengembalikan `Command`, yaitu objek perintah. Di bawah ini adalah *pseudocode* untuk implementasi metode tersebut. (Selengkapnya ada di `Bot.java`.)

```
depend on (posisi cacing yang digunakan sekarang)
      cacing berada di fixed point : selektor adalah DefenseSelector
      cacing tidak berada di fixed point : selektor adalah FormationSelector

iterate
    tempSolution ← solusi dari selektor
    found ← validator mengecek kelayakan (valid) tempSolution
stop (found or selektor tidak punya solusi lain)
    selektor mencari solusi selanjutnya
{ found or selektor tidak punya solusi lain }
```

```
if (found) then
    → perintah sesuai tempSolution
else
    → perintah tidak melakukan apa-apa
```

Jika dikaitkan dengan elemen-elemen *greedy*, fungsi solusi hanyalah berupa ditemukannya solusi (`found`) atau tidak mengingat setiap perintah yang dikembalikan hanyalah satu. Fungsi seleksi dan fungsi objektif bergantung pada solusi yang diberikan selektor sehingga `Bot` menganggap solusi pertama yang diberikan selektor adalah solusi paling optimal. Himpunan kandidat dapat dilihat kekosongannya atau tidak dengan cara mengetahui selektor masih punya solusi atau tidak. Itu juga tanggung jawab selektor. Di sisi lain, fungsi kelayakan ditangani oleh `Validator` agar `tempSolution` layak menjadi solusi.

4.1.2 `FormationSelector`

`FormationSelector` bertugas untuk memberikan solusi-solusi yang ada untuk menuntun cacing menuju posisi yang sudah ditentukan. `FormationSelector` sendiri memberikan tiga solusi yaitu solusi untuk menentukan pergerakannya, solusi untuk menyerang lawan ketika berjumpa di perjalanannya, dan fungsi untuk mencari rute lain ketika menemui cacing teman di rute yang ingin dia tuju.
Di sini kami membuat atribut `arahX` dan `arahY` yang akan menyimpan arah yang menuju ke *cell* yang akan cacing kami proses (*move*, *dig*, *attack*). Berikut adalah *pseudocode*-nya:
(Selengkapnya ada di `greedy/FormationSelector.java`.)

```
moveSolution()
(asumsi fixed point sudah terdefinisi berdasarkan profesi cacing)
if (posisi cacing ≠ fixed point) then
        // Jadi awal pergerakan dimulai dengan bergerak diagonal mendekati fixed point.
        // Kita melakukan pengecekan komponen x dan y posisi cacing dengan fixed point.
        // Ketika keduanya tidak sama berarti bisa bergerak diagonal
        if (fixed point.x > posisi cacing.x and fixed point.y > posisi cacing.y) then
                // Disini mengecek arah diagonal mana yang akan di tempuh.
                // ketika fixed point berada di kanan dan atas posisi cacing maka arahnya adalah
                // kanan atas (1,1)
                arahX ← 1
                arahY ← 1
                → moveDecision(posisi cacing.x + arahX, posisi cacing.y + arahY)
                // moveDecision adalah fungsi untuk membuat cacing kita bergerak atau men-dig dirt
                // pada koordinat yang sudah ditentukan
        else if (kondisi untuk ketika bergerak diagonal ke kiri atas, kiri bawah dan kanan bawah
dengan kode disesuaikan dengan kode di kondisional if pertama)
else
```

```
if (posisi cacing.x = fixed point.x and posisi cacing.y != fixed point.y) then
// kondisi ketika koordinat x posisi cacing sudah sama dengan fixedpoint, jadi kita hanya perlu bergerak ke arah atas atau bawah saja
    if (posisi cacing.y > fixed point.y) then
        arahX ← 0
        arahY ← -1
        → moveDecision(posisi cacing.x + arahX,posisi cacing.y + arahY)
    else
        arahX ← 0
        arahY ← 1
        → moveDecision(posisi cacing.x + arahX,posisi cacing.y + arahY)

else //kondisi dimana koordinat y sudah sama, jadi bergerak ke kanan atau kiri
    if (posisi cacing.x > fixed point.x) then
        arahX ← -1
        arahY ← 0
        → moveDecision(posisi cacing.x + arahX,posisi cacing.y + arahY)
    else
        arahX ← 1
        arahY ← 0
        → moveDecision(posisi cacing.x + arahX,posisi cacing.y + arahY)
```

solutionAttack()

(asumsi *fixed point* sudah terdefinisi berdasarkan profesi cacing)
Ubah koordinat tujuan menjadi arah dengan `resolveDirection(0,0,arahX,arahY)`
→ `ShootCommand(direction)`

solutionAltRoute()

(asumsi *fixed point* sudah terdefinisi berdasarkan profesi cacing)

```
if (arahX ≠ 0 and arahY ≠ 0) then
    → MoveCommand(posisi cacing.x+arahX, posisi cacing.y +arahY-1)
else if (arahX!=0 and arahY=0) then
    if (posisi cacing.y > fixed point.y) then
        → MoveCommand(posisi cacing.x+arahX, posisi cacing.y+arahY-1)
    else
        → MoveCommand(posisi cacing.x+arahX, posisi cacing.y+arahY+1)
else
    if (posisi cacing.x > fixed point.x) then
        → MoveCommand(posisi cacing.x+arahX-1, posisi cacing.y+arahY)
    else
        → MoveCommand(posisi cacing.x+arahX+1, posisi cacing.y+arahY)
```

4.1.3 DefenseSelector

`DefenseSelector` berfungsi untuk melakukan mode *defense* jika cacing telah berada di posisi yang telah ditetapkan. Dalam melakukan mode *defense*, ada lima solusi yang akan

dikembalikan dari `DefenseSelector` yakni `BombEnemy`, `FreezeEnemy`, `ShootEnemy`, `DigDirt`, dan `RandomShoot`. `BombEnemy` berfungsi untuk melempar *banana* ke musuh, `FreezeEnemy` berfungsi untuk melempar *snowball* ke musuh, `ShootEnemy` berfungsi untuk melakukan tembakan biasa ke musuh, `DigDirt` berfungsi untuk menggali *dirt* disekitar cacing, dan `RandomShoot` berfungsi untuk melakukan tembakan random untuk menghindari *command* `DoNothingCommand`. Berikut adalah *pseudocode*-nya: (Selengkapnya ada di `greedy/DefenseSelector.java`.)

```
function BombEnemy → Command
        //Kamus lokal :
        //enemyWorm : Worm;
        //BananaBombCommand(x,y),DoNothingCommand : fungsi yang mengembalikan Command

        //Algoritme
        enemyWorm ← getFirstWormInArea
        if (enemyWorm is not null) then
                → BananaBombCommand(enemyWorm.position.x, enemyWorm.position.y)
        else
                → DoNothingCommand()

function FreezeEnemy → Command
        //Kamus lokal :
        //enemyWorm : Worm;
        //SnowballCommand(x,y),DoNothingCommand : fungsi yang mengembalikan Command

        //Algoritme
        enemyWorm ← getFirstWormInArea
        if (enemyWorm is not null and enemyWorm.roundsUntilUnfrozen=0 ) then
                → SnowballCommand(enemyWorm.position.x, enemyWorm.position.y)
        else
                → DoNothingCommand()

function ShootEnemy → Command
        //Kamus lokal :
        //enemyWorm : Worm;
        //ShootCommand(x,y),DoNothingCommand : fungsi yang mengembalikan Command
        //enemyDir : Direction

        //Algoritme
        enemyWorm ← getFirstWormInArea
        if (enemyWorm is not null and enemyDir is not null) then
                enemyDir ← resolveDirection(currentWorm.position, enemyWorm.position)
                → ShootCommand(enemyDir)
        else
                → DoNothingCommand()

Function DigDirt → Command
        //Kamus lokal :
        //currentWorm : Worm;
        //DigCommand(x,y),DoNothingCommand : fungsi yang mengembalikan Command
        //result : Command
        //enemyDir : Direction
        //block1,block2,block3,block4,block5 : Cell

        //Algoritme
        i traversal [-1..1]
           if (i!=0) then
                Cell block1 ← gameState.map[currentWorm.position.x+i][currentWorm.position.y]
                Cell block2 ← gameState.map[currentWorm.position.x][currentWorm.position.y+i]
                Cell block3 ← gameState.map[currentWorm.position.x+i] [currentWorm.position.y+i]
                Cell block4 ← gameState.map[currentWorm.position.x-i] [currentWorm.position.y+i]
                Cell block5 ← gameState.map[currentWorm.position.x+i][currentWorm.position.y-i]
                if (block1.type = CellType.DIRT)  then
                        result ← DigCommand((currentWorm.position.x+i), (currentWorm.position.y))
```

```
                        → result
                 else if (block2.type = CellType.DIRT) then
                        result ← DigCommand((currentWorm.position.x),  (currentWorm.position.y+i))
                        → result
                else if (block3.type = CellType.DIRT) then
                        result ← DigCommand((currentWorm.position.x+i), (currentWorm.position.y+i))
                        → result;
                else if (block4.type = CellType.DIRT) then
                        result ← DigCommand((currentWorm.position.x-i),
                                                (currentWorm.position.y+i))
                        → result
                else if (block5.type = CellType.DIRT) then
                        result ← new DigCommand((currentWorm.position.x+i), (currentWorm.position.y-i))
                        → result
        → DoNothingCommand()

Function RandomShoot → Command
        //Kamus lokal :
        //ShootCommand(int x,int y),Direction(string arah): fungsi yang mengembalikan Command
        //d : Direction

        //Algoritme
        D ← Direction('N')
        → ShootCommand(d)
```

4.1.4 `Validator`

`Validator` bertugas memberitahukan kevalidan atau kelayakan dari solusi yang diterimanya. `Validator` mengecek hal-hal berikut:

- Koordinat yang diberikan valid, atau tidak keluar dari peta.
- Cacing bergerak tanpa ada tanah (*dirt*) dan cacing lain di tujuannya.
- Cacing melakukan penggalian pada tanah, bukan tempat lain.
- Cacing melakukan penyerangan tanpa mengenai temannya.
- Cacing menembak tanpa terhalang oleh (cacing) temannya ataupun tanah (*dirt*) pada radius tertentu.
- Tidak menerima solusi tidak melakukan apa-apa (*do nothing*).

Kami tidak melampirkan *pseudocode* untuk `Validator` karena implementasinya hanya mengecek terpenuhinya semua kondisi di atas untuk suatu solusi. (Selengkapnya ada di `greedy/Validator.java`.)

4.2 Analisis dari Desain Solusi Algoritma *Greedy*

Setelah beberapa pengujian, ada beberapa kasus yang membuat algoritma ini menarik:

1) Gerakan cacing pemain tidak optimal saat berhadapan dengan tanah tanpa memperhatikan rute yang terbuka (tanpa tanah). Itu bisa saja lebih cepat daripada menggali.
2) Agent tidak memanfaatkan *banana bomb* untuk penggalian yang lebih cepat. Mungkin diharapkan *banana bomb* dapat disimpan untuk *defense*.
3) Jika rute cacing pemain dihalangi oleh cacing lawan, penyerangan hanya dilakukan jika cacing pemain sangat dekat (*adjacent*) dengan cacing lawan. Mungkin itu memastikan musuhnya tidak bergerak.

4) Meskipun terdapat cacing lawan di sekitar cacing pemain, dia tidak menyerangnya jika dia masih bisa bergerak. Hal ini menunjukkan konsistensi untuk bergerak secepat mungkin ke *fixed point*.
5) Pada kenyataannya, *banana bomb* dan *snowball* tidak digunakan sama sekali. Bisa saja itu disebabkan oleh `Validator` yang terlalu protektif.
6) Perintah Select tidak dimanfaatkan. Ini adalah perintah di luar strategi kami mengingat kami bisa melakukannya dengan cara melakukan aksi bergantian. Namun, ini akan lebih optimal jika Select digunakan untuk melewati cacing pemain yang terkena *snowball*.
7) Cacing pemain di *fixed point* yang telah ditentukan akan terbebas dari lava hanya jika *radius* maksimal yang tidak ditempati oleh lava adalah 4 (empat) sel.
8) Perintah pertama selalu tidak dibaca. *We don't know why*.
9) Keberadaan *power-up* diabaikan.
10) Terkadang cacing pemain menembak ke arah yang salah. Padahal, ada arah yang dapat mengenai cacing lawan.
11) Strategi ini cukup stabil. Artinya, jika lawan melakukan gerakan yang sama dalam dua pertandingan yang berbeda dengan asumsi petanya sama, sangat besar kemungkinan strategi ini menang dua kali atau kalah dua kali.

Bab 5 Kesimpulan dan Saran

Algoritma greedy dapat diterapkan dalam permainan Entelect Challenge 2019: Worms. Program ini diharapkan dapat dikembangkan oleh pemrogram lain agar solusi dapat lebih optimal tanpa mengubah makna dari strategi itu sendiri. Sarannya untuk pengembangan bisa mengoptimalkan pada pola penyerangan.

Pustaka

Munir, Rinaldi. 2021. “Algoritma *Greedy* (Bagian 1).” https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/stima20-21.htm (diakses pada 7 Februari 2021).