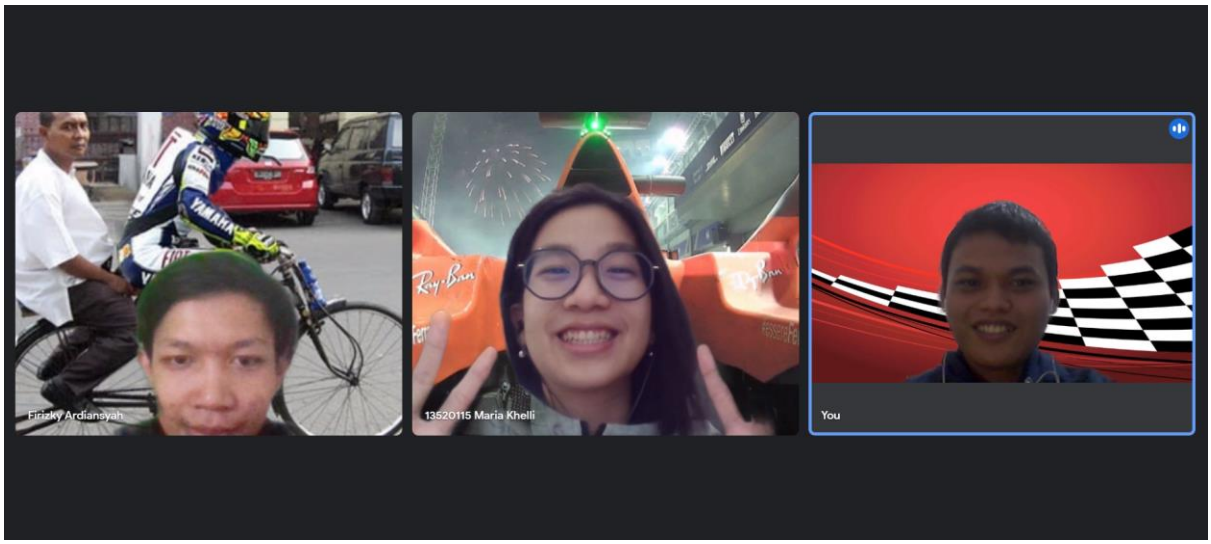
LAPORAN TUGAS BESAR 1 ALGORITMA GREEDY

IF2211 Strategi Algoritma



Nama Kelompok: Traveling F1 Problem

Anggota Kelompok:

Firizky Ardiansyah – 13520095

Maria Khelli – 13520115

Bryan Amirul Husna – 13520146

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFROMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

BAB I

DESKRIPSI TUGAS

Overdrive adalah sebuah game yang mempertandingan 2 bot mobil dalam sebuah ajang balapan. Setiap pemain akan memiliki sebuah bot mobil dan masing-masing bot akan saling bertanding untuk mencapai garis *finish* dan memenangkan pertandingan. Agar dapat memenangkan pertandingan, setiap pemain harus mengimplementasikan strategi tertentu untuk dapat mengalahkan lawannya.

Tugas yang perlu dilakukan adalah mengimplementasikan bot mobil dalam permainan Overdrive dengan menggunakan strategi greedy untuk memenangkan permainan. Program dituliskan dalam bahasa Java. Aturan umum permainan Overdrive adalah:

1. Peta permainan memiliki bentuk array 2 dimensi yang memiliki 4 jalur lurus. Setiap jalur dibentuk oleh block yang saling berurutan, panjang peta terdiri atas 1500 block. Terdapat 5 tipe block, yaitu Empty, Mud, Oil Spill, Flimsy Wall, dan *Finish* Line yang masing-masing karakteristik dan efek berbeda. Block dapat memuat powerups yang bisa diambil oleh mobil yang melewati block tersebut.
2. Beberapa powerups yang tersedia adalah:
    a. Oil item, dapat menumpahkan oli di bawah mobil anda berada.
    b. Boost, dapat mempercepat kecepatan mobil anda secara drastis.
    c. Lizard, berguna untuk menghindari lizard yang mengganggu jalan mobil anda.
    d. Tweet, dapat menjatuhkan truk di block spesifik yang anda inginkan.
    e. EMP, dapat menembakkan EMP ke depan jalur dari mobil anda dan membuat mobil musuh (jika sedang dalam 1 lane yang sama) akan terus berada di lane yang sama sampai akhir pertandingan. Kecepatan mobil musuh juga dikurangi 3.
3. Bot mobil akan memiliki kecepatan awal sebesar 5 dan akan maju sebanyak 5 block untuk setiap babak. Game state akan memberikan jarak pandang hingga 20 block di depan dan 5 block di belakang bot sehingga setiap bot dapat mengetahui kondisi peta permainan pada jarak pandang tersebut.
4. Terdapat *command* yang memungkinkan bot mobil untuk mengubah jalur, mempercepat, memperlambat, serta menggunakan powerups. Pada setiap babak, masing-masing pemain dapat memberikan satu buah *command* untuk mobil mereka. Berikut jenis-jenis *command* yang ada pada permainan:
    a. NOTHING

b. ACCELERATE
c. DECELERATE
d. TURN_LEFT
e. TURN_RIGHT
f. USE_BOOST
g. USE_OIL
h. USE_LIZARD
i. USE_TWEET
j. USE_EMP
k. FIX

5. *Command* dari kedua pemain akan dieksekusi secara bersamaan (bukan sekuensial) dan akan divalidasi terlebih dahulu. Jika *command* tidak valid, bot mobil tidak akan melakukan apa-apa dan akan mendapatkan pengurangan skor.
6. Bot pemain yang pertama kali mencapai garis *finish* akan memenangkan pertandingan. Jika kedua bot mencapai garis *finish* secara bersamaan, bot yang akan memenangkan pertandingan adalah yang memiliki kecepatan tercepat, dan jika kecepatannya sama, bot yang memenangkan pertandingan adalah yang memiliki skor terbesar.

Repository dan peraturan yang lebih lengkap dari permainan Overdrive dapat dilihat pada laman:
https://github.com/EntelectChallenge/2020-Overdrive
https://github.com/EntelectChallenge/2020-Overdrive/blob/develop/game-engine/game-rules.md

BAB II

LANDASAN TEORI

2.1 Deskripsi algoritma greedy

Algoritma greedy merupakan algoritma yang menyelesaikan sebuah masalah secara langkah demi langkah. Algoritma ini akan mencoba mencari solusi optimal lokal pada tiap langkah dengan harapan bahwa hasil akhir dari pencarian tersebut adalah solusi optimal global.

Dalam algoritma greedy, program akan memilih pilihan terbaik pada setiap langkah berdasarkan informasi yang diketahui sampai pada langkah tersebut. Namun, jika ternyata diketahui bahwa pilihan tersebut kurang optimal pada langkah selanjutnya, program tidak diperkenankan mundur ke langkah sebelumnya untuk memperbaiki pilihan (tidak boleh *backtracking*). Akibat perilaku (*behaviour*) tersebut, solusi yang dihasilkan dari algoritma greedy tidak selalu optimal.

2.2 Cara kerja program secara umum

Pertandingan dijalankan oleh sebuah *game engine* yang menerima konfigurasi berisi parameter pertandingan termasuk dua buah bot yang akan dipertandingkan. Bot yang dipertandingkan perlu dicompile dari source code jika menggunakan bahasa pemrograman yang tidak interpreted. Setelah mendapat konfigurasi yang benar, *game engine* dapat dijalankan untuk memulai pertandingan.

Bot berkomunikasi dan menyetorkan aksinya ke game engine melalui standar input dan output (stdin dan stdout). Tiap babak, program bot perlu membaca nomor babak dari stdin dan membaca suatu file json (state.json) untuk mengetahui keadaan permainan saat ini. Bot kemudian menerapkan logika berdasarkan keadaan saat ini untuk menentukan langkah yang tepat dan mengirimkan hasilnya ke stdout melalui format tertentu.

Untuk mengimplementasikan algoritma greedy, perlu dibuat sebuah *project* baru dengan file bot.json. File json ini menyimpan informasi yang akan dibaca game engine berupa nama bot, lokasi file bot yang akan dijalankan, dan bahasa pemrograman bot. Setelah itu, kode program bot ditulis dalam bahasa pemrograman yang dipilih. Logika greedy ditulis berdasarkan pertimbangan state saat ini dengan upaya memaksimalkan keuntungan. Setelah logika selesai ditulis, kode sumber di-*compile* dan file hasil kompilasi ini dispesifikasikan di bot.json untuk dijalankan game engine.

BAB III

APLIKASI STRATEGI GREEDY

3.1 Mapping persoalan

Persoalan overdrive adalah persoalan optimisasi minimasi, yaitu meminimasi jumlah babak untuk sampai *finish* sehingga dapat mengalahkan lawan. Pada persoalan ini, elemen-elemen algoritma greedy adalah

1. Himpunan kandidat: Semua *command* atau aksi yang dimiliki bot mobil, maksimal sebanyak jumlah babak.

   $$H = \{C_1, C_2, C_3, \dots, C_{max\ round}\}$$

   C_n adalah representasi angka dari *command* terpilih. Berikut representasi angkanya.

Accelerate	1	Emp	6
Boost	2	Fix	7
ChangeLane	3.1, 3.2	Lizard	8
Decelerate	4	Oil	9
DoNothing	5	Tweet	10

Tabel 3.1 Daftar representasi angka *command*

   C_n akan bernilai 0 jika permainan telah selesai sebelum jumlah babak mencapai nilai maksimumnya. Nilai 0 adalah nilai yang diberi secara otomatis ketika sudah mencapai garis *finish*.

2. Himpunan solusi: *command-command* yang terpilih

   $S = \{N_1, N_2, \dots, N_{max\ round}\}$ dengan $N = 0, 1, 2, 3.1, 3.2, 4, 5, 6, 7, 8, 9, 10$

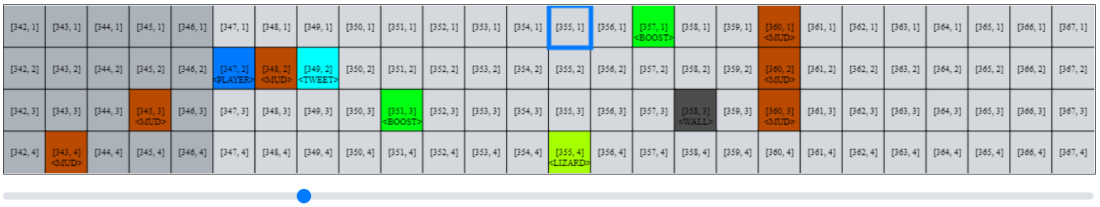
3. Fungsi solusi: Memeriksa apakah posisi pemain sudah lebih dari atau sama dengan kolom ke 1500.
4. Fungsi seleksi: Memilih *command* yang memberikan keuntungan tertinggi yang diukur berdasarkan perhitungan dengan pembobotan pada kondisi-kondisi tertentu. Lebih jauh akan dijelaskan pada **Bagian 4.2.2** bersama struktur data pohon.
5. Fungsi kelayakan: Memeriksa apakah *command* dapat dijalankan atau tidak berdasarkan kondisi mobil saat ini. Misalnya, *command* ChangeLane(-1) tidak layak saat kita berada pada baris paling atas (baris 1 jika 1-indexed).

6. Fungsi objektif: Jumlah babak untuk sampai ke garis *finish* adalah minimum (atau, dengan kata lain, memaksimalkan representasi 0 pada himpunan solusi) dan mengalahkan lawan.

3.2 Eksplorasi alternatif solusi greedy

Alternatif solusi pertama adalah dengan pendekatan linear satu langkah *command* ke depan. *Command* dipilih berdasarkan ada atau tidaknya *obstacle* di depan mobil. Jika ada *obstacle*, mobil akan menghindar dengan memilih jalur depan yang memiliki *obstacle* paling sedikit atau menggunakan power up lizard. Jika tidak ada *obstacle*, mobil meningkatkan kecepatannya hingga maksimum (bisa menggunakan boost) dan jika sudah maksimum, mobil menggunakan power up (tweet, emp, dan oil) untuk memperlambat lawan.

Alternatif pertama cukup bagus, tetapi banyak langkah yang tidak optimum. Seperti pada gambar di bawah, bot mengambil langkah yang tidak tepat dengan berbelok ke kiri padahal langkah terbaik adalah berbelok ke kanan agar dapat menghindari tiga tile mud di depan. Akan tetapi, bot hanya melihat satu langkah *command* ke depan sehingga solusi yang dihasilkan bukanlah yang terbaik.



Gambar 3.2.1 Visualisasi Babak Pada Permainan

Alternatif kedua adalah dengan membuat pohon kombinasi kemungkinan *command* yang bisa dipilih dan menyimulasikannya. State akhir tiap daun (*speed*, *damage*, kepemilikan power ups, dsb.) setelah simulasi dinilai berdasarkan pembobotan parameter, yaitu tiap parameter memiliki nilai bobot yang berbeda-beda. Alternatif ini masih dapat disebut greedy karena pohon dibuat tiap babak, bukan untuk seluruh babak sekaligus.

Jika membuat pohon ini dengan exhaustive search untuk seluruh babak sekaligus, akan membutuhkan kompleksitas waktu $T(c \cdot 8^r)$, dengan r adalah jumlah babak maksimum dan c adalah waktu untuk menghitung setiap simpul sampai keluar dari kalang *while*. Pencarian akan berhenti saat sudah melebihi kedalaman yang dispesifikasi atau sampai di luar jarak pandang (bergantung yang mana yang memenuhi

terlebih dahulu). Namun, dengan algoritma greedy, alternatif ini memiliki kompleksitas waktu $T(cr \cdot 8^4)$ skenario terburuk.

3.3 Analisis efisiensi

Dari segi komputasi, alternatif pertama paling efisien. Bot hanya mengevaluasi beberapa blok di depan secara linear dan langsung memutuskan solusi. Bot ini memiliki kompleksitas terburuk 26 × 4, yaitu jika harus mengevaluasi seluruh block pada peta round saat ini. Jika dirumuskan dalam notasi Big O, kompleksitasnya adalah O(mn), dengan m adalah jumlah lane dan n adalah panjang peta tiap babak.

Alternatif solusi kedua perlu memeriksa beberapa kemungkinan sehingga kompleksitasnya tidak linear. Dalam notasi Big O, kompleksitasnya adalah $O(m^n)$ dengan m adalah banyak jenis *command* dan n adalah kedalaman pencarian dalam satu round. Secara kasar, n dapat dihitung sebagai j / s, dengan j adalah jarak pandang ke depan dan s adalah kecepatan mobil pada round tersebut.

3.4 Analisis efektivitas

Alternatif kedua paling efektif. Algoritma ini menerapkan pencarian dan penilaian menyeluruh sehingga didapatkan solusi optimum lokal tiap babak sesuai dengan pembobotan yang diterapkan. Akan tetapi, karena pohon dibuat tiap babak, bukan menyeluruh, solusinya tidak selalu menghasilkan optimum global.

Di sisi lain, alternatif pertama kurang menghasilkan solusi yang optimal. Hal ini terjadi karena bot tidak memeriksa secara menyeluruh informasi round yang diketahui saat ini untuk menentukan langkah. Bot hanya menerapkan langkah-langkah agresif berdasarkan pertimbangan yang sederhana.

3.5 Strategi yang diimplementasikan

Yang diimplementasikan adalah alternatif solusi kedua karena kompleksitas program tidak dipertimbangkan dalam menentukan kemenangan. Yang menjadi penilaian utama adalah sampai ke garis *finish* secepat mungkin. Meski sebenarnya *game engine* membatasi waktu eksekusi, solusi kedua masih bisa selesai dieksekusi di bawah waktu yang ditentukan. Oleh karena itu, alternatif kedua lebih cocok diterapkan dalam pertandingan untuk meraih kemenangan.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Pseudocode

Method Bot.run

```
procedure Bot.run()
{ Membaca state tiap round dan menghasilkan output hasil logika bot ke stdout }
Deklarasi
    roundNumber: integer
    state : string
    gameState : GameState
    curState : GlobalState
    globalMap : Map
    command, prevCommand : Command
    commands : List of Command
Algoritma:
while (true)
    input(roundNumber)
    gameState ← fromJson(state, GameState.class)
    takeRound(gameState, prevCommand)  { Mengupdate state global }
    commands ← searchBestAction(myCar.speed)  { Mencari langkah terbaik }
    command ← commands.get(0)
    if (isCommandEqual(command, DO_NOTHING))
        command ← bestAttack(commands, curState, globalMap)
        if (isCommandEqual(command, OIL))
            globalMap.setTile(myCar.position.block, myCar.position.lane, OIL_SPILL)
    output (roundNumber, command)
```

Method Bot.searchBestAction

```
function Bot.searchBestAction (v : integer) → List of Command
{ Mencari best action, dengan kedalaman berdasarkan kecepatan v
  Mengembalikan pilihan command terbaik untuk beberapa langkah ke depan}
Deklarasi
    candidates : Search
```

```
    { atribut class Bot }
    curState : GlobalState
    globalMap : Map
Algoritma:
    if (v = 0 or v = 3) then
        { pencarian dengan kedalaman 4 dan kedalaman musuh 0 }
        candidates ← Search(curState, globalMap, 4, 0)
        → candidates.bestAction(curState, globalMap, 0)
    else if (v = 6) then
        candidates ← Search(curState, globalMap, 3, 1)
        → candidates.bestAction(curState, globalMap, 1)
    else
        candidates ← Search(curState, globalMap, 3, 2)
        → candidates.bestAction(curState, globalMap, 2)
```

Method Search.Search

```
function Search(state: GlobalState, globe: Map, depth, deptOpp: integer) → Search
{ Mengembalikan hasil penelusuran }
Deklarasi
    p, curNode, newCmd, candidate : Node
    q : queue of Node
    candidates : List of Node
    { atribut class Search }
    candidateActions : List of Node
Algoritma:
    p ← Node(state)
    q.add(p)
    while (not q.isEmpty()) do
        p ← q.remove()
        if (p.Actions.size() == depth) then
            candidates.add(p)
        if (p.Actions.size() < depth) then
            { Mengecek command yang valid dan menambahkan ke queue }
            for (newCmd in validAction(p.State.player) do
```

```
            curNode ← p
            curNode.Actions.add(newCmd)
            curNode.State = simulateActions(newCmd, predictAction(p.State, globe,
                                                depthOpp), p.State, globe);
            if (curNode.State.player.pos_x >= globe.nxeff) then
               candidates.add(curNode)
            else
               q.add(curNode)
   { Menambahkan kandidat yang tidak kosong ke daftar kandidat objek }
   for (candidate in candidates) do
      if (candidate.Action.size() != 0) then
         candidateActions.add(candidate)
   → this
```

Method Search.bestAction

```
function bestAction (state : GlobalState, globe : Map, depthOpp : integer)
                                                        → List of Command
{ Dari sekumpulan kandidat (atribut class), mengembalikan sekuens Command
terbaik }
Deklarasi
   idmx : integer
   mx, currentScore : real
   finalGame : boolean
   node : Node
   { atribut class }
   candidateActions : List of Node
Algoritma:
   idmx ← -1
   finalGame ← false
   for (node in candidateActions) do
      if (node.State.player.pos_x >= 1500) then
         finalGame ← true
         break
   if (finalGame) then
```

```
    { Jika sudah di round terakhir (final round), prioritaskan command yang
      menghasilkan kecepatan tertinggi }
    mx ← Integer.MIN_VALUE
    for i ← 0 to candidateActions.size() do
      if (mx < node.State.player.speed and node.State.player.pos_x >= 1500)
        idmx ← i
        mx ← node.State.player.speed
  else
    { Melakukan scoring dari command-command di kandidat }
    mx ← -Double.MAX_VALUE
    for i ← 0 to candidateActions.size() do
      node ← candidateActions.get(i)
      currentScore ← scorePlayer(node, state, globe, depthOpp)
      if (mx < currentScore) then
        idmx ← i
        mx ← currentScore
  → candidateActions.get(idmx).Actions;
```

Method Scoring.scorePlayer

```
function scorePlayer (n : Node, initialState : GlobalState, globe : Map,
                      depthOpp : integer) → real
{ Menghasilkan skor setelah menjalankan command pada Node n. Skor dihitung
dengan membandingkan n.State dengan initialState }
Deklarasi
  score : real
  immediateNextState : GlobalState
Algoritma:
  score ← multiplyWeights (initialState, n.State, false)
  { Pilih command dengan mempertimbangkan langkah langsung selanjutnya yang
    memberikan keuntungan tertinggi dulu }
  immediateNextState ← simulateActions (n.Actions.get(0),
                predictAction(initialState, globe, depthOpp), initialState, globe)
  score ← Weights.AFTERSTATE * multiplyWeights(n.State,
            immediateNextState, false)
```

```
    → score
```

Method Scoring.multiplyWeights

```
function multiplyWeights (initialState, finalState : GlobalState,
                                    opponentWise : boolean) → real
{ Menghasilkan skor dengan pembobotan }
Deklarasi
    score  : real
    initialPlayer, finalPlayer : Player
Algoritma:
    score ← 0
    if (not opponentWise) then
        initialPlayer ← initialState.player
        finalPlayer ← finalState.player
    else
        initialPlayer ← initialState.enemy
        finalPlayer ← finalState.enemy
    score ← score + (finalPlayer.pos_x - initialPlayer.pos_x) * Weights.POSITION
    score ← score + (finalPlayer.speed – initialPlayer.speed) * Weights.SPEED
    { … Seluruh atribut player dinilai dengan pembobotan … }
    score ← score + (finalPlayer.emp – initialPlayer.emp) * Weights.EMP
    ← score
```

Method simulateActions

```
function simulateActions (PlayerAction, EnemyAction : Command,
                                    InitState : GlobalState, globe : Map) → GlobalState
{ Menerima command player dan (perkiraan) enemy, menjalankan simulasi dan
menghasilkan state setelah command tersebut dijalankan }
Deklarasi
    ret : GlobalState
    player, enemy : Player
    PlayerPath, EnemyPath : Path
    PlayerResource, EnemyResource, PlayerCyber, EnemyCyber : Resource
Algoritma:
    ret ← InitState
```

```
player ← ret.player
enemy ← ret.enemy
if (player.nBoost > 0) then
    player.nBoost--;
    if (player.nBoost == 0) then
        player.speed = getCurrentSpeedLimit(player.damage)
    else if (isCommandEqual(PlayerAction, Abilities.DECELERATE) then
        player.nBoost = 0
if (enemy.nBoost > 0) then
    enemy.nBoost--;
    if (enemy.nBoost == 0) then
        enemy.speed = getCurrentSpeedLimit(player.damage)
    else if (isCommandEqual(EnemyAction, Abilities.DECELERATE) then
        enemy.nBoost = 0
player.getFromAction(PlayerAction)
enemy.getFromAction(EnemyAction)
{ Menghitung path }
PlayerPath.updatePath(PlayerAction, player)
EnemyPath.updatePath(EnemyAction, enemy)
playerCyber ← Resource(globe, PlayerPath, PlayerAction, true)
enemyCyber ← Resource(globe, EnemyPath, EnemyAction, true)
for (P in PlayerCyber.cyberPos) do
    ret.deleteCyberTruck(P.x, P.y)
for(P in EnemyCyber.cyberPos) do
    ret.deleteCyberTruck
PlayerPath.resolveCollision (EnemyPath, PlayerAction, EnemyAction)
PlayerResource ← Resource(globe, PlayerPath, PlayerAction, false)
EnemyResource ← Resource(globe, EnemyPath, EnemyAction, false)
player.updatePos(PlayerPath)
enemy.updateResource(EnemyResource)
→ ret
```

Method predictAction

```
function predictAction (state: GlobalState, globe: Map, depth: integer)→Command
```

```
{ Menghasilkan prediksi musuh akan melakukan command apa }
Deklarasi
Algoritma:
  { Jika musuh di depan atau kedalaman pencarian musuh 0, kembalikan
    ACCELERATE secara default. Jika musuh di depan kita tidak memiliki data
    peta sehingga tidak bisa memprediksi }
  if (depth = 0 or state.enemy.pos_x > globe.nxeff) then
    → ACCELERATE
  { Jika musuh di depan persis mobil dan kecepatannya 0, kembalikan
    DO_NOTHING }
  if (state.player.pos_x = state.enemy.pos_x – 1 and
      state.player.pos_y = state.enemy.pos_y and state.enemy.speed = 0) then
    → DO_NOTHING
  → OpponentMove(state, globe, depth).bestMove(state, globe).get(0)
```

4.2 Penjelasan struktur data

Pada permasalahan ini, struktur data diimplementasikan sebagai sebuah kelas, bukan *abstract data type*. Terdapat dua subbagian, yaitu struktur data dasar yang diperlukan agar *game* dapat berjalan dengan baik dan struktur data tambahan untuk menerapkan algoritma greedy.

4.2.1 Struktur data dasar

Struktur data dasar pada permasalahan ini diambil dari bot referensi yang diberikan. Struktur data atau kelas pada bagian ini bersifat krusial. Apabila ada yang dihapus, bisa saja program tidak dapat berjalan dengan baik. Berikut kelas-kelas utama berdasarkan foldernya.

1. Folder command
   1.a AccelerateCommand, kecepatan mobil akan bertambah 1 level (jika belum mencapai kecepatan maksimal) ketika kelas ini dikembalikan (di-*return*) pada *game engine*.
   1.b BoostCommand, membuat kecepatan objek Car pada level *boost* sebanyak 5 babak jika kelas Car tidak memanggil DecelerateCommand dan tidak menabrak *obstacle*.

1.c ChangeLaneCommand, menyebabkan objek Car yang mengembalikan kelas ini berubah satu lane jika valid. Misal, Car berada pada lane 2 saat babak n-1, maka pada babak n, Car akan berada pada lane 3 jika argumennya 1, atau lane 1 jika argumennya -1.

1.d Command (Interface), merupakan kelas abstrak untuk kelas Command yang ada di dalam folder ini.

1.e DecelerateCommand, kebalikan dari AccelerateCommand, objek Car akan berkurang 1 level (jika belum mencapai kecepatan minimal) ketika dikembalikan pada *game engine*.

1.f DoNothingCommand, tidak ada tindakan yang dilakukan pada objek Car. Kecepatan objek Car yang mengembalikan kelas ini pada *game engine* akan tetap seperti babak sebelumnya.

1.g EmpCommand, menembakkan ledakan pada lane, lane - 1, dan lane + 1 dari lane objek Car yang mengembalikan kelas ini. Ketika mengenai objek Car lain, objek tersebut akan langsung berkurang kecepatannya menjadi 3 dan akan diam untuk beberapa babak.

1.h FixCommand, mengurangi *damage* milik objek Car yang mengembalikan kelas ini. Kelas Car tidak akan pindah ketika melakukan FIX.

1.i LizardCommand, melompati semua *obstacle* di depan objek Car yang mengembalikan kelas ini sehingga kelas Car tersebut tidak mendapat *damage*.

1.j OilCommand, menumpahkan oli pada kelas Terrain di posisi objek Car untuk menambah *obstacle* pada map.

1.k TweetCommand, meletakkan *cybertruck* pada posisi x dan y argumen yang diberikan saat kelas dikembalikan pada *game engine*.

2. Folder entities

2.a Car, merupakan subjek utama dalam permasalahan ini. Kelas ini merupakan representasi pemain di dalam *game*. Atribut-atributnya meliputi identifier, posisi, kecepatan, kondisi (state) setelah melakukan *command* sebelumnya, *power up* yang dimiliki, status *boosting*, dan counter untuk *boosting*.

2.b GameState, kelas yang menyimpan kondisi yang dapat diketahui oleh pemain, misalnya letak lawan, map yang dapat dilihat dalam bentuk larik kelas Lanes.

2.c Lane, kelas yang menyimpan koordinat (x dan y) melalui kelas Position. Kelas ini juga menyimpan informasi tanah (Terrain) pada koordinat tersebut. Selain

itu, juga menyimpan informasi apakah di koordinat tersebut ada *cybertruck* atau ada objek Car lain.

2.d Position, menyimpan block dan lane (koordinat x dan y) yang merupakan representasi koordinat letak kelas Car.

3. Folder enums

3.a Direction

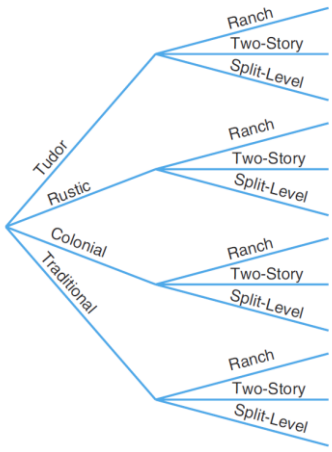
3.b PowerUps, berisi enumerasi dari *drop* atau *power up* yang bisa digunakan pemain, contohnya BOOST, OIL, TWEET, LIZARD, EMP.

3.c State, berisi enumerasi dari *state* setelah kita melakukan command tertentu pada babak sebelumnya, misalnya USED_BOOST ketika kita berhasil menggunakan *power up* BOOST.

3.d Terrain, berisi enumerasi dari representasi tanah yang ada di dalam game, misalnya *obstacle* WALL yang akan memberi *damage* pada kelas Car jika dilewati.

4.2.2 Struktur data tambahan

Dalam implementasi algoritma greedy pada permasalahan ini, kami menambahkan struktur data pohon sebagai struktur data utama untuk membangkitkan *command* yang dimasukkan ke fungsi seleksi. Pohon dipilih karena karakteristiknya yang memiliki level atau kedalaman. Selain itu, algoritma breadth-first-search (BFS) cocok dengan pohon. Ilustrasinya adalah sebagai berikut. Analoginya tulisan tersebut adalah *command*.



Gambar 4.2.2 Ilustrasi pohon

Perlu diperhatikan bahwa pohon digunakan hanya untuk “mengunjungi” seluruh simpul dalam dan daun dari pohon (*searching*), bukan untuk menyaring

keputusan-keputusan yang ada seperti yang ada pada pohon keputusan. Maka dari itu, simpul dari pohon hanya menyimpan GlobalState (state virtual untuk disimulasikan) dan action (command yang diambil oleh simpul tersebut). Pohon akan diimplementasikan pada kelas Search (untuk menghitung langkah kita) dan OpponentMove (untuk menghitung langkah lawan).

Intinya, algoritma greedy akan mempertimbangkan pilihan terbaik berdasarkan dua atau tiga langkah ke depan—bergantung dari kedalaman pohon yang dispesifikasi dan bergantung pada situasi yang diketahui oleh player pada saat itu—sehingga tidak hanya satu langkah seperti yang dilakukan bot referensi. Pertimbangan pilihan terbaik akan dilakukan oleh kelas statis Scoring yang menghitung *command-command* yang disimulasikan berdasarkan bobot yang ada pada kelas statis Weights. Kelas Scoring juga menjadi komponen utama dalam fungsi seleksi.

Perincian dari kelas tambahan atau struktur data tambahan adalah sebagai berikut. Kelas yang akan dirincikan adalah seluruh kelas tambahan dari kelas utama.

1. Folder algorithm
   1.a Node, merupakan simpul yang digunakan dalam algoritma searching pada pohon. Menyimpan informasi objek GlobalState dan larik objek Command.
   1.b OpponentMove, implementasi algoritma searching untuk memprediksi gerakan lawan. Asumsinya adalah lawan menggunakan algoritma yang sama dengan pemain.

   Kelas ini memiliki atribut larik objek Node dan antrian Node untuk melakukan BFS. Method bestMove akan mengembalikan larik objek Command yang merupakan urutan Command terbaik pada saat itu. Command yang dikembalikan hanya Command *movement* (bukan misalnya EMP).
   1.c Scoring, menghitung skor fungsi seleksi untuk pemain dan lawan. Prediksi *command* lawan juga dipertimbangkan karena mempengaruhi *command* kita.
   1.d Search, implementasi algoritma searching untuk pemain. Mirip seperti kelas OpponentMove, hanya saja perspektifnya diubah dari lawan menjadi pemain.
2. Folder globalentities
   2.a GlobalState, kelas ini menyimpan objek Player dari pemain dan lawan. Kelas ini juga menyimpan posisi cybertruck saat sebelum diperbarui dan setelah diperbarui. Tujuannya adalah mendeteksi apakah cybertruck pindah atau tidak

sehingga dapat memperbarui objek Map. Jika tidak, posisi sebelum dan sesudah akan sama nilainya.

2.b Map, menyimpan peta permainan dari awal sampai sejauh yang dapat dilihat pemain. Memiliki atribut nxeff yang merupakan posisi kolom terjauh yang pernah dilihat oleh pemain. Representasi struktur data Map adalah matriks dari objek Tile.

Kelas Map memiliki method updateNewRound yang akan memperpanjang nilai kolom efektif ketika memasuki babak baru. Selain itu, memiliki method createSimulation untuk meletakkan *cybertruck* dan tumpahan oli secara virtual.

2.c Path, merupakan kelas yang merepresentasikan lintasan yang dilalui oleh pemain atau lawan. Memiliki method resolveCollision untuk memperbarui path ketika menabrak objek Car lain. Kemudian, ada method allPath yang mengembalikan larik dari objek Tile. Analoginya seperti getBlocks pada bot referensi.

2.d Player, kelas ini adalah representasi pemain yang digunakan untuk mencatat keadaan pemain dan lawan sebagai "catatan pribadi" (bukan kerangka dasar dari bot referensi).

Kelas ini menyimpan id, posisi x dan y, kecepatan, *damage*, seluruh power ups dalam representasi primitif integer (berbeda dengan kelas Car yang menyimpan larik dari objek) dan skor dari pemain.

Kelas ini juga memiliki method update untuk pemain dan lawan yang memperbarui atribut dari kelas ini.

2.e Resource, hanya menyimpan power up dalam representasi primitif integer. Digunakan untuk mengumpulkan *power up* yang ada pada Tile dan memberi penalti apabila menabrak *obstacle*.

2.f Tile, adalah kelas representasi balok-balok dalam map. Kelas ini memiliki koordinaat x dan y, kemudian ada dua objek Terrain, yaitu tile dan layer yang digunakan untuk menangani kasus cybertruck yang berada di atas Terrain *nonempty*. Kelas ini merupakan pengganti Terrain.

3. Folder utils

3.a Abilities, merupakan kelas statis yang menyimpan konstnta objek Command yang sudah diinisialisasi. Memiliki method convertOffensive yang mengubah *command* yang menggunakan power up menjadi objek DoNothingCommand.

3.b Actions, merupakan kelas untuk mensimulasikan GlobalState berdasarkan *command* pemain dan lawan. Memiliki method validAction untuk menentukan *command* apa saja yang valid. Selain itu, memiliki command predictAction yang memprediksi gerakan lawan.

Kelas ini juga memiliki method bestAttack yang merupakan pelengkap dari kelas Search yang hanya membangkitkan *command* pergerakan. Di sini, ditambahkan *command* ofensif seperti TWEET atau EMP.

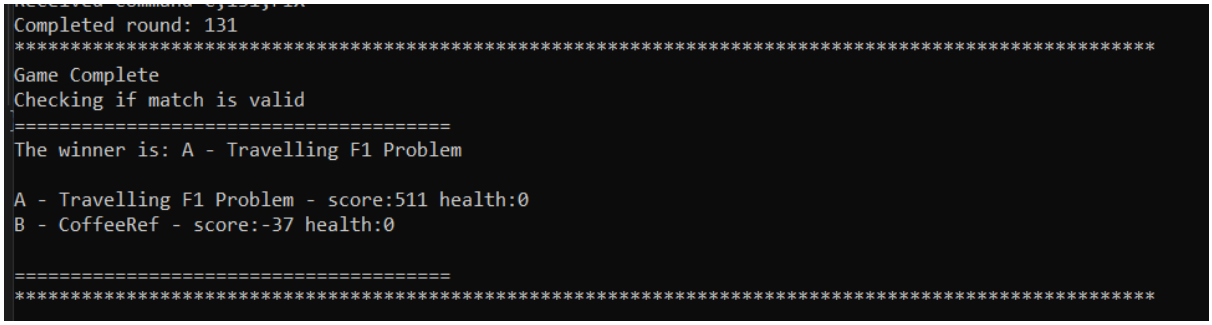
3.c LogState, kelas ini memiliki atribut objek Command, objek GlobalState sebelum dan GlobalState sesudah. Atribut objek GlobalState sesudah adalah hasil dari GlobalState sebelum yang telah diberi implementasi objek Command. LogState juga memiliki method calcOpponentCommand yang memprediksi gerakan lawan (yang sebenarnya) berdasarkan state sebelum dan sesudah.

3.d Supports, merupakan kelas statis yang menyimpan method untuk mendukung kelas lain, misalnya mencari batas kecepatan maksimal berdasarkan *damage*, mencari kecepatan hasil akselerasi, deakselerasi, dan *boosting*.

3.e Weights, merupakan kelas statis yang menyimpan konstanta bobot penilaian.

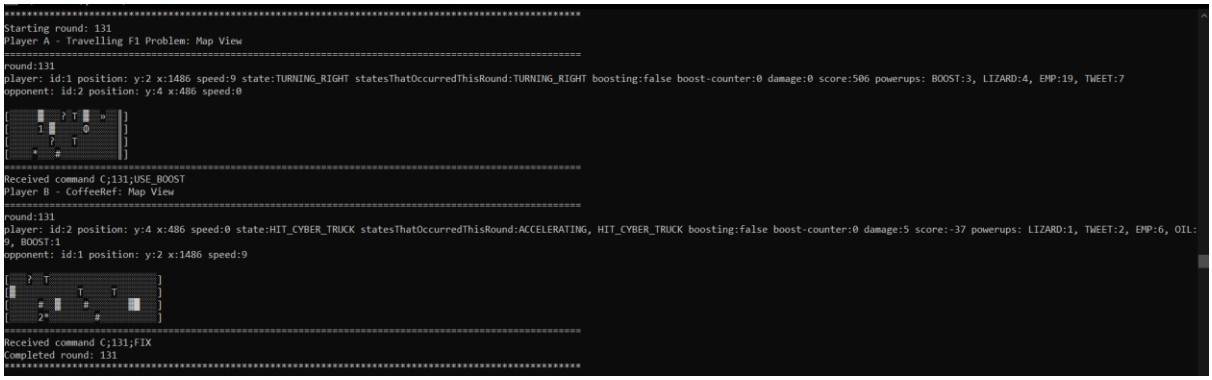
4.3 **Pengujian**

Kami melakukan pengujian dengan *reference bot* yang disediakan, berikut adalah peroleh akhirnya.



Gambar 4.3.1 Perolehan Hasil Pengujian

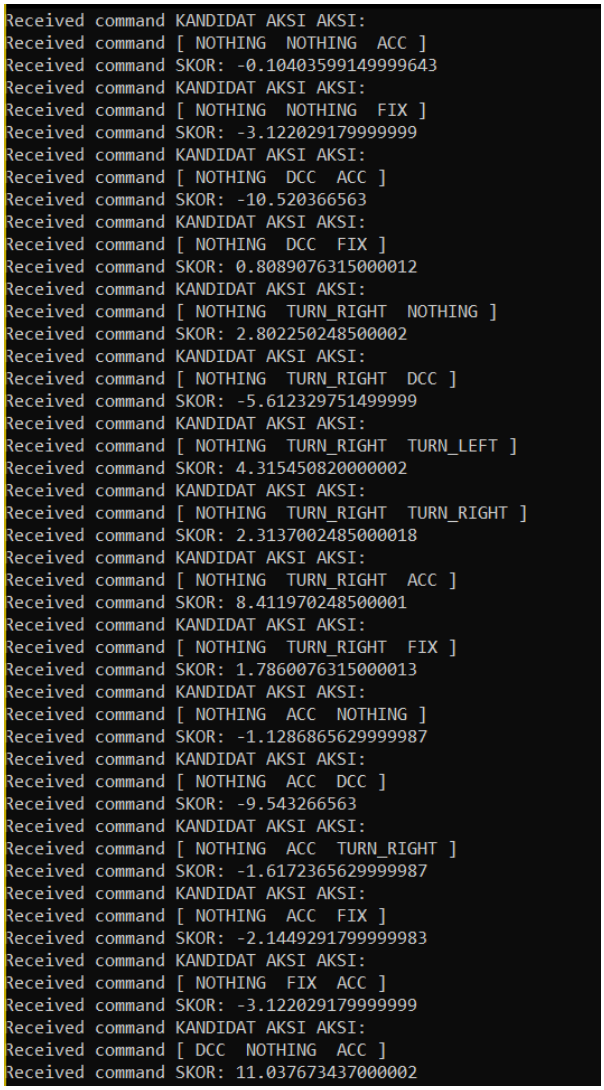
Untuk menyelesaikan sebuah permainan, bot hanya perlu melakukan 131 buah babak. Diperoleh juga bahwa perbedaan skor lawan dengan skor bot mencapai 548 unit skor. Adapun keadaan akhir bot dengan *reference bot* sebagai berikut.



Gambar 4.3.2 Kondisi Babak Terakhir Hasil Pengujian

Perbedaan posisi absis pemain dengan *reference bot* mencapai 1000 blok. Pemain juga menggunakan *resource*-nya dengan baik, terlihat dari sisa *resource* yang dimiliki pemain tidak menumpuk. Kerusakan pemain juga sudah ditangani setiap saat, terbukti pada akhir babak, pemain tidak memiliki kerusakan sama sekali.

Adapun detail desain algoritma *greedy* yang kami implementasikan dapat dilihat pada tangkapan layar berikut.



Gambar 4.3.3 Visualisasi Pohon Pencarian Kandidat Aksi

Kandidat-kandidat berisi noda-noda pada tree yang dilalui. Aksi-aksi tersebut disimulasikan dan dicari skornya berdasarkan hasil simulasi tersebut. Skor dicari berdasarkan bobot masing-masing parameter. Parameter tersebut berupa posisi pemain setelah dan sebelum simulasi, kecepatan pemain setelah simulasi, kurasakan pemain sebelum dan sesudah simulasi, serta *resource-resource* yang berhasil didapatkan selama simulasi. Aksi dengan skor terbesar akan menjadi kandidat utama dari algoritma *greedy*.

Setelah memaksimalkan skor hasil simulasi, kandidat aksi terbaik berdasarkan skor akan lebih lanjut diproses untuk menemukan aksi *attack* terbaik. Aksi *attack* hanya akan diperoleh jika salah satu aksi pada kandidat utama adalah *DO_NOTHING* dengan begitu, perhitungan *movement* hasil skoring sebelumnya tidak

berpengaruh. Gabungan kandidat utama dengan pencarian aksi *attack* terbaik dari algoritma ini akhirnya menjadi langkah selanjutnya bagi pemain di babak tersebut.

Algoritma *greedy* yang diterapkan pada bot *overdrive* ini, sejauh percobaan yang dilakukan sudah berjalan dengan baik. Namun, ada beberapa hal yang membuat algoritma *greedy* kami kurang baik di antaranya adalah sebagai berikut

1. Saat melakukan simulasi, prediksi aksi lawan tidak terlalu akurat, hanya berdasarkan tebakan dan data yang ada dari babak tersebut.
2. Konstrain waktu tidak memungkinkan bagi bot untuk menyimpan peta global dan menyalinnya dengan cepat, sehingga pada beberapa kasus, prediksi posisi akhir lawan menjadi bias. Akibatnya penyimpanan *cybertruck* tidak optimal.
3. Kurangnya eksperimen membuat bobot yang digunakan tidak sesuai dengan yang ada pada lapangan.

4.4 **Source code** dan video

Source code : https://github.com/bryanahusna/Stima-Overdrive-Bot
Video : https://youtu.be/s8KcNIYBPsM

BAB V

KESIMPULAN DAN SARAN

5.1 **Kesimpulan**

Algoritma greedy mampu menghasilkan solusi yang cukup optimal untuk permainan Overdrive. Optimum lokal dihasilkan tiap *round* dengan harapan dapat menghasilkan optimum global, yaitu sampai garis *finish* dengan jumlah *round* serendah mungkin dan mengalahkan lawan. Akan tetapi, hasilnya tidak selalu optimum jika dibandingkan dengan *exhaustive search* yang memeriksa semua kemungkinan pada seluruh babak.

5.2 **Saran**

Mekanisme dan nilai pembobotan parameter masih belum sempurna. Karena sifat peta pertandingan yang acak, lebih banyak eksperimen diperlukan untuk mendapatkan bobot yang optimal.

Pencarian bobot paling optimal perlu dilakukan eksperimen lebih banyak agar bekerja lebih umum dan lebih optimal. Salah satu cara pencarian bobot untuk mencari kandidat terbaik adalah dengan menggunakan *machine learning*. Penggunaan *machine learning* atau algoritma lain yang lebih kompleks juga bisa digunakan sebagai optimalisasi lain seperti prediksi aksi lawan, prediksi *resource* pada blok tertentu, dan sebagainya.

DAFTAR PUSTAKA

Munir, Rinaldi. 2021. Algoritma Greedy (Bagian 1) [PowerPoint slides]. Diambil dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf

Munir, Rinaldi. 2021. Algoritma Greedy (Bagian 2) [PowerPoint slides]. Diambil dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf

Munir, Rinaldi. 2022. Algoritma Greedy (Bagian 3) [PowerPoint slides]. Diambil dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf