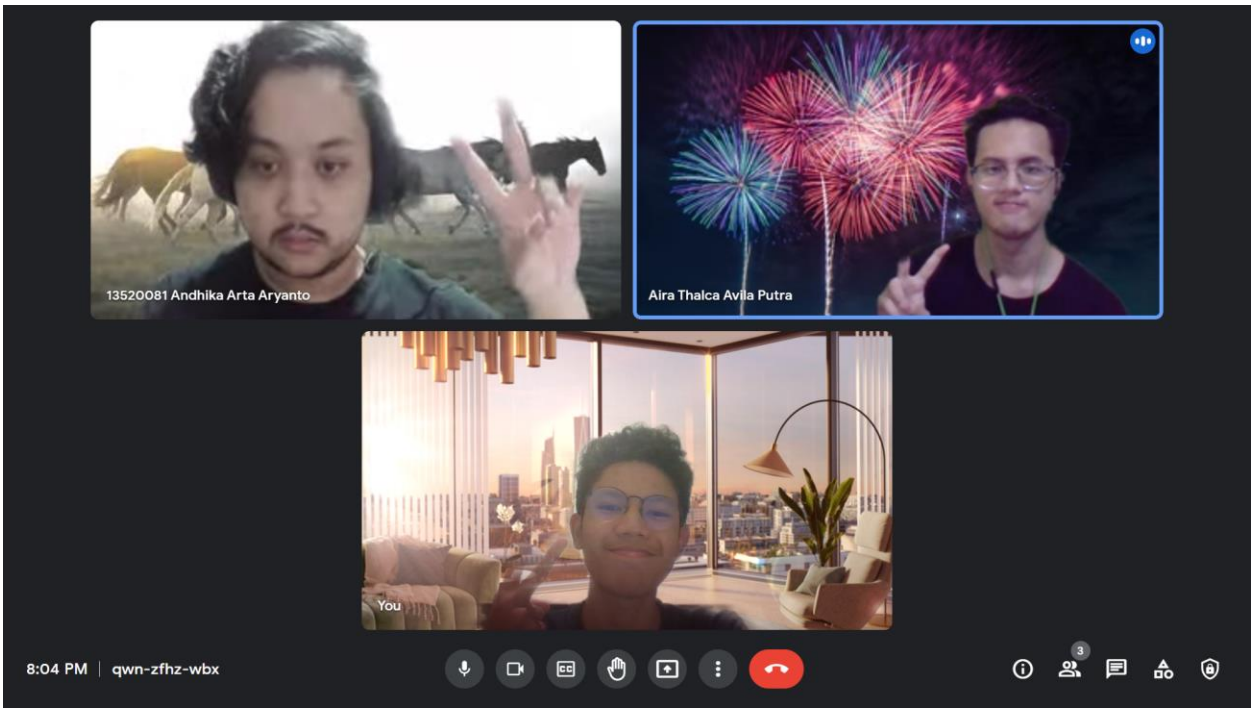
Tugas Besar 1 IF2211 Strategi Algoritma

Semester II tahun 2021/2022

Pemanfaatan Algoritma *Greedy* dalam Aplikasi Permainan "Overdrive"

Disusun oleh:

Rayhan Kinan Muhannad	13520065
Andhika Arta Aryanto	13520081
Aira Thalca Avila Putra	13520101



PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

Bab 1: 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.

Bahasa pemrograman yang digunakan pada tugas besar ini adalah Java. Bahasa Java tersebut digunakan untuk membuat algoritma pada *bot*. IDE yang digunakan untuk membantu membuat projek ini adalah Intellij IDEA. Intellij IDEA merupakan IDE yang kompatibel dengan bahasa Java, dikarenakan beberapa *tools*-nya seperti Maven sudah *built in* tanpa perlu menambahkan *extension*. Untuk menjalankan permainan, digunakan sebuah *game engine* yang diciptakan oleh Entellect Challenge yang terdapat pada *repository* githubnya. *Game engine* yang dibuat oleh Entellect Challenge menggunakan bahasa Scala sebagai bahasa pemrograman utama dalam pembuatannya.

Spesifikasi permainan yang digunakan di dalam tugas besar ini disesuaikan dengan spesifikasi permainan "Overdrive" yang terdapat pada *repository* Entellect Challenge. Beberapa peraturan umum yang harus diikuti oleh *bot* adalah diantaranya:

1. Peta permainan berbentuk *array* dua dimensi dengan empat jalur (*lane*) lurus. Setiap *lane* dibentuk oleh *block* yang sekuensial, dengan setiap *lane* terdiri atas 1500 *blocks* (panjang peta adalah 1500 *blocks* dari *start* hingga *finish line*). Terdapat lima jenis *block*, yaitu *Empty*, *Mud*, *Oil Spill*, *Flimsy Wall*, serta *Finish Line*. Berikut ini adalah penjelasan detail mengenai setiap jenis *block* tersebut:
    - *Empty*: *Block* ini tidak memiliki efek apa-apa terhadap *car*. *Block* ini dapat mengandung *power up*.
    - *Mud*: *Block* ini mengandung lumpur dan jika dilewati oleh *car* maka *speed* dari *car* tersebut akan melambat satu tingkatan serta *car* akan menerima -1 *damage*.
    - *Oil Spill*: *Block* ini mengandung tumpahan oli dan jika dilewati oleh *car* maka *speed* dari *car* tersebut akan melambat satu tingkatan serta *car* akan menerima -1 *damage*. *Oil spill* ini dihasilkan oleh *power up Oil Item* dimana *car* pengguna *Oil Item* ini menumpahkan oli pada *block* saat diaktifkan.

- *Flimsy Wall*: *Block* ini mengandung tembok yang rapuh dan jika dilewati oleh *car* maka *speed* dari *car* tersebut akan melambat secara drastis hingga *speed* paling lambat serta *car* akan menerima -2 *damage*.
- *Finish Line*: *Block* ini mengandung garis akhir dan jika dilewati oleh *car* maka *car* yang terlebih dahulu melewati *block* ini akan menang. Bisa kedua *car* melewati *finish line* secara bersamaan, maka *car* dengan *speed* terbesarlah yang menang. Jika kedua *car* tersebut memiliki *speed* yang sama, maka ditentukan dari peraih *score* terbesar.

2. Terdapat beberapa *power up* yang dapat membantu pemain dalam memenangkan permainan. Untuk menggunakan *power up*, pemain terlebih dahulu harus mengambil *power up* tersebut pada *Empty block* dan memberi perintah *USE_<power up>* untuk menggunakan *power up* tersebut. Perlu diingatkan bahwa jika menggunakan perintah *USE_<power up>* tetapi tidak mempunyai *power up* yang ingin digunakan, maka terdapat sanksi pengurangan poin. Terdapat lima jenis *power up*, yaitu *Oil Item*, *Boost*, *Lizard*, *Tweet*, serta *EMP*. Berikut ini adalah penjelasan detail mengenai setiap *power up* tersebut:
    - *Oil Item*: Menumpahkan genangan oli pada *block* tempat *car* berada. Setiap *car* yang melewati genangan oli tersebut akan melambat satu tingkatan dan menerima -1 *damage*.
    - *Boost*: Mendapatkan penambahkan *speed* secara drastis dan berlaku selama lima ronde. Efek *boost* akan hilang bila *car* terkena efek perlambatan dan *speed* dari *car* akan melambat menjadi *maximum_speed*.
    - *Lizard*: Membuat *car* menghindari semua *power up pickups*, *obstacles*, serta *car* dari pemain lainnya selama satu ronde. Tetapi, efek tersebut tidak berlaku pada *block* terakhir yang ditempati oleh *car* pada ronde selanjutnya.
    - *Tweet*: Memanggil *cyber truck* pada *block target* (terdapat parameter sumbu *X* dan sumbu *Y* dari *block* tempat *cyber truck* dipanggil). Ketika suatu *car* menabrak *cyber truck* tersebut maka *car* tersebut akan melambat secara drastis hingga *speed* paling lambat dan *car* menerima -2 *damage* (layaknya *car* menabrak *flimsy wall*). *Car* juga akan *stuck* di belakang *cyber truck* selama ronde tersebut. Setelah terjadi tabrakan antara *cyber truck* dengan suatu *car*, *cyber truck* tersebut akan menghilang dari *map*. Hanya mungkin terdapat satu *cyber truck* yang dipanggil oleh pemain pada

suatu waktu, jika pemain memanggil *cyber truck* dan *cyber truck* sebelumnya belum tertabrak maka *cyber truck* akan dipindahkan ke posisi baru.

- *EMP*: Menembakkan gelombang EMP (*Electromagnetic Pulse*) pada tiga *lane*: *lane* kiri pemain, *lane* pemain, serta *lane* kanan pemain dengan jangkauan *block* hingga *finish line*. Jika terdapat *car* yang terkena gelombang EMP, maka *car* tersebut akan melambat hingga *speed* paling lambat selama ronde tersebut.

3. *Car* memiliki kecepatan awal sebesar 5 *blocks* per ronde. Selain itu, *game state* memperbolehkan *bot* melihat sejauh 20 *blocks* ke depan dan 5 *blocks* ke belakang. Seperti yang sudah dijelaskan sebelumnya, *speed* dari *car* dapat berubah-ubah tergantung tingkatan yang sudah ditentukan oleh *game*. Berikut ini adalah tingkatan kecepatan serta nilai kecepatannya dalam *block* per ronde dari *car*:
    - *MINIMUM_SPEED*: 0 *block* per ronde
    - *SPEED_STATE_1*: 3 *blocks* per ronde
    - *INITIAL_SPEED*: 5 *blocks* per ronde
    - *SPEED_STATE_2*: 6 *blocks* per ronde
    - *SPEED_STATE_3*: 8 *blocks* per ronde
    - *MAXIMUM_SPEED*: 9 *blocks* per ronde
    - *BOOST_SPEED*: 15 *blocks* per ronde
4. Terdapat beberapa *command* yang memungkinkan *bot* mengubah keadaan *car*, seperti mengubah *lane*, menambahkan kecepatan, mengurangi kecepatan, menggunakan *power up*, dan lain-lain. Pada setiap ronde, setiap pemain dapat memberikan satu *command* untuk dieksekusi oleh *car* mereka. *Command* dijalankan secara bersamaan untuk setiap pemain, bukan secara sekuensial. Berikut ini adalah daftar *command* yang dapat dieksekusi oleh *car* serta penjelasannya:
    - *NOTHING*: *Command* ini tidak akan mengubah *state* apapun dari *car*. *Speed* dan *lane* dari *car* akan tetap.
    - *ACCELERATE*: *Command* ini akan menambah *speed* dari *car* sesuai dengan tingkatan *speed* yang telah dipaparkan. *Car* hanya bisa mencapai *speed* tertentu bergantung pada seberapa banyak *damage* yang dimiliki oleh *car* (akan dibahas lebih lanjut pada bagian *damage*). *Lane* dari *car* akan tetap.

- *DECELERATE*: *Command* ini akan mengurangi *speed* dari *car* sesuai dengan tingkatan *speed* yang telah dipaparkan. *Car* dapat mencapai tingkatan *MINIMUM_SPEED* dimana *car* akan berhenti.
- *TURN_LEFT*: *Command* ini akan mengubah *lane* dari *car* menjadi *lane* pada sebelah kiri *lane* awal *car*. Jumlah *blocks* yang ditempuh selama satu ronde berkurang satu dikarenakan diperlukan perpindahan satu *block* pada awal pergantian *lane*. Ketika *car* berada pada *lane* paling kiri dan melakukan *command TURN_LEFT*, maka *car* akan menabrak sirkuit dan dikenai penalti *score*.
- *TURN_RIGHT*: *Command* ini akan mengubah *lane* dari *car* menjadi *lane* pada sebelah kanan *lane* awal *car*. Jumlah *blocks* yang ditempuh selama satu ronde berkurang satu dikarenakan diperlukan perpindahan satu *block* pada awal pergantian *lane*. Ketika *car* berada pada *lane* paling kanan dan melakukan *command TURN_RIGHT*, maka *car* akan menabrak sirkuit dan dikenai penalti *score*.
- *USE_BOOST*: *Command* ini akan menggunakan *power up Boost* (efek dari *Boost* sudah dipaparkan sebelumnya). Ketika pemain menggunakan *command USE_BOOST* tetapi tidak mempunyai *power up Boost*, maka *car* akan melakukan *NOTHING* dan akan dikenai penalti *score*.
- *USE_OIL*: *Command* ini akan menggunakan *power up Oil Item* (efek dari *Oil Item* sudah dipaparkan sebelumnya). Ketika pemain menggunakan *command USE_OIL* tetapi tidak mempunyai *power up Oil Item*, maka *car* akan melakukan *NOTHING* dan akan dikenai penalti *score*.
- *USE_LIZARD*: *Command* ini akan menggunakan *power up Lizard* (efek dari *Lizard* sudah dipaparkan sebelumnya). Ketika pemain menggunakan *command USE_LIZARD* tetapi tidak mempunyai *power up Lizard*, maka *car* akan melakukan *NOTHING* dan akan dikenai penalti *score*.
- *USE_TWEET <X> <Y>*: *Command* ini akan menggunakan *power up Tweet* pada *block* pada posisi *< X, Y >* (efek dari *Tweet* sudah dipaparkan sebelumnya). Ketika pemain menggunakan *command USE_TWEET* tetapi tidak mempunyai *power up Tweet*, maka *car* akan melakukan *NOTHING* dan akan dikenai penalti *score*.

- *USE_EMP*: *Command* ini akan menggunakan *power up EMP* (efek dari *EMP* sudah dipaparkan sebelumnya). Ketika pemain menggunakan *command USE_EMP* tetapi tidak mempunyai *power up EMP*, maka *car* akan melakukan *NOTHING* dan akan dikenai penalti *score*.
- *FIX*: *Command* ini akan memperbaiki *car* dengan menghilangkan 2 *damage* (akan dibahas lebih detail pada bagian *damage*). Ketika melakukan *command* ini, mobil akan tetap berada pada *block* yang sama untuk ronde tersebut.

5. Pada permainan ini, juga terdapat mekanisme *damage* yang diterima oleh *car*. *Damage* ini dapat bertambah bisa pemain melakukan interaksi dengan beberapa *block* khusus (*Flimsy Wall*, *Oil Spill*, dan *Mud*), terjadi *collision* antara dengan *cyber truck* maupun *car* pemain lain. Jumlah *damage* yang diterima oleh *car* per interaksi/*collision* dengan objek lain dapat dilihat dibawah ini:
   - *Mud*: 1 *damage*
   - *Oil Spill*: 1 *damage*
   - *Flimsy Wall*: 2 *damage*
   - *Cyber Truck*: 2 *damage*
   - *Car* pemain lain: 2 *damage* untuk setiap *car* yang *collision*

   Damage yang diterima oleh *car* berefek pada tingkatan *speed* maksimum yang dapat dicapai oleh *car* tersebut. Berikut ini adalah daftar *damage* yang diterima car serta tingkatan *speed* maksimum yang dapat dicapai oleh *car* tersebut:
   - 5 *damage*: *MINIMUM_SPEED*
   - 4 *damage*: *SPEED_STATE_1*
   - 3 *damage*: *SPEED_STATE_2*
   - 2 *damage*: *SPEED_STATE_3*
   - 1 *damage*: *MAXIMUM_SPEED*
   - 0 *damage*: *BOOST_SPEED*

   Untuk menghilangkan *damage*, pemain dapat melakukan *command FIX* untuk menghilangkan 2 poin *damage* pada setiap *command*-nya.
6. Pada permainan ini, terdapat pula mekanisme *scoring* dari setiap pemain. Mekanisme lengkap dari *scoring* adalah sebagai berikut:
   - Terjadi pengurangan 3 poin jika *car* pemain terkena *Mud block*.

- Terjadi pengurangan 4 poin jika *car* pemain terkena *Oil Spill block.*
- Terjadi penambahan 4 poin jika *car* berhasil mengambil *power up pickups*.
- Terjadi penambahan 4 poin jika pemain menggunakan *command USE_<power up>*.
- Terjadi pengurangan 5 poin jika pemain dikenakan penalti *score*. Beberapa *command* yang mengakibatkan dikenakannya penalti *score* adalah:
    - *TURN_LEFT* ketika berada pada *lane* paling kiri.
    - *TURN_RIGHT* ketika berada pada *lane* paling kanan.
    - *USE_<power up>* ketika tidak mempunyai *power up* yang dispesifikasi oleh *command*.
    - *Command* yang tidak terdapat pada daftar *command*.
    - *Command* belum diberikan sebelum waktu tenggat yang diberikan (algoritma *bot* terlalu lama).

7. Pemain akan memenangkan permainan bisa *car* pemain merupakan yang pertama kali sampai *finish line*. Jika ada dua *car* pemain yang tiba bersamaan pada *finish line*, maka pemenangnya ditentukan oleh *speed* pemain yang lebih besar. Jika dua *speed* pemain tersebut sama, maka pemenangnya ditentukan oleh peraih poin terbanyak.

Bab 2: Landasan Teori

2.1. Gambaran Algoritma *Greedy* secara Umum

Algoritma *greedy* merupakan sebuah algoritma yang mengimplementasikan konsep "*greedy*" dalam pendefinisian solusinya. Algoritma *greedy* biasanya digunakan untuk mencari solusi dari persoalan optimisasi (*optimization problem*) untuk memaksimumkan atau meminimumkan suatu parameter. Algoritma *greedy* dibentuk dengan memecah permasalahan dan membentuk solusi secara langkah per langkah (*step by step*). Pada setiap langkah tersebut, terdapat banyak langkah yang dapat dievaluasi. Pada algoritma *greedy*, pemrogram harus menentukan keputusan terbaik pada setiap langkahnya. Tetapi, di dalam algoritma *greedy* tidak diperbolehkan adanya *backtracking* (tidak dapat melihat mundur ke solusi sebelumnya untuk menentukan solusi langkah sekarang). Oleh karena itu, diharapkan pemrogram memilih solusi yang merupakan optimum lokal (*local optimum*) pada setiap langkahnya. Hal ini bertujuan bahwa langkah-langkah optimum lokal tersebut mengarah pada solusi dengan optimum global (*global optimum*).

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

- Solusi optimal dari persoalan dapat ditentukan dari solusi optimal subpersoalan tersebut.
- Pada setiap persoalan, terdapat suatu langkah yang dapat dilakukan dimana langkah tersebut menghasilkan solusi optimal pada subpersoalan tersebut. Langkah ini juga dapat disebut sebagai *greedy choice*.

Terdapat beberapa elemen/kompenen yang perlu didefinisikan di dalam algoritma *greedy*. Beberapa elemen/komponen algoritma *greedy* tersebut adalah:

- Himpunan kandidat (C): Berisi kandidat yang mungkin dipilih pada setiap langkahnya.
- Himpunan solusi (S): Berisi kandidat yang sudah terpilih sebagai solusi.
- Fungsi solusi (*solution function*): Menentukan apakah himpunan solusi yang dikumpulkan sudah memberikan solusi. (Domain: himpunan objek, Range: boolean).

- Fungsi seleksi (*selection function*): Memilih kandidat berdasarkan strategi *greedy* tertentu. Fungsi ini memiliki sifat heuristik (fungsi dirancang untuk mencari solusi optimum dengan mengabaikan apakah fungsi tersebut terbukti paling optimum secara matematis). (Domain: himpunan objek, Range: objek).
- Fungsi kelayakan (*feasibility function*): Memeriksa apakah kandidat yang terpilih oleh fungsi seleksi dapat dimasukkan ke dalam himpunan solusi. (Domain: himpunan objek, Range: boolean).
- Fungsi objektif (*objective function*): Memaksimumkan atau meminimumkan suatu parameter pada suatu persoalan. (Domain: himpunan objek, Range: himpunan objek).

Dengan menggunakan elemen/komponen di atas, algoritma *greedy* dapat didefinisikan sebagai berikut: "Algoritma *greedy* merupakan pencarian sebuah himpunan bagian S dari himpunan kandidat C, dimana S memenuhi kriteria kelayakan sebagai solusi paling optimum, yaitu S merupakan suatu himpunan solusi dan S dioptimisasi oleh fungsi objektif."

Skema umum algoritma *greedy* menggunakan *pseudocode* dengan pendefinisian elemen/komponennya adalah sebagai berikut:

```
function greedy(C : himpunan_kandidat) → himpunan_solusi
{ Mengembalikan solusi dari persoalan optimasi dengan algoritma greedy }
Deklarasi
  x : kandidat
  S : himpunan_solusi

Algoritma:
  S ← {}     { inisialisasi S dengan kosong }
  while (not SOLUSI(S)) and (C ≠ {} ) do
      x ← SELEKSI(C)    { pilih sebuah kandidat dari C}
      C ← C – {x}       { buang x dari C karena sudah dipilih }
      if LAYAK(S ∪ {x}) then     { x memenuhi kelayakan untuk dimasukkan ke dalam himpunan solusi }
          S ← S ∪ {x}    { masukkan x ke dalam himpunan solusi }
      endif
  endwhile
  {SOLUSI(S) or C = {} }

  if SOLUSI(S) then    { solusi sudah lengkap }
    return S
  else
    write('tidak ada solusi')
  endif
```

Gambar 2.1. *Pseudocode* algoritma *greedy*

Pada akhir tiap iterasi, solusi yang terbentuk adalah optimum lokal, dan paka akhir *loop while-do* akan ditemukan optimum global, namun optimum global yang ditemukan ini

belum tentu merupakan solusi yang terbaik karena algoritma *greedy* tidak melakukan operasi secara menyeluruh kepada semua kemungkinan yang ada.

Kesimpulannya, algoritma *greedy* dapat digunakan untuk masalah yang hanya membutuhkan solusi hampiran dan tidak memerlukan solusi terbaik mutlak. Solusi ini terkadang lebih baik daripada algoritma yang menghasilkan solusi eksak dengan kebutuhan waktu yang eksponensial. Untuk contoh dari hal ini kita dapat melihat *Traveling Salesman Problem*, dimana penggunaan algoritma *greedy* akan jauh lebih cepat dibandingkan dengan penggunaan *brute force*, walaupun solusi yang ditemukan biasanya hanya hampiran dari solusi optimal.

2.2. Garis Besar Cara Kerja Bot Permainan Overdrive

Secara garis besar, cara kerja *bot* dari permainan Overdrive ini adalah pertama-tama *bot* akan mengambil data yang tersedia pada *state files*. Data-data yang terdapat pada *state files* adalah sebagai berikut:

- *currentRound*: Nomor ronde saat ini.
- *maxRounds*: Jumlah maksimal ronde yang dapat dimainkan pada permainan ini.
- *player*: Detail kondisi *car* pemain.
    - *id*: ID dari *car* pemain.
    - *position*: Posisi dari *car* pemain.
        - *x*: Posisi *block* dari *car* pemain (dihitung dari *start block*).
        - *y*: Posisi *lane* dari *car* pemain (dihitung dari *lane* terkiri).
    - *speed*: Kecepatan dari *car* pemain.
    - *state*: Keadaan pemain setelah *command* terakhir dieksekusi.
    - *powerups*: *List* yang berisi *power up* yang dimiliki oleh pemain.
    - *boosting*: Menunjukkan apakah *car* pemain sedang mengalami *boost*.
    - *boostCounter*: Menunjukkan berapa ronde tersisa dari *car* pemain mengalami *boost*.
    - *damage*: Menunjukkan jumlah *damage* yang diterima oleh *car* pemain.
- *opponent*: Detail dari kondisi *car* lawan.
    - *id*: ID dari *car* lawan.
    - *position*: Posisi dari *car* lawan.

        - *x*: Posisi *block* dari *car* lawan (dihitung dari *start block*).
        - *y*: Posisi *lane* dari *car* lawan (dihitung dari *lane* terkiri).
    - *speed*: Kecepatan dari *car* lawan.
- *worldMap*: *Array of objects* yang menjelaskan keadaan setiap *blocks* pada *map*. Setiap *object* pada *worldMap* akan dijelaskan sebagai berikut:
    - *position*: Posisi dari *block.*
        - *x*: Koordinat x dari *block* ($1 \leq x \leq$ *finish line*).
        - *y*: Koordinat y dari *block* ($1 \leq y \leq 4$).
    - *surfaceObject*: Mendefinisikan terdapat objek apa pada *block* ini.
    - *occupiedByPlayerId*: ID dari *car* pemain pada *block* ini. Jika tidak terdapat *car* maka *occupiedByPlayerId* akan bernilai 0.
    - *isOccupiedByCyberTruck*: Mengembalikan *true* jika terdapat *cyber truck* dan *false* jika tidak.

Setelah *bot* mengambil data-data di atas, maka *bot* dapat melakukan analisis menggunakan algoritma yang telah dituliskan sebelumnya untuk mencari *command* paling tepat untuk dieksekusi pada ronde selanjutnya. Algoritma yang dituliskannya tersebut berbentuk fungsi objektif dari algoritma *greedy*. Perlu ditekankan pula, untuk setiap keadaan pada data, *bot* harus mengirimkan *command* yang bersesuaian. Jika *bot* tidak mengirimkan *command* kepada *game engine* sebelum waktu yang ditentukan, maka akan ada penalti *score*.

2.3. Implementasi Algoritma *Greedy* ke dalam Bot Permainan Overdrive

Terdapat beberapa algoritma yang dapat digunakan sebagai algoritma *bot*. Beberapa algoritma yang dapat digunakan adalah diantaranya algoritma *brute force*, algoritma *greedy*, algoritma *tree traversal*, dan lain-lain. Tetapi, semua algoritma tersebut memiliki kelebihan dan kekurangannya masing-masing. Pada algoritma *brute force*, dibutuhkan waktu dan *resource* yang banyak untuk mengetes dan menilai setiap kemungkinan *command* yang dapat terjadi, meskipun hasil paling optimum yang ditemukan merupakan optimum global pada keadaan tersebut. Kemudian, pada algoritma *greedy*, tidak diperlukan waktu atau *resource* yang banyak. Tetapi, dibutuhkan teknik heuristik yang didefinisikan oleh logika dan pola pikir masing-masing pemrogram. Teknik heuristik tersebut mungkin

belum dapat menghasilkan solusi optimum global dan walaupun solusi tersebut merupakan solusi optimum global, solusi tersebut susah dibuktikan kebenarannya secara matematis. Algoritma berikutnya yang dapat digunakan adalah algoritma *tree traversal*. Algoritma ini umum digunakan pada *bot game*, dikarenakan pada algoritma ini terdapat pengecekkan beberapa kasus tertentu untuk menentukan langkah selanjutnya yang diambil. Tetapi, algoritma ini tidak seintuitif algoritma *brute force* maupun algoritma *greedy* bagi pemrogram.

Oleh karena itu, penulis menggunakan algoritma *greedy* sebagai algoritma pembentukan *bot* dikarenakan cukup intuitif serta tidak memerlukan waktu atau *resource* yang terlalu banyak. Selain itu, terdapat cukup banyak kemungkinan solusi *greedy* yang dapat dieksplorasi oleh penulis. Meskipun terdapat peluang bahwa penulis tidak dapat menciptakan algoritma *greedy* yang menghasilkan solusi optimum global, tetapi setidaknya penulis dapat selalu mencapai solusi optimum lokal yang nilainya mendekati solusi optimum global.

2.4. Garis Besar *Game Engine* Permainan Overdrive

Game engine adalah suatu *framework* perangkat lunak yang didesain sebagai kumpulan *tools* dan *features* yang digunakan untuk membantu *development* dari suatu *game*. Umumnya di dalam suatu *game engine* terdapat beberapa *core development tools*, diantaranya adalah *rendering engine* (untuk melakukan *render* pada grafik 2D atau 3D), *physics engine* atau *collision engine* (untuk mengatur bagaimana aturan fisika pada *game*), *sound engine*, *scripting language*, *animation engine*, *artificial intelligence*, *networking engine, streaming engine*, *memory management*, *threading support*, *localization support*, *scene graph*, serta *video support*. Pada permainan ini tidak semua *tools* disediakan oleh pemrogram, hanya beberapa *tools* esensial seperti *rendering engine for command prompt*, *collisions engine*, dan *scripting language* saja yang terdapat pada repository. Untuk beberapa *tools* tambahan, seperti 2D *rendering engine* dapat diunduh dari *third party software*.

Pada permainan Overdrive ini, *game engine* yang digunakan dibuat menggunakan bahasa Scala dan sudah tersedia sebelumnya pada *repository* yang diunggah sebelumnya oleh Entelect (Entelect merupakan penyelenggara dari *challenge* ini). *Game engine* yang

diberikan pada *starter-pack* sudah berbentuk *file* .jar, jadi kita tidak perlu melakukan *compile* dan *build* ulang projek secara manual. *Prerequisites* untuk menjalankan *game engine* adalah memiliki JDK (Java Development Kit) pada mesin eksekusi untuk menjalankan *game engine*. Hal ini ditujukan untuk memroses *file* .jar, melakukan kompilasi *file* dengan bahasa Java, serta mengeksekusinya. Jika *prerequisites* telah dipenuhi, untuk menjalankan *game engine* pengguna dapat menjalankan *file* run.bat untuk pengguna dengan OS Windows atau menjalankan perintah *shell script* pada *file* Makefile dengan mengetikkan perintah "make run" pada command prompt untuk pengguna dengan OS Linux atau macOS.

Selain file *game engine*, terdapat beberapa *file* dan *folder* yang perlu ditambahkan pada *starter-pack* oleh pemrogram. Beberapa *file* dan *folder* yang wajib ada agar *game engine* dapat dijalankan adalah diantaranya:

- game-engine.jar dan game-engine-jar-with-dependencies.jar: Berisi *source code* serta *dependecies* dari *game engine*.
- game-config.json: Berisi *rules* yang diikuti oleh *game engine* ketika mengeksekusi permainan.
- game-runner-config.json: Berisi informasi mengenai *directory* dari *file bot* serta parameter-parameter lainnya yang dibutuhkan oleh *game engine*.
- match-logs: Berisi *folder* hasil *match* setiap permainan dengan hasil setiap *round* dicatat dalam *folder*-nya masing-masing. Di dalam *folder* tersebut dicantumkan *file* json serta *file* txt yang merepresentasikan keadaan *game state* dalam bentuk *file* yang bernama GlobalState.json.
- bot-player-a dan bot-player-b: Berisi *folder* yang di dalamnya terdapat hasil *compile* dan *install* dari Maven *project* dari *bot* yang telah dibuat sebelumnya oleh pemrogram. Terdapat pula *file* bot.json yang menspesifikasi nama *bot*, bahasa pemrograman yang digunakan untuk membuat *bot*, nama *file bot*, serta *folder* yang berisi hasil *compile* dan *install* dari Maven *project*.
- Makefile dan run.bat: Berisi *file* dengan *scripting language* yang digunakan untuk menjalankan *game engine* pada OS. Makefile digunakan untuk OS Linux dan macOS serta run.bat digunakan untuk OS Windows.

Agar program dapat berinteraksi dengan *bot* yang pemrogram bangun, pemrogram dapat menggunakan *file* dengan format json untuk mengubah parameter-parameter yang dibutuhkan *game engine*, seperti *directory folder* dari *bot*, *directory file* dari *game engine*, *maximum runtime* dari setiap *round*, dan lain-lain. Beberapa *file* dengan format json yang dapat diubah oleh pemrogram adalah *file* bot.json yang terdapat pada setiap *folder bot* serta *file* game-runner-config.json yang terdapat pada *starter-pack*. Sebaiknya, pemrogram tidak mengubah-ubah *game rule* yang terdapat pada *file* game-config.json agar tidak terjadi konflik ketika *game engine* dijalankan. Berikut ini adalah penjelasan konfigurasi *file* bot.json serta game-runner-config.json.

- bot.json:
  - author: Berisi nama penulis/pemrogram algoritma *bot*.
  - email: Berisi email penulis/pemrogram algoritma *bot*.
  - nickName: Berisi *nickname* dari *bot*.
  - botLocation: Berisi *relative path* tempat hasil *compile* dan *install* menggunakan *built tools* Maven.
  - botFilename: Berisi nama *file* jar dari *bot* yang ingin dijalankan.
  - botLanguage: Berisi nama bahasa pemrograman yang digunakan untuk membuat algoritma *bot*.
- game-runner-config.json:
  - round-state-output-location: Berisi *relative path* tempat hasil *match* setiap permainan dituliskan.
  - game-config-file-location: Berisi *relative path* tempat *file* game-config.json berada. *File* tersebut mengandung semua *rules* yang diikuti oleh *game engine*.
  - game-engine-jar: Berisi *relative path* tempat *file* game-engine.jar berada. *File* tersebut merupakan *game engine* dari permainan.
  - verbose-mode: Berisi nilai *boolean* yang menyatakan apakah pemrogram ingin mencatat *match-logs* dari permainan yang dijalankan. Default dari verbose-mode adalah *true*.
  - max-runtime-ms: Berisi nilai *integer* yang menyatakan waktu maksimum (dalam milisekon) setiap ronde berlangsung.

- player-a dan player-b: Berisi *relative path* tempat *file* bot.json. Jika pemrogram ingin menggunakan console sebagai *input*, parameter ini dapat diisi dengan *string* "console" sebagai *path*.
- player-a-id dan player-b-id: Berisi nilai *integer* unik yang menandakan ID dari pemain.
- max-request-retries: Berisi nilai *integer* yang menyatakan jumlah maksimal percobaan melakukan *request* melalui *network*.
- request-time-ms: Berisi nilai *integer* yang menyatakan waktu maksimum (dalam milisekon) setiap *request* berlangsung.
- is-tournament-mode: Berisi nilai *boolean* yang menyatakan apakah mode turnamen diaktifkan.
- tournament: Berisi *object* dengan data parameter yang dibutuhkan untuk menjalankan turnamen.

Pada awalnya, *starter bot* yang didefinisikan oleh Entellect belumlah lengkap, masih terdapat *file* Java yang mengandung fitur-fitur penting belum ditambahkan ke dalam folder *src* dari *starter bot*. Oleh karena itu, penulis harus terlebih dahulu mendefinisikan beberapa *file* Java dan melakukan *import* seluruh *file* tersebut ke dalam *file* bot.java. Beberapa contoh *file* yang belum lengkap pada *starter bot* adalah LizardCommand.java dan TweetCommand.java belum terdefinisi sama sekali serta Lane.java, PowerUp.java, State.java, dan Terrain.java belum memiliki definisi yang lengkap.

Sebelum penulis mendefinisikan algoritma pada *file* bot.java, penulis terlebih dahulu melengkapkan pendefinisian *file* diatas. Pada *file* LizardCommand.java dan TweetCommand.java, penulis harus terlebih dahulu mendefinisikan *class* LizardCommand dan TweetCommand dengan meimplementasikan *interface* Command.java. Selain itu, pada *file* Lane.java, PowerUp.java, State.java, dan Terrain.java, penulis harus menambahkan atribut serta SerializedName baru agar *bot* dapat membaca data-data pada *file game state*. SerializedName *annotation* merupakan salah satu cara program membaca *file* dengan format json. Pada program *starter bot*, SerializedName *annotation* digunakan untuk membaca file GlobalState.json yang terdapat pada *match logs*. Hasil pembacaan tersebut kemudian disimpan pada atribut pada kelas yang berkorespondensi. Oleh karena

itu, program *starter bot* yang dibuat oleh penulis dapat mengakses data yang disediakan oleh *map*.

Setelah algoritma *greedy* para *starter bot* telah dibuat, maka harus dilakukan kompilasi agar perubahan tersebut dapat diaplikasikan pada *file* jar dari *starter bot*. Pada program ini, telah disediakan *build tools* dari Apache Maven Project. Intellij IDEA telah menyediakan *build tools* Maven sehingga penulis tidak perlu menambahkan *extension* apapun di dalam IDEnya. Terdapat beberapa *command* dari *build lifecycle* yang dapat dijalankan oleh Maven, diantaranya adalah berikut ini:

- *validate*: Melakukan validasi apakah seluruh data dan informasi yang dibutuhkan untuk melakukan *build project* sudah terpenuhi.
- *compile*: Melakukan kompilasi terhadap *source code* yang telah dibuat oleh pemrogram.
- *test*: Melakukan *testing* menggunakan *source code* hasil kompilasi terhadap suatu *framework* yang sudah ditentukan sebelumnya.
- *package*: Melakukan *packaging* dari hasil kompilasi *source code* menjadi suatu *file* dengan format jar.
- *verify*: Melakukan tes verifikasi integrasi terhadap *file* jar yang telah dibangun (menentukan apakah *project* sudah siap untuk di-*install*).
- *install*: Melakukan instalasi *project* ke dalam *local repository* beserta *dependencies project* tersebut.
- *deploy*: Menyudahi *build environment* yang telah dibuat sebelumnya dan melakukan *copy project* ke dalam *remote repositories*.

Pada program Overdrive ini, untuk melakukan kompilasi perubahan *source code* dan kemudian mengubahnya ke dalam bentuk *file* jar, pemrogram hanya perlu menjalankan perintah *compile* dan *install* saja. Pemrogram tidak perlu melakukan semua perintah pada *build lifecycle*.

Bab 3: Aplikasi Strategi *Greedy*

3.1. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Bot Permainan Overdrive

3.1.1. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan *General Bot*

Dalam permainan Overdrive, tujuan setiap *bot* pemain berusaha untuk menjadi yang pertama melewati *finish line*. Terdapat banyak cara untuk meraih hal tersebut, seperti berusaha mencapai kecepatan maksimum pada setiap waktu, membuat *bot* lawan tidak bisa bergerak/bergerak lebih lamban, dan lain-lain.

Nama Elemen/Komponen	Definisi Elemen/Komponen
Himpunan kandidat	Seluruh permutasi dari *command* yang telah dipaparkan sebelumnya untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang dapat mengalahkan *bot* lawannya.
Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut dapat membuat *bot* melewati *finish line*.
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti fungsi-fungsi seleksi subbagian dekomposisi untuk setiap kasus pada *game state*. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid telah dipaparkan pada bab sebelumnya.

Fungsi objektif	Mencari permutasi dari *command* yang membuat *bot* pemain memenangkan permainan dengan jumlah ronde paling sedikit serta pada saat yang bersamaan membuat *bot* lawan tidak semakin mendekati *finish line*.

3.1.2. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan *Fix* dan *Damage Mechanism*

Salah satu subpermasalahan dalam permainan Overdrive adalah adanya *damage mechanism*. *Car* pemain dapat menerima *damage* jika *car* tersebut melakukan *collisions* dengan beberapa *block* khusus (*Oil Spill, Cyber Truck, Flimsy Wall,* dan *Mud*) atau *car* lawannya. Jumlah *damage* yang diterima pemain tersebut mempengaruhi *speed* maksimum dari *car*. Untuk menghilangkan *damage*, pemain dapat melakukan *command FIX* untuk menghilangkan 2 poin *damage* dengan bayaran *car* tidak bergerak selama ronde berikutnya.

Nama Elemen/Komponen	**Definisi Elemen/Komponen**
Himpunan kandidat	Permutasi dari *command FIX* atau tidak melakukan *command FIX* untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang membuat *car* memiliki *maximum speed* yang optimum serta tidak menghambat laju *car* dengan *command FIX*.
Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut membuat *bot* berjalan seperti seharusnya (tidak terdapat keadaan *bot* melakukan *command FIX* tidak pada tempatnya).
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti strategi *fix* dan *damage mechanism* paling optimum untuk

	setiap ronde yang berlangsung. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid pada strategi ini adalah *bot* melakukan *command FIX* atau *bot* tidak melakukan *command FIX*.
Fungsi objektif	Mencari permutasi dari *command* yang membuat *bot* pemain memiliki *maximum speed* seperti seharusnya tanpa menghambat laju mobil dengan dilakukannya *command FIX*.

3.1.3. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan *Speed* dan *Boost Power Up*

Salah satu subpermasalahan yang paling penting dan mendasar pada permainan Overdrive adalah subpermasalahan *speed mechanism*. Tujuan utama dari permainan Overdrive adalah mendahului *bot* lawan untuk mencapai *finish line* terlebih dahulu. Oleh karena itu, dibutuhkan algoritma *greedy* yang menangani *speed mechanism* secara efektif dan efisien.

Nama Elemen/Komponen	**Definisi Elemen/Komponen**
Himpunan kandidat	Permutasi dari *command ACCELERATE*, *command USE_BOOST*, atau tidak melakukan kedua *command* tersebut untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang membuat *car* memiliki rata-rata *speed* atau jarak tempuh *block* per satuan waktu yang paling optimum dengan penggunaan *boost power up* paling sedikit.

Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut membuat *bot* berjalan seperti seharusnya (tidak terdapat keadaan *bot* melakukan *command BOOST* ketika tidak mempunyai *boost power up* pada *inventory* atau melakukan *command ACCELERATE* ketika sudah mencapai *speed* maksimum).
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti strategi *speed* dan *boost power up mechanism* paling optimum untuk setiap ronde yang berlangsung. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid pada strategi ini adalah *bot* melakukan *command ACCELERATE*, *command USE_BOOST*, atau *bot* tidak melakukan kedua *command* tersebut.
Fungsi objektif	Mencari permutasi dari *command* yang membuat *bot* pemain memiliki rata-rata *speed* paling besar dengan penggunaan *boost power up* paling sedikit.

3.1.4. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan Menghindari *Obstacle* dan *Lizard Power Up*

Kemudian, subpermasalahan selanjutnya yang dapat diselesaikan oleh algoritma *greedy* adalah subpermasalahan penghindaran suatu *obstacle* dengan cara perpindahan *lane* atau dengan penggunaan *lizard power up*. Hal ini berguna pula

untuk menghindari *damage* serta pengurangan *speed* akibat *collisions* antara *car* pemain dengan *block* khusus (*Oil spill*, *Cyber truck*, *Flimsy Wall*, dan *Mud*) ataupun *car* lawan. *Lizard power up* sendiri memiliki efek *car* akan melompati semua *power up pick-ups*, *obstacles*, serta *car* lawan selama ronde tersebut berlangsung dengan *car* pemain masih terdapat pada *lane* yang sama.

Nama Elemen/Komponen	**Definisi Elemen/Komponen**
Himpunan kandidat	Permutasi dari *command TURN_LEFT*, *command TURN RIGHT*, *command USE_LIZARD* atau tidak melakukan ketiga *command* tersebut untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang membuat jumlah *damage* yang diterima *car* pemain paling sedikit dengan penggunaan *lizard power up* paling sedikit pula.
Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut membuat *bot* berjalan seperti seharusnya (tidak terdapat keadaan *bot* melakukan *command USE_LIZARD* ketika tidak mempunyai *lizard power up* pada *inventory* atau melakukan *command TURN_LEFT* maupun *command TURN_RIGHT* ketika tidak terdapat *obstacle* di depan *lane car* pemain).
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti strategi penghindaran *obstacle* dan *lizard power up mechanism* paling optimum untuk setiap ronde yang berlangsung. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung

	(tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid pada strategi ini adalah *bot* melakukan *command TURN_RIGHT*, *command TURN_LEFT*, *command USE_LIZARD*, atau *bot* tidak melakukan ketiga *command* tersebut.
Fungsi objektif	Mencari permutasi dari *command* yang membuat *bot* pemain memiliki jumlah *damage* yang diterima paling sedikit dengan penggunaan *lizard power up* paling sedikit pula.

3.1.5. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan Pengambilan *Power Up Pick-Ups*

Selanjutnya, subpermasalahan yang dapat diselesaikan dengan algoritma *greedy* adalah subpermasalahan bagaimana cara mendapatkan *power up pick-ups* paling banyak dengan cara perpindahan *lane*. Meskipun sekilas subpermasalahan ini tidak seurgensi subpermasalahan lain seperti penghindaran *obstacle* ataupun *speed mechanism*, tetapi *bot* pemain juga membutuhkan bantuan *power up* untuk memenangkan permainan. Selain itu, bila *bot* pemain tidak mengambil *power up pick-ups*, maka *bot* lawan akan mendapatkan peluang untuk mengambil *power up pick-ups* tersebut dan menggunakannya untuk mengalahkan *bot* pemain.

Nama Elemen/Komponen	**Definisi Elemen/Komponen**
Himpunan kandidat	Permutasi dari *command TURN_LEFT*, *command TURN_RIGHT*, atau tidak melakukan kedua *command* tersebut untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang membuat jumlah *power up pick-ups* yang diambil oleh *car* pemain paling banyak.

Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut membuat *bot* berjalan seperti seharusnya (tidak terdapat keadaan *bot* melakukan *command TURN_LEFT* maupun *command TURN_RIGHT* ketika tidak terdapat *power up pick-ups* di *lane* baru *car* pemain).
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti strategi pengambilan *power up pick-ups* paling optimum untuk setiap ronde yang berlangsung. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid pada strategi ini adalah *bot* melakukan *command TURN_RIGHT*, *command TURN_LEFT*, atau *bot* tidak melakukan kedua *command* tersebut.
Fungsi objektif	Mencari permutasi dari *command* yang membuat *bot* pemain memiliki jumlah *power up pick-ups* paling banyak.

3.1.6. Pemetaan Elemen/Komponen Algoritma *Greedy* pada Permasalahan Penggunaan *Offensive Power Up*

Terakhir, subpermasalahan yang dapat diselesaikan dengan algoritam *greedy* adalah subpermasalahan penggunaan *power up* yang bersifat “ofensif” secara efektif dan efisien. Harapannya, setiap *command USE_<POWER_UP>* yang dieksekusi dapat mengenai *car* lawan dan mengakibatkan efek paling kentara. Hal

ini dilakukan agar *power up pick-ups* yang telah diambil sebelumnya tidak terbuang sia-sia serta tidak terjadi penumpukan *power up* pada *inventory*.

Nama Elemen/Komponen	Definisi Elemen/Komponen
Himpunan kandidat	Permutasi dari *command* USE_OIL, *command USE_EMP*, *command USE_TWEET <X> <Y>*, atau tidak melakukan ketiga *command* tersebut untuk setiap rondenya.
Himpunan solusi	Kemungkinan permutasi dari *command* yang membuat jumlah penggunaan *power up* dari *car* pemain mengenai *car* lawan paling banyak serta memiliki efek paling efektif.
Fungsi solusi	Melakukan pengecekkan apakah permutasi dari *command* tersebut membuat *bot* berjalan seperti seharusnya (tidak terdapat keadaan *bot* melakukan *command* penggunaan *power up* yang tidak memiliki kemungkinan sama sekali untuk mengenai *car* lawan).
Fungsi seleksi	Memilih *command* berdasarkan data keadaan *game state* saat tersebut serta fungsi heuristik tingkat prioritas *command* yang harus diikuti. Bentuk fungsi heuristik tersebut mengikuti strategi penggunaan *power up* paling optimum untuk setiap ronde yang berlangsung. Tingkat prioritas tersebut bersifat statik selama permainan berlangsung (tidak ada perubahan kondisi pada fungsi heuristik).
Fungsi kelayakan	Memeriksa apakah *command* yang dituliskan oleh *bot* merupakan *command* yang valid. Daftar *command* yang valid pada strategi ini adalah *bot* melakukan *command USE_OIL*, *command*

	USE_EMP, *USE_TWEET* <*X*> <*Y*>, atau *bot* tidak melakukan ketiga *command* tersebut.
Fungsi objektif	Mencari permutasi dari *command* yang membuat jumlah penggunaan *power up bot* pemain dan mengenai *bot* lawan secara efektif paling banyak.

3.2. Eksplorasi Alternatif Solusi Algoritma *Greedy* pada Bot Permainan Overdrive

Terdapat banyak sekali alternatif solusi algoritma *greedy* yang dapat diimplementasikan pada *bot* permainan Overdrive. Hal ini dikarenakan terdapat banyak elemen dari *game engine* yang dapat diakses oleh *bot* dan kemudian diubah *state* pada elemen tersebut. Selain itu, elemen-elemen tersebut saling berinteraksi dengan elemen lainnya sehingga jika terdapat perubahan pada suatu *state* elemen, terdapat peluang bahwa *state* lainnya berubah pula. Sebagai contoh, elemen *damage* suatu *car* pemain di dalam *damage mechanism* mempengaruhi jalannya *speed mechanism*, dikarenakan *damage mechanism* mengatur *maximum speed* dari *speed mechanism*. Oleh karena itu, strategi algoritma *greedy* yang disusun oleh penulis memiliki interaksi dengan strategi algoritma *greedy* pada bidang yang berbeda. Berikut ini merupakan beberapa alternatif strategi algoritma *greedy* yang dapat diimplementasikan pada *bot* pemain:

3.2.1. Strategi Heuristik *Fix* dan *Damage Mechanism*

Seperti yang telah diketahui sebelumnya, *bot* pada permainan membutuhkan adanya *fix mechanism* dikarenakan adanya *damage* yang dapat dikenai pada *bot*. *Damage* tersebut berpengaruh pada *maximum speed* yang dapat dicapai oleh *car* pemain. Terdapat beberapa strategi heuristik yang dapat diimplementasikan dalam bentuk algoritma *greedy* untuk melakukan *handling damage*.

Strategi pertama yang dapat diimplementasikan adalah melakukan *command FIX* ketika benar-benar dibutuhkan, yaitu ketika *damage* yang dimiliki oleh *car* pemain melebihi 5 poin *damage*. Ketika *damage* yang diterima *car* pemain melebihi 5 poin *damage*, maka *car* pemain tidak bisa lagi bergerak. Strategi ini memastikan *car* pemain tidak melakukan *command fix* jika *car* pemain masih bisa bergerak. Tetapi, hal tersebut belum tentu berdampak baik terhadap performa *overall* dari *bot*. Hal ini dikarenakan meskipun *car* menerima *damage* kurang dari

5 poin, *maximum speed* yang dapat dicapai oleh *car* pemain akan menurun sesuai tingkatan yang telah dijelaskan pada bab 1 dan bab 2. Oleh karena itu, dibutuhkan strategi baru yang lebih memperhatikan *maximum speed*.

Strategi kedua yang dapat diimplementasikan adalah melakukan langsung melakukan *command FIX* pada ronde setelah ronde dimana *car* pemain menerima *damage*. Hal ini memastikan bahwa *maximum speed* yang dimiliki oleh *car* pemain selalu terdapat pada tingkatan *BOOST_SPEED*. Sekilas, strategi ini terlihat lebih baik dari strategi pertama. Secara *overall* mungkin strategi *greedy* ini merupakan strategi yang paling baik, tetapi menurut penulis strategi ini masih menyisakan ruang untuk melakukan *improvement*. *Bot* tidak perlu langsung diperbaiki bila *bot* tersebut belum memerlukan perbaikan. Strategi yang terbaik menurut penulis merupakan strategi yang berada diantara spektrum strategi pertama serta strategi kedua.

Strategi ketiga yang dapat diimplementasikan adalah melakukan *command FIX* ketika *car* sudah mencapai tingkatan *maximum speed* sesuai dengan *damage* yang diterima (relasi ini dapat dilihat pada Bab 1). Strategi ketiga ini menggabungkan kedua strategi yang telah dipaparkan sebelumnya, dimana *bot* perlu melakukan *command FIX* meskipun masih bisa bergerak dan hanya jika *bot* telah mencapai keadaan dimana *speed* dari *bot* tidak dapat bertambah kecuali dilakukan *command FIX*. Menurut penulis, strategi ini merupakan strategi paling efisien untuk menyelesaikan subpermasalahan *damage mechanism* pada permainan.

3.2.2. Strategi Heuristik *Speed* dan *Boost Power Up*

Salah satu strategi heuristik yang paling penting dalam permainan ini adalah adanya algoritma *greedy* yang menangani *speed mechanism* serta penggunakan *boost power up*. Strategi ini merupakan strategi inti dari *bot* dikarenakan hampir seluruh strategi heuristik lainnya berinteraksi dengan strategi *speed mechanism*. Oleh karena itu, strategi heuristik yang dikembangkan pada bagian ini akan bersifat lebih mendetail dan terdapat lebih banyak parameter dari strategi heuristik lainnya. Terdapat beberapa strategi heuristik yang dapat diimplementasikan dalam bentuk algoritma *greedy* untuk melakukan *handling speed mechanism*.

Strategi pertama yang dapat diimplementasikan adalah pada setiap kemungkinan, *bot* akan berusaha untuk mempercepat *speed* dari *car* pemain dengan cara menggunakan *command ACCELERATE* atau menggunakan *boost power up*. Strategi ini merupakan strategi yang paling *straightforward* dalam memaksimalkan fungsi objektif dimana *bot* harus berusaha memperbesar rata-rata dari *speed* pemain. Akan tetapi, strategi tersebut akan susah diintegrasikan dengan strategi lainnya. Hal tersebut dikarenakan kondisi untuk melakukan *command ACCELERATE* akan terlalu mudah terpenuhi, sehingga penggunaan *command* lainnya akan lebih mudah terabaikan oleh algoritma *greedy* bot. Dibutuhkan strategi heuristik baru dimana di dalam strategi baru tersebut diberi batasan-batasan keadaan untuk melakukan *command ACCELERATE* maupun penggunaan *boost power up*.

Strategi kedua yang dapat diimplementasikan adalah pada setiap kemungkinan dengan batasan-batasan tertentu, *bot* dapat melakukan *command ACCELERATE* atau menggunakan *boost power up*. Batasan-batasan tersebut haruslah didefinisikan secara heuristik oleh pemrogram. Beberapa contoh dari batasan yang didefinisikan secara heuristik oleh penulis adalah sebagai berikut:

- Jika pemain mempunyai *boost power up*, *damage* yang dimiliki oleh *car* pemain bernilai 0, serta pemain tidak sedang menggunakan efek *boost power up*, maka gunakanlah *command USE_BOOST* untuk mengaktifkan efek *boost power up*. Jika ternyata terdapat *damage* yang dimiliki oleh *car* pemain, maka terlebih dahulu pemain harus melakukan *command FIX* pada *car*. Hal tersebut bertujuan untuk memaksimalkan *boost power up* yang diaktifkan oleh pemain (jika terdapat poin *damage* pada *car*, maka tingkatan *maximum speed* dari *car* tidak akan mencapai 15 *blocks* per ronde).
- Jika *speed* pemain belum mencapai maksimum relatif dengan *maximum speed* yang dapat dicapai dikarenakan adanya *damage mechanism*, maka *bot* akan melakukan *command ACCELERATE*. Jika ternyata *speed* pemain telah mencapai *maximum speed*, maka program akan memutuskan bahwa *car* pemain telah mencapai keadaan dimana *speed* dari *car* sudah optimum.

Menurut penulis, strategi kedua merupakan strategi yang lebih superior dibandingkan strategi pertama. Pada strategi kedua, diusahakan penggunaan *command ACCELERATE* serta penggunaan *boost power up* menghasilkan capaian paling optimum dan tidak sia-sia.

3.2.3. Strategi Heuristik Menghindari *Obstacle* dan *Lizard Power Up*

Strategi heuristik yang tidak kalah penting dari strategi *speed mechanism* pada *bot* permainan ini adalah strategi untuk menghindari *obstacle* dengan mengubah *lane* atau dengan menggunakan *lizard power up*. Strategi ini mengatur *command* mana yang paling tepat untuk digunakan bila ternyata hasil pembacaan *game state* terdapat *obstacle* atau *car* lawan pada *lane* tempat *car* berada. Mirip dengan strategi *speed mechanism*, terdapat banyak interaksi antara strategi ini dengan strategi lainnya. Oleh karena itu, dibutuhkan strategi heuristik yang mendetail dan memperhitungkan banyak kasus kemungkinan pada *game state*. Terdapat beberapa strategi heuristik yang dapat diimplementasikan dalam bentuk algoritma *greedy* untuk melakukan *handling* penghindaran *obstacle*.

Strategi pertama yang dapat diimplementasikan adalah ketika terdapat *obstacle* pada *lane* pemain, maka *bot* akan menghindar secara naif ke *lane* sebelahnya (*lane* kiri atau *lane* kanan dari *car* pemain jika memungkinkan) atau menggunakan *lizard power up* jika ternyata *car* pemain tidak dapat menghindari dengan penggantian *lane*. Strategi ini sebenarnya sudah cukup memenuhi syarat untuk dijadikan solusi. Pemrogram hanya perlu menambahkan beberapa detail dalam program dengan memasukkan data-data yang terdapat pada *game state* sebagai parameter dalam algoritma *greedy* agar pemilihan *lane* baru atau penggunaan *lizard power up* lebih dapat terstruktur.

Strategi kedua yang dapat diimplementasikan adalah ketika *car* pemain mendeteksi adanya *obstacle* pada *lane* tempat *car* pemain berada, maka *bot* akan mencari *lane* paling optimum untuk dilalui. Perhitungan *lane* paling optimum tersebut dapat diinferensikan dari data *game state* yang telah dibaca oleh program. Algoritma pencarian *lane* paling optimum dapat dicari menggunakan implementasi algoritma *greedy*. Beberapa contoh dari penanganan kasus pada *obstacle avoidance mechanism* adalah sebagai berikut:

- Ketika *car* pemain terdapat pada *lane* terkiri:
    - Jika proyeksi *damage* yang diterima pada *lane* pemain lebih kecil dari proyeksi *damage* yang diterima pada *lane* sebelah kanan:
        - Jika pemain memiliki *lizard power up*, maka *bot* akan menggunakan *command USE_LIZARD*.
        - Jika pemain tidak memiliki *lizard power up*, maka *bot* akan menggunakan *command ACCELERATE*.
    - Jika proyeksi *damage* yang diterima pada *lane* sebelah kanan lebih kecil dari proyeksi *damage* yang diterima pada *lane* pemain, maka *bot* akan menggunakan *command TURN_RIGHT*
    - Jika proyeksi *damage* yang diterima pada *lane* pemain sama dengan proyeksi *damage* yang diterima pada *lane* sebelah kanan:
        - Jika pemain memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command USE_LIZARD*.
        - Jika pemain tidak memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command TURN_RIGHT*.
- Ketika *car* pemain terdapat pada *lane* terkanan:
    - Jika proyeksi *damage* yang diterima pada *lane* pemain lebih kecil dari proyeksi *damage* yang diterima pada *lane* sebelah kiri:
        - Jika pemain memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command USE_LIZARD*.
        - Jika pemain tidak memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command ACCELERATE*.
    - Jika proyeksi *damage* yang diterima pada *lane* sebelah kiri lebih kecil dari proyeksi *damage* yang diterima pada *lane* pemain, maka *bot* akan menggunakan *command TURN_LEFT*

- Jika proyeksi *damage* yang diterima pada *lane* pemain sama dengan proyeksi *damage* yang diterima pada *lane* sebelah kiri:
  - Jika pemain memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command USE_LIZARD*.
  - Jika pemain tidak memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command TURN_LEFT*.

- Ketika pemain berada pada *lane* tengah:
  - Jika *damage* yang diterima pada *lane* sebelah kiri merupakan *lane* dengan proyeksi *damage* terkecil, maka *bot* akan menggunakan *command TURN_LEFT*
  - Jika *damage* yang diterima pada *lane* sebelah kanan merupakan *lane* dengan proyeksi *damage* terkecil, maka *bot* akan menggunakan *command TURN_RIGHT*
  - Jika *damage* yang diterima *lane* pemain merupakan *lane* dengan proyeksi *damage* terkecil:
    - Jika pemain memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command USE_LIZARD*.
    - Jika pemain tidak memiliki *lizard power up* dalam *inventory*, maka *bot* akan menggunakan *command ACCELERATE*.

Menurut penulis, dengan adanya pendefinisian penanganan beberapa kasus secara mendetail maka strategi kedua dapat bekerja dengan performa lebih optimum daripada strategi pertama. Pada strategi kedua, jika *bot* tidak dapat berganti *lane* maka akan diusahakan *car* pemain ditambah *speed*-nya. Hal ini dilakukan agar terdapat *buffer* dalam pengurangan *speed* oleh *obstacle* yang ditabrak oleh *car* pemain.

3.2.4. Strategi Heuristik Pengambilan *Power Up*

Strategi heuristik berikutnya yang dapat diimplementasikan dalam program *bot* adalah strategi pengambilan *power up* yang terdapat pada *map*. Strategi ini mengatur kumpulan *command* perpindahan *lane* yang dibutuhkan agar *bot* dapat

mengambil jumlah *power up pick-ups* paling banyak. Strategi ini juga mengatur pembobotan nilai prioritas dari setiap *power up* pada permainan. Hal tersebut bertujuan untuk menentukan *power up* manakah yang lebih berguna bagi pemain. Sebagai contoh, *power up EMP* lebih diprioritaskan daripada *power up Oil Spill* dikarenakan efek yang disebabkan oleh *power up EMP* lebih terasa dibandingkan *power up Oil Spill* (*power up EMP* menyebabkan *car* lawan terdiam selama ronde berlangsung dan membuat *speed* dari *car* tersebut menjadi 3, sedangkan *power up Oil Spill* hanya men-*damage* sebanyak 1 poin dan menurunkan 1 tingkatan *speed* dari *car* lawan). Sama halnya seperti strategi algoritma *greedy*, penentuan nilai prioritas dari setiap *power up* juga bersifat heuristik (tergantung kesepakatan dari pemrogram *bot*). Berikut ini merupakan daftar dari *power up* pada permainan beserta nilai prioritasnya:

- *Boost* serta *EMP* memiliki nilai prioritas 4. Hal tersebut dikarenakan kedua *power up* tersebut sangat berkontribusi dalam memenangkan permainan. *Power up boost* memiliki efek menaikkan *speed* pemain menjadi *maximum speed* yang mungkin pada keadaan *damage* tersebut. Sedangkan *power up EMP* dapat membuat *car* lawan terdiam selama satu ronde dan kemudian mengurangi *speed car* lawan tersebut menjadi 3 *blocks* per ronde. Selain itu, *range* dari *power up EMP* juga sangatlah menyeluruh, sehingga peluang untuk mengenai suatu target dengan kondisi tertentu hampir mendekati 100%
- *Tweet* serta *lizard* memiliki nilai prioritas 2. Meskipun tidak sebagus kedua *power up* di atas, kedua *power up* ini juga memiliki kontribusi yang tidak kecil dalam memenangkan permainan. *Power up tweet* bekerja dengan memanggil *cyber truck* pada suatu *block*. Jika *car* lawan menabrak *cyber truck*, maka *car* lawan akan terdiam di belakang *cyber truck* selama ronde berlangsung, *speed* dari *car* tersebut berkurang menjadi 3 *blocks* per ronde, serta dikenai 2 poin *damage*. Tetapi, *power up tweet* dapat dengan mudah dihindari oleh *car* lawan jika strategi *obstacle avoidance* yang dimiliki oleh *bot* lawan sudah cukup terlatih. Oleh karena itu, akan sangat langka suatu *cyber truck* ditabrak oleh *car* lawan (hanya terjadi ketika *car* lawan tidak

bisa menghindar). *Power up lizard* bekerja dengan membuat *car* pemain tidak mengenai *object* apapun pada arah geraknya. *Power up* ini berguna jika pemain tidak bisa menghindari suatu *obstacle* dengan melakukan perpindahan *lane*. Tetapi, terdapat efek samping ketika pemain menggunakan *power up lizard*, yaitu ketika *power up lizard* aktif *car* pemain tidak dapat mengambil *power up pick-ups* meskipun *lane* tersebut dilewati oleh *car* pemain.

- *Oil spill* memiliki prioritas 1. *Power up oil spill* bekerja dengan mengubah *block* dibawah *car* pemain menjadi genangan oli. Jika *car* lawan melewati genangan tersebut, maka *speed* dari *car* tersebut akan berkurang 1 tingkatan serta dikenai 1 poin *damage*. Jika dibandingkan dengan *power up* lainnya, efek dari *oil spill* ini tidak terlalu kentara. Selain itu, jika *bot* lawan memiliki *obstacle avoidance*, maka akan sangat mudah untuk *car* lawan menghindari *oil spill* ini.

Salah satu strategi yang dapat diimplementasikan pada kasus ini adalah memilih *lane* dengan jumlah nilai prioritas *power up pick-ups* paling banyak. Jumlah *block* yang masuk perhitungan tersebut hanyalah *block* yang mungkin dicapai pada ronde berikutnya, bukan seluruh *block* pada depan *car* pemain yang dapat diakses.

3.2.5. Strategi Heuristik Penggunaan *Offensive Power Up*

Terakhir, strategi heuristik yang dapat diimplementasikan adalah mekanisme penggunaan *power up* yang memiliki sifat “menyerang” seperti *tweet*, *EMP*, dan *oil spill*. Strategi ini mengatur kondisi apa saja yang perlu dipenuhi agar *power up* tersebut dapat dieksekusi. Kondisi tersebut haruslah tidak terlalu spesifik agar *power up* sering digunakan serta menciptakan keadaan dimana *power up* tersebut memiliki efek paling kentara pada *car* lawan. Salah satu strategi heuristik yang dapat diimplementasikan dalam bentuk algoritma *greedy* untuk melakukan *handling* penghindaran penggunakan *offensive power up* adalah sebagai berikut:

- *Tweet*: Digunakan ketika pemain memiliki *power up tweet* pada *inventory*, perbedaan *blocks* antara *car* pemain serta *car* lawan lebih besar dari 8 *blocks*. Kemudian, nilai parameter *x* dan *y* pada *command USE_TWEET* ditentukan oleh posisi serta *speed* dari *car* lawan. Nilai parameter *x* adalah

posisi *lane* tempat *car* lawan berada sementara nilai parameter *y* ditentukan dari posisi *block* tempat *car* lawan berada dan juga *speed* dari *car* tersebut.

- *EMP*: Digunakan ketika pemain memiliki *power up EMP* pada *inventory*, posisi *block* dari *car* pemain lebih kecil daripada posisi *block* dari *car* lawan, perbedaan *lane* dari *car* pemain dengan *lane* dari *car* lawan lebih kecil sama dengan 1, serta *speed* dari *car* lawan lebih besar dari 3 *blocks* per ronde.
- *Oil spill*: Digunakan ketika pemain memiliki *power up oil spill* pada *inventory*, posisi *block* dari *car* pemain berada pada *range* posisi *block* dari *car* lawan dengan posisi *block* dari *car* lawan ketika lawan sudah bergerak pada ronde selanjutnya. Hal tersebut menjamin bahwa *car* lawan akan mengenai *oil spill* yang ditumpahkan oleh *car* pemain.

3.3. Analisis Efisiensi dari Kumpulan Solusi Algoritma *Greedy*

Pada permainan Overdrive, banyak *gamestate* yang bisa kita ketahui dengan mudah seperti posisi pemain, posisi lawan, list *power up* pemain, dan lain-lain. Tentu hal tersebut memudahkan program agar berjalan dengan efektif. Selain itu, state Map yang bisa dideteksi oleh pemain hanyalah 20 blok kedepan dan 5 blok kebelakang sehingga pencarian pada map juga sangat singkat.

Pada strategi *Fix* dan *Damage mechanism*, kita cukup melihat data damage dan speed pada *gamestate* player yang sudah ada sehingga pada strategi ini, kompleksitas waktunya konstan atau O(1).

Pada strategi *Boost* dan *Accelerate*, kita perlu melakukan pencarian pada array *power up* yang dimiliki oleh pemain untuk mencari ketersediaan *power up boost* dan array *power up* ini sudah tersedia pada *gamestate* sehingga pada strategi ini, perkiraan kompleksitas waktu adalah O(n) dengan n banyaknya elemen pada array *power up*.

Pada strategi menghindari *obstacle* dan pencarian *power up*, program akan melakukan pencarian terhadap blok yang ada di depan posisi mobil. Pada algoritma yang dibuat, blok yang dicari maksimal adalah 15 *block* di depan mobil di lane yang sama, lane di kiri mobil, dan lane di kanan mobil sehingga strategi ini membutuhkan kompleksitas yang cukup baik yaitu O(n) dengan n banyak blok yang dicek dan $n \leq 15$ sehingga bisa dianggap konstan atau O(1).

Untuk strategi terakhir yaitu strategi *Offensive*, program hanya akan melakukan pengecekan pada kondisi-kondisi yang ada pada *game state* seperti posisi pemain, posisi musuh dan ketersediaan *powerup* pada array *powerup* sehingga kompleksitas waktunya adalah $O(n)$ dengan n banyak elemen pada *array*.

3.4. Analisis Efektivitas dari Kumpulan Solusi Algoritma *Greedy*

3.4.1. Efektivitas Strategi Heuristik *Fix* dan *Damage Mechanism*

Strategi pertama belum tentu berdampak baik terhadap performa *overall* dari *bot*. Hal ini dikarenakan meskipun *car* menerima *damage* kurang dari 5 poin, *maximum speed* yang dapat dicapai oleh *car* pemain akan menurun sesuai tingkatan yang telah dijelaskan pada bab 1 dan bab 2. Oleh karena itu, dibutuhkan strategi baru yang lebih memperhatikan *maximum speed*.

Strategi kedua sekilas terlihat lebih baik dari strategi pertama. Secara *overall* mungkin strategi *greedy* ini merupakan strategi yang paling baik, tetapi menurut penulis strategi ini masih menyisakan ruang untuk melakukan *improvement*. *Bot* tidak perlu langsung diperbaiki bila *bot* tersebut belum memerlukan perbaikan. Strategi yang terbaik menurut penulis merupakan strategi yang berada diantara spektrum strategi pertama serta strategi kedua.

Strategi ketiga menggabungkan kedua strategi yang telah dipaparkan sebelumnya, dimana *bot* perlu melakukan *command FIX* meskipun masih bisa bergerak dan hanya jika *bot* telah mencapai keadaan dimana *speed* dari *bot* tidak dapat bertambah kecuali dilakukan *command FIX*. Menurut penulis, strategi ini merupakan strategi paling efisien untuk menyelesaikan subpermasalahan *damage mechanism* pada permainan.

3.4.2. Efektivitas Strategi Heuristik *Speed* dan *Boost Power Up*

Strategi pertamasusah diintegrasikan dengan strategi lainnya karena kondisi untuk melakukan *command ACCELERATE* akan terlalu mudah terpenuhi, sehingga penggunaan *command* lainnya akan lebih mudah terabaikan oleh algoritma *greedy* bot. Dibutuhkan strategi heuristik baru dimana di dalam strategi baru tersebut diberi

batasan-batasan keadaan untuk melakukan *command ACCELERATE* maupun penggunaan *boost power up*.

Strategi kedua merupakan strategi yang lebih superior dibandingkan strategi pertama. Pada strategi kedua, diusahakan penggunaan *command ACCELERATE* serta penggunaan *boost power up* menghasilkan capaian paling optimum dan tidak sia-sia.

3.4.3. Efektivitas Strategi Heuristik Menghindari *Obstacle* dan *Lizard Power Up*

Pada bagian ini, strategi yang digunakan sebenarnya tidak jauh berbeda , namun pada strategi kedua algoritma akan lebih efektif karena semua kondisi dan kemungkinan dari kondisi permainan dibuat menjadi pertimbangan, sehingga strategi kedua akan lebih efektif daripada strategi pertama

3.4.4. Efektivitas Strategi Heuristik Pengambilan *Power Up*

Disini, hanya ada satu strategi yang digunakan karena pengimplementasian sederhana, hanya butuh mencari lane dengan powerup yang lebih banyak. Namun untuk menambah efektivitas, seperti yang sudah dijelaskan, kami melakukan pembobotan pada powerup yang ada. Menurut kami, pembobotan yang kami lakukan sudah merupakan yang paling efektif menghitung dari kekuatan dan kondisi dari masing – masing powerup. Misal kami menghitung boost sebagai prioritas utama, karena bagaimanapun player yang menang adalah player dengan rata rata kecepatan terbesar tiap waktunya, sehingga powerup boost merupakan salah satu powerup terkuat yang bisa kita gunakan

Selain boost, terdapat EMP, LIZARD, OIL, dan TWEET. EMP memiliki bobot yang sama dengan boost, namun pengambilan boost akan tetap menjadi prioritas utama karena alasan yang sudah dijelaskan sebelumnya. Setelah itu, untuk tweet dan lizard juga memiliki bobot yang sama karena keduanya sangat berguna pada kondisi tertentu dan terakhir oil memiliki bobot paling kecil karena kondisinya yang sempit dan mudah dihindari.

3.4.5. Efektivitas Strategi Heuristik Penggunaan *Offensive Power Up*

Pada strategi ini dilakukan hal yang sama pada strategi sebelumnya, namun hanya dilakukan untuk powerup yang bersifat *offensive*. Disini kami membobot powerup sama seperti sebelumnya untuk memilih powerup mana yang paling

efektif digunakan pada kondisi permainan saat itu juga. Pembobotan masih sama dengan strategi sebelumnnya

3.5. Strategi *Greedy* yang Digunakan pada Program *Bot*

Strategi heuristik yang penulis ambil sebagai algoritma *greedy* utama dari program *bot* permainan Overdrive ini adalah penggabungan dari seluruh strategi yang telah dipaparkan pada subbab 3.2. Penulis mengambil seluruh strategi untuk setiap *mechanism* pada permainan Overdrive agar program *bot* dapat menangangi seluruh kemungkinan kasus dari *game state* secara efektif dan tepat sasaran. Agar seluruh strategi heuristik dapat digabungkan menjadi suatu algoritma *greedy*, penulis harus mendefinisikan urutan dari strategi mana yang harus dieksekusikan terlebih dahulu. Salah satu cara mendefinisikan urutan tersebut adalah dilihat dari seberapa besar prioritas suatu strategi jika dibandingkan dengan strategi lainnya. Urutan prioritas tersebut memiliki sifat heuristik pula, dimana pemrogram harus menentukan bagaimana urutannya agar menghasilkan algoritma *greedy* yang paling optimum. Pemrogram dapat menggunakan logika atau mengecek urutan strategi untuk memformulasikan urutan mana saja yang dapat diimplementasikan sebagai algoritma *greedy*.

Setelah dipikirkan dan kemudian mencobanya pada *game engine*, penulis telah berhasil memformulasikan urutan pengeksekusian strategi heuristik yang menghasilkan algoritma *greedy* paling optimum. Penulis berusaha mencari permutasi urutan pengeksekusian strategi yang memiliki peluang menang paling besar bila ditandingkan dengan *bot* lawan. Algoritma *bot* lawan yang digunakan sebagai tolak ukur adalah *reference bot* bawaan yang dibuat secara *random*, *bot* kolega penulis yang dibuat menggunakan algoritma *greedy* pula, serta *bot* yang memenangkan Entellect Challenge 2020 dibuat menggunakan dengan struktur data *graph* serta menggunakan algoritma *graph traversal*. Urutan prioritas dari pengeksekusian strategi heuristik yang telah diformulasikan oleh penulis adalah sebagai berikut:

- Strategi *fix* dan *damage mechanism* ketika *car* pemain sudah benar-benar rusak.
- Strategi *speed mechanism* ketika *car* pemain mencapai *minimum speed*.
- Strategi penggunaan *offensive power up* ketika *car* lawan sudah mendekati *finish line*.

- Strategi menghindari *obstacle* dan penggunaan *lizard power up* digabungkan dengan strategi pengambilan *power up* dengan batasan.
- Strategi *speed mechanism* dan penggunaan *boost power up*.
- Strategi penggunaan *offensive power up*.
- Strategi *speed mechanism* digabungkan dengan strategi pengambilan *boost power up*.
- Strategi *fix* dan *damage mechanism* ketika *car* pemain mencapai *maximum speed*.
- Strategi pengambilan *power up* ketika tidak terdapat *obstacle*.

Strategi *greedy* yang dibuat penulis selalu menghasilkan *command* untuk setiap kasus pada *game state*. Hal ini dilakukan agar *bot* pemain tidak dikenai penalti *score* ketika tidak berhasil melakukan suatu *command* pada ronde tersebut. *Command default* ketika *game state* tidak menghasilkan suatu kasus yang membutuhkan *command* khusus adalah dengan menggunakan *command DO_NOTHING*. Tetapi, penggunaan *command DO_NOTHING* tersebut akan sangat jarang dikarenakan banyaknya *handling* kasus yang membutuhkan dilakukan suatu aksi tertentu.

Bab 4: Implementasi dan Pengujian

4.1. Implementasi Algoritma *Greedy* pada Bot Permainan Overdrive

Implementasi algoritma *greedy* pada program terdapat pada *file* Bot.java, dalam method run. Terdapat method lain yang kami buat untuk kemudahan penggunaan berulang pada kode, pseudocode yang dilampirkan hanya dari bagian run(). Untuk melihat method – method lain yang kami gunakan disini, dapat dilihat pada subbab 4.2.

4.1.1. Public Command run

```
function run() -> Command
{ Fungsi utama yang akan mengembalikan command yang akan dilakukan bot }

KAMUS LOKAL
myCar, opponent = Car
blocksMax, rightblocks, leftblocks, blocks,blocksacc = list of lane
terrainMax, terrainRight, terrainLeft, terrainBlocks = list of terrain
cyberTruck = list of boolean
powerFront, powerRight, powerLeft, damageFront, damageRight, damageLeft = integer
minSpeed, speedState1, initialSpeed, speedState2, speedState3, maxSpeed, boostSpeed
= integer
damageCheck, powerCheck, speedState = list of integer
ACCELERATE, DECELERATE, DO_NOTHING, FIX, LIZARD, OIL, BOOST, EMP, TWEET,
TURN_RIGHT, TURN_LEFT = Command

ALGORITMA FUNGSI

// Inisialisasi data yang dibutuhkan
myCar <- gamestate.player
opponent <- gamestate.opponent
blocksMax <- getBlocks(myCar.position.lane, myCar.position.block, gameState, 0)
terrainMax <- blocksmax.terrain
rightblocks <- getBlocks(myCar.position.lane, myCar.position.block, gameState,1)
terrainRight <- rightblocks.terrain
leftblocks <- getBlocks(myCar.position.lane, myCar.position.block, gameState, -1)
terrainLeft <- leftblocks.terrain
rightblocks <- getBlocks(myCar.position.lane, myCar.position.block, gameState,1)
terrainRight <- rightblocks.terrain

if (myCar.boostCounter == 1) then
   if (myCar.boostCounter == 1) then
         blocks = blocksMax[0, Bot.maxSpeed]
   else
         blocks = blocksMax[0, myCar.speed]
else
   blocks = blocksMax


terrainBlocks = blocks.terrain

if (checkAcc(gamestate)) then
   if (blocksMax.size() >= higherSpeed(myCar.speed)) then
         blocksAcc = blocksMax[0,higherSpeed(myCar.speed)
   else
         blocksAcc = blocksMax
else
   blocks
```

```
if (myCar.damage >= 5) then
   -> FIX
```

```
if (myCar.speed == minSpeed) then
   if (canBoost(gameState)) then
         -> BOOST
   else
         -> ACCELERATE

if (myCar.position.block >= 1485) then
   cybertruck = blocksMax.cyberTruck
   if (canBoost(gamestate) && !cybertruck) then
         -> BOOST
   if (myCar.position.x + higherSpeed(myCar.speed) >= 1500 && checkAcc(gamestate)
&& (!cybertruck)) then
         -> ACCELERATE
   if (myCar.positon.x + myCar.speed >= 1500 && !cybertruck || countDamage(blocks)
   == 0) then
         -> attack(gameState)

   if (opponent.position.block >= 1400 || myCar.position.block >= 1400 &&
   countDamage(blocks) == 0) then
         if PowerUp(EMP, myCar.powerups) > 0 && myCar.position.x <
         opponent.position.x then
                if abs(myCar.position.y – opponent.position.y && myCar.speed >= 6
                || opponent.position.block > 1450 then
                     -> EMP
// APABILA LANE DIDEPAN MENGANDUNG TERRAIN YANG BEREFEK NEGATIF, MENGGUNAKAN GREEDY
   BY POWERUP DAN DAMAGE

if(checkObstacle(blocksMax,gameState) then
   powerFront <- countPowerup(terrainMax)
   powerRight <- countPowerup(terrainRight)
   powerLeft <- countPowerup(terrainLeft)
   damageFront <- countDamage(blocksmax)
   damageRight <- countDamage(rightblocks)
   damageLeft <- countDamage(leftblocks)

   if(myCar.position.y == 1) then
         if(damageFront != damageRight) then
               if(PowerUp(LIZARD, mycar.Powerups) > 0 then
                     -> jumpORattack(gamestate,blocks)
               -> accelORattack(gameState)
         -> chooseMoveObstacle(gameState,blocksAcc,blocks,”RIGHT”,damageRight)
         if(PowerUp(LIZARD,myCar.powerups) > 0) then
               -> jumpORattack(gameState, blocks)
         if(powerFront >= powerRight) then
               -> accelORattack(gameState)
         -> chooseMoveObstacle(gamestate,blocksAcc,blocks, “RIGHT”, damageRight)

   if(myCar.position.y == 4) then
         if(damageFront != damageLeft) then
               if(PowerUp(LIZARD, mycar.Powerups) > 0 then
                     -> jumpORattack(gamestate,blocks)
               -> accelORattack(gameState)
         -> chooseMoveObstacle(gameState,blocksAcc,blocks,”LEFT”,damageLeft)
         if(PowerUp(LIZARD,myCar.powerups) > 0) then
               -> jumpORattack(gameState, blocks)
         if(powerFront >= powerLeft) then
               -> accelORattack(gameState)
         -> chooseMoveObstacle(gamestate,blocksAcc,blocks, “LEFT”, damageLeft)
```

```
   if(damageLeft != 0 && damageRight != 0) then
         if(PowerUp(LIZARDS, myCar.powerups > 0)) then
               move <- jumpORattack(gameState,blocks)
```

```
               if(move == LIZARD) then
                     -> LIZARD

         damageCheck = [damageFront, damageLeft, damageRight]
         sort(damageCheck)

         if(damageCheck[0] == damageCheck[2]) then
               powerCheck = [powerFront, powerLeft, powerRight]
               sort(powerCheck)
               -> CompareLine(powerCheck, powerFront, powerLeft, gameState,
               blocksAcc,blocks)

         if(damageCheck[0] == damageCheck[1]) then
               if(damageCheck[2] == damageRight) then
                     if(powerFront >= powerLeft) then
                           -> accelORattack(gameState)

               if(damageCheck[2] == damageLeft) then
                     if(powerFront >= powerRight) then
                           -> accelORattack(gameState)
                      ->chooseMoveObstacle (gameState, blocksAcc, blocks,
                      "RIGHT",damageRight)
               -> powerLeftOrRight(powerLeft,powerRight,damageLeft,damageRight
                  , gamesState,blocksAcc,blocks)
         -> CompareLine(damageCheck,damageFront,damageLeft,damageRight,gameState
            , blocksAcc, blocks)

   if(damageRight == 0) then
         if(damageLeft == 0) then
               -> powerLeftOrRight(powerLeft,powerRight,damageLeft,damageRight
                  , gameState, blocksAcc, blocks)
         -> chooseMoveObstacle(gameState,blocksAcc,blocks,"RIGHT",damageRight)
   -> chooseMoveObstacle(gameState,blocksAcc,blocks,"LEFT",damageLrft
if(terrainLeft.contains(BOOST) && countDamage(leftblocks == 0)) then
   -> TURN_LEFT
if(checkAcc(gameState)) then
   -> ACCELERATE

// APABILA LANE DIDEPAN TIDAK ADA RINTANGAN, PRIORITAS MELAKUKAN BOOST ATAU ACCELERATE
APABILA BISA. BILA TIDAK, MOBIL AKAN MENCARI LANE YANG MEMPUNYAI BOOST (GREEDY BY
POWERUP & SPEED)

if(PowerUp(BOOST, myCar.powerups) > 0) then
   if(canBoost(gameState) then
         -> BOOST
   if(myCar.damage > 0) then
         -> FIX

if(terrainMax.contains(BOOST)) then
   if(checkAcc(gameState)) then
         -> ACCELERATE
   -> attack(gameState)

if(terrainRight.contains(BOOST) && countDamage(rightblocks) == 0) then
   if(terrainLeft.contains(BOOST) && countDamage(leftblocks == 0) then
         if(countPowerup(terrainLeft) > countPowerup(terrainRight) || countPowerup
   (terrainLeft) == countPowerup(terrainRight) && (myCar.position.y – myCar.p
   osition.y >= 0) then
               -> TURN_LEFT
         -> TURN_RIGHT
```

```
// APABILA LANE DEPAN SUDAH KOSONG & TIDAK ADA BOOST DISEKITAR , MOBIL AKAN MENGECEK
APAKAH DAMAGE MEMBUAT DIRINYA TIDAK BISA MENJADI LEBIH CEPAT, BILA BEGITU KASUSNYA MOBIL
AKAN MELAKUKAN FIX PADA DIRINYA

if(myCar.damage == 4 && myCar.speed == speedState1) then
   -> FIX
if(myCar.damage == 3 && myCar.speed == speedState2) then
   -> FIX

attack <- attack(gameState)

if(attack = “EMP”) then
   -> EMP
if(myCar.damage == 2 && myCar.speed == speedState3) then
   -> FIX

// ALGORITMA TERAKHIR PADA MAIN, APABILA SEMUA KONDISI DIATAS TIDAK TERPENUHI, MOBIL
AKAN MENCARI LANE DENGAN POWERUP PALING BANYAK (NAMUN LANE HARUS TIDAK ADA
OBSTACLE/BLOCKER)

if ((countDamage(rightblocks) == 0) || (countDamage(leftblocks) == 0)) then
   powerupList = [countPowerup(terrainBlocks), countPowerup(terrainRight),
   countPowerup(terrainLeft)]
   sort(powerupList, descending)

   if (powerupList[0] == countPowerup(terrainBlocks)) then
          -> attack(gameState)

   if (powerupList[0] == countPowerup(terrainLeft)) then
          if (myCar.position.y != 1 && countDamage(leftblocks) == 0) then
                if (powerupList.get(0) == countPowerup(terrainRight)) then
                if (myCar.position.lane != 4 && countDamage(rightblocks) == 0) then
                      if (myCar.position.lane - opponent.position.lane >= 0) then
                                  -> TURN_LEFT
                            -> TURN_RIGHT
                -> TURN_LEFT
          if (powerupList[1] == countPowerup(terrainBlocks)) then
                     -> attack(gameState)

         if (powerupList.get(1) == countPowerup(terrainRight)) then
                if (myCar.position.y != 4 && countDamage(rightblocks) == 0) then
                -> TURN_RIGHT;


   if (powerupList[0] == countPowerup(terrainRight)) then
          if (myCar.position.lane != 4 && countDamage(rightblocks) == 0) then
                -> TURN_RIGHT
          if (powerupList[1] == countPowerup(terrainBlocks)) then
                -> attack(gameState)
          if (powerupList.get(1) == countPowerup(terrainLeft))
                if (myCar.position.lane != 1 && countDamage(leftblocks) == 0)
                      -> TURN_LEFT

-> attack(gameState)
```

Untuk kode lengkap dapat dilihat pada link berikut:

https://github.com/rayhankinan/Tubes1_Stima

Untuk video demo beserta penjelasan lengkap algoritma dapat dilihat pada link berikut:

https://www.youtube.com/watch?v=5rKVX4CT4i0

4.2. Penjelasan Struktur Data pada Bot Permainan Overdrive

Struktur data pada permainan worms ini berbentuk *class*. *Class* tersebut dapat dibagi menjadi 5 kategori utama, yaitu: *Command* yang berisi perintah – perintah yang bisa diberikan kepada *bot*, *Entities* berisi objek – objek yang bisa di instansiasi pada *game* Overdrive, *Enums* berisi konstan – konstan objek yang merupakan bagian dari permainan, seperti *power ups*, *directions*, dan lain-lain. Terdapat juga Main.java, 4 kategori diatas merupakan *class* yang sudah disediakan dari *entellect challenge* dari awal. Kategori terakhir adalah Bot.java, disini merupakan tempat logika bot disimpan. Pembuatan bot dan pengaplikasian algoritma *greedy* dilakukan pada *class* ini.

Berikut pemaparan lebih mendalam mengenai beberapa class yang ada pada *bot* Overdrive:

A. Kategori *Command*

Kelas – kelas pada kategori ini merupakan aksi/*ability* yang bisa dilakukan oleh *bot*, semua *class* disini merupakan implementasi dari Command.java. Karena mayoritas dari *command* hanya melakukan suatu aksi dan tidak membutuhkan ada atribut, biasanya *class* hanya berisi 1 method yang melakukan *return* suatu string yang menandakan bot akan melakukan aksi/*power up* tersebut, detail sebagai berikut.

i. AccelerateCommand.java

Class yang menghasilkan *new Command* untuk membuat *bot* melakukan *accelerate*.

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “ACCELERATE”

ii. BoostCommand.java

Class yang menghasilkan *new Command* untuk membuat *bot* menggunakan *boost*.

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “USE_BOOST”

iii. ChangeLaneCommand.java

Class ini menggunakan *enum* Direction dan menggunakan *class* tersebut untuk menentukan apakah mobil akan belok kanaan atau kiri.

- Attributes

Attributes	Description
private Direction direction	Atribut dari Class direction, yang memiliki lane dan block sebagai atributnya. Direction juga melakukan enumerasi pada arah arah seperi RIGHT, LEFT, FORWARD, dan BACKWARD

- Methods

Methods	Description
public ChangeLaneCommand(int laneIndicator)	Method yang akan mengubah atribut direction berdasarkan parameter, 1 menandakan kanan dan 2 menandakan kiri
public String render()	Method yang mengembalikan String “TURN X”, X merupakan LEFT atau RIGHT

iv. Command.java

Induk dari semua struktur data yang mewakili perintah yang akan dijalankan

- Methods

Methods	Description
public interface Command()	Method yang akan menghasilkan suatu string sesuai dengan perintah yang akan dijalankan

v. DecelerateCommand.java

Class yang akan menghasilkan *new Command* untuk membuat bot melakukan *Decelerate*

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “DECELERATE”

vi. DoNothingCommand.java

Class yang akan menghasilkan *new Command* untuk membuat *bot* tidak melakukan apa – apa.

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “Accelerate”

vii. EmpCommand.java

Class yang akan menghasilkan *new Command* untuk membuat bot menggunakan *power up Emp*.

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “USE_EMP”

viii. FixCommand.java

Class yang akan menghasilkan *new Command* untuk membuat *bot* melakukan *fix* pada dirinya sendiri

- Methods

Methods	**Description**
public String render()	Method yang akan mengembalikkan string “FIX”

ix. LizardCommand.java

Class yang akan menghasilkan *new Command* untuk membuat *bot* menggunakan *power up Lizard*.

- Methods

Methods	Description
public String render()	Method yang akan mengembalikkan string “USE_LIZARD”

x. OilCommand.java

Class yang akan menghasilkan *new Command* untuk membuat *bot* menggunakan *power up Oil Spill*.

- Methods

Methods	**Description**
public String render()	Method yang akan mengembalikkan string “USE_OIL”

xi. TweetCommand.java

Class yang akan menghasilkan *new Command* untuk membuat bot menggunakan *power up Tweet* dan memanggil *cyber truck* pada (*block*, *lane*).

- Attributes

Attributes	**Description**
private int lane	menandakan *lane* pada map (1-4)

private int block	menandakan *block* di map (posisi horizontal)

- Methods

Methods	**Description**
public TweetCommand (int lane, int block)	Method yang akan memasukan atribut lane dan block dengan parameter
public String render()	Method yang akan mengembalikkan string “USE TWEET lane block”

B. Kategori Entities

Kategori ini berisi kelas – kelas yang anggotanya akan diinstansiasi saat permainan berjalan, anggotanya berupa bagian bagian yang menjadi *resource* permainan seperti mobil, *lane*, *position*, dan yang paling penting merupakan *game state*. Karena mayoritas hanya berupa suatu nilai yang merepresentasikan kondisi pada permainan, isi dari class pada kategori ini hanya berupa atribut – atribut. Berikut ini adalah penjelasan dari setiap *class* yang ada:

i. Car.java

Class yang merepresentasikan mobil pada permainan. Berisi semua status, atribut, dan *state* dari mobil. *Class* ini menggunakan enum PowerUps, yang merupakan *power up* yang bisa digunakan oleh *car* dan State.java, yaitu *enum* yang berisi keadaan/kondisi dari mobil saat ini

- *Attributes*

Attributes	**Description**
public int id	ID dari mobil

public Position position	Posisi dari mobil, posisi terdiri dari lane dan block yang bisa digambarkan sebagao lane = y dan block = y
public int speed	Kecepatan dari mobil, terdapat 7 kecepatan yang bisa dicapai mobil
public State state	Keadaan dari mobil, akan lebih jelas apabila melihat struktur data State.java
public int damage	Besar kerusakan dari mobil, memiliki batas 5
public PowerUps[] powerups	Powerups yang dimiliki oleh mobil
public Boolean boosting	Menandakan apakah mobil sedang dalam kondisi melakukan *boost* atau tidak
public int boostCounter	banyak boost yang dimiliki mobil

ii. GameState.java

Class yang berisi kondisi dari permainan, berisi informasi terkait dengan permainan yang sedang berjalan

- *Attributes*

Attributes	**Description**
public int currentRound	Round permainan yang sedang berjalan
public int maxRounds	Round maksimal dari permainan
public Car player	Merepresentasikan mobil pemain
public Car opponent	Merepresentasikan mobil lawan
public List<Lane[]> lanes	Berisi lanes/ jalur balap yang mendeskripsikan tiap *block* di *map* yang dapat dilihat

iii. Lane.java

Class ini merupakan objek lanes yang merupakan jalur tempat para mobil berjalan, berisi atribut yang merepresentasikan kondisi dari suatu lane.

- Attributes

Attributes	**Description**
public Position position	Merepresentasikan dimana suatu block terletak pada map
public Terrain terrain	Mendefinisikan apa yang ada pada suatu block
public int occupiedByPlayerId	Nilai id player yang ada pada suatu block, bernilai 0 apabila kosong
public boolean cyberTruck	Bernilai true apabila terdapat *cyber truck* dan *false* jika tidak

iv. Position.java

Class yang menggambarkan posisi pada map

- Attributes

Attributes	Description
public int lane	merepresentasikan posisi vertikal sehingga bisa juga disebut “y”
public int block	merepresentasikan posisi horizontal sehingga bisa juga disebut “x”

C. Kategori Enums

Kategori ini berisi kelas – kelas yang anggotanya merupakan konstanta atau suatu nilai yang akan digunakan sepanjang permainan. Isinya merupakan enumerasi – enumerasi untuk objek yang ada pada permainan. Berikut isi detail (Hanya *enum* direction yang memiliki atribut dan method, sisanya hanya berisi enumerasi):

i. Direction.java

Berisi enumerasi dari arah arah yang ada permainan, terdapat juga metode constructor untuk class direction. Enumerasi yang ada pada class ini adalah FORWARD, BACKWARD, LEFT, RIGHT

- Attributes

Attributes	Description
private int lane	Menandakan lane pada map (1-4)
private int block	Menandakan block di map (posisi horizontal)

- Methods

Methods	Description
public TweetCommand (int lane, int block)	Method yang akan memasukan atribut lane dan block dengan parameter
public String render()	Method yang akan mengembalikkan string “USE TWEET lane block”

ii. Powerups.java

Berisi enumerasi powerups yang bisa digunakan oleh mobil pada permainan, Enumerasi yang ada adalah BOOST, OIL, TWEET, LIZARD, EMP

iii. State.java

Berisi enumerasi dari kondisi – kondisi yang bisa dialami pemain, terdapat cukup banyak, yaitu: ACCELERATING, READY, NOTHING, TURNING_RIGHT, TURNING_LEFT, HIT_MUD, HIT_OIL, DECELERATING, PICKED_UP_POWERUP, USED_BOOST, USED_OIL, USED_LIZARD, USED_TWEET, USED_EMP, HIT_WALL, HIT_CYBER_TRUCK, FINISHED

iv. Terrain.java

Berisi enumerasi dari hal hal yang bisa ada pada suatu block, diantaranya EMPTY, MUD, OIL_SPILL, OIL_POWER, FINISH, BOOST, WALL, LIZARD, TWEET, EMP

D. Bot.java

Bot.java berisi logika – logika yang digunakan untuk mobil dalam permainan. Struktur data ini berisi *class bot* yang menggunakan attribut – attribut yang sudah dijelaskan sebelumnya seperti *Command, Powerup, Lanes*, dan bagian permainan lainnya untuk menyusun logika bot yang akan digunakan dalam pemilihan perintah yang akan dijalankan tiap rondenya. Berikut merupakan beberapa detail dari *file* Bot.java:

- *Attributes*

  Atribut pada *class* ini mayoritas merupakan objek – objek yang diambil dari *class* lain dan sudah dijelaskan sebelumnya, sehingga akan redundan apabila dijelaskan ulang. Atribut yang akan dilampirkan disini hanya atribut yang ekslusif ada pada Bot.java

Attributes	**Description**
public static final int minSpeed	Kecepatan paling kecil mobil, sebesar 0
public static final int speedState1	Kecepatan mobil sebesar 3
public static final int initialSpeed	Kecepatan awal mobil saat permainan dimulai, sebesar 5
public static final int speedState2	Kecepatan apabila melakukan accelerate dari state kecepatan 1 atau kecepatan awal, sebesar 6
public static final int speedState3	Kecepatan apabila melakukan accelerate dari state kecepatan 2 sebesar 8
public static final int maxSpeed	Kecepatan maksimal yang bisa dicapai mobil, sebesar 9
public static final int boostSpeed	Kecepatan saat mobil sedang dalam kondisi boosting, sebesar 15

public static final int[] speedState	Array berisi kecepatan yang mungkin untuk mobil

- Methods

Methods	**Description**
private Command powerLeftOrRight (int powerLeft, int powerRight, int damageLeft, int damageRight, GameState gameState, List<Lane> blocksAcc, List<Lane> blocks)	Method untuk memilih antara *lane* kiri atau kanan, digunakan saat terdapat rintangan pada *lane* saat ini
private Command CompareLine (Integer[] damageCheck, int damageFront, int damageLeft, int damageRight, GameState gameState, List<Lane> blocksAcc, List<Lane> blocks)	Method untuk mencari *lane* terbaik yang dapat dipilih, mencari *lane* dengan damage terkecil
private Command accelORattack (GameState gameState)	Method untuk memilih antara melakukan *accelerate* atau mencoba menggunakan *power up* menyeranh
private Command jumpORattack (GameState gameState, List<Lane> blocks)	Method untuk memilih antara menggunakan *lizard* atau *power up* menyerang
private Command chooseMoveObstacle (GameState gameState, List<Lane> blocksAcc, List<Lane> blocks, String side, int damage)	Method untuk melihat *lane* sekarang sebelum belok kanan atau kiri
private int countPowerup(List<Terrain> blocks)	Method untuk menghitung banyak *power up* pada suatu kumpulan *block*
private Command attack(GameState gameState)	Method untuk memutuskan penggunaan *attack* atau *do_nothing*

private Command GetPoint()	Method untuk mencari perintah mana yang akan menghasilkan poin paling besar pada kondisi saat ini
private boolean checkObstacle(List<Lane> blocks, GameState gameState)	Method untuk melihat ada rintangan apa pada suatu *lane*
private boolean checkAcc(GameState gameState)	Method untuk mengecek apakah mungkin melakukan *accelerate* pada ronde ini
private int higherSpeed(int speed)	Method yang akan mengembalikan nilai *speed* mobil kita setelah menggunakan *accelerate*
private int countDamage(List<Lane> blocks)	Method untuk menghitung *damage* yang akan kita dapatkan apabila memilih satu *lane* tertentu
private boolean canBoost(GameState gameState)	Method untuk mengecek apakah mungkin melakukan *boost* pada kondisi permainan saat ini
private List<Lane> getBlocks(int lane, int block, GameState gameState, int pos)	Method untuk mendapat semua kumpulan blocks didepan mobil.

E. Main.java

Class bawaan dari permainan overdrive. Digunakan untuk memulai dan menjalankan permainan

4.3. Pengujian Bot serta Analisis Performansi Bot Permainan Overdrive

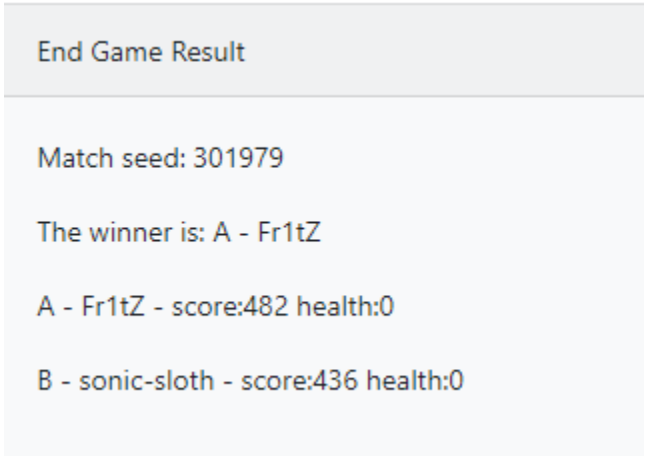
Pengujian bot dilakukan dengan melawankan bot kami, Fr1tZ, dengan bot sonic_sloth. Bot ini merupakan bot yang dibuat oleh kobus-v-schoor dan berhasil meraih posisi ke – 4 pada Entellect Challenge 2020. Berikut *link* dari github bot tersebut: https://github.com/kobus-v-schoor/entelect-2020.

Pengujian bisa saja dilakukan dengan melawan *reference bot* yang diberikan dari challenge awal, namun logika bot ini masih sangat buruk, bahkan beberapa *command* tidak

digunakan sama sekali, sehingga terkadang tidak bisa melakukan pengujian saat terkena suatu powerup, misalnya karena reference bot tidak pernah menggunakan *tweet*, tidak bisa dilakukan pengujian pada deteksi *cyber truck*. Oleh karena itu, kami memilih *bot* sonic_sloth ini untuk benar – benar melakukan pengujian dan melihat performa *bot* kami saat dihadapkan *bot* yang benar – benar tangguh.

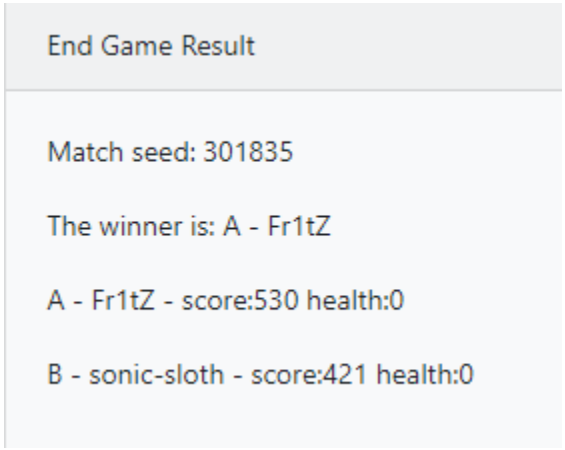
Permainan Overdrive ini memiliki banyak faktor random dan kehokian, sehingga perlu dilakukan banyak pengujian untuk mendapatkan hasil yang sesuai. Diputuskan untuk melakukan pengujian sebanyak 10 kali, dan digunakan visualizer https://entelect-replay.raezor.co.za/# untuk melihat hasil yang didapat. Berikut hasil akhir match match tersebut:

- Match 1



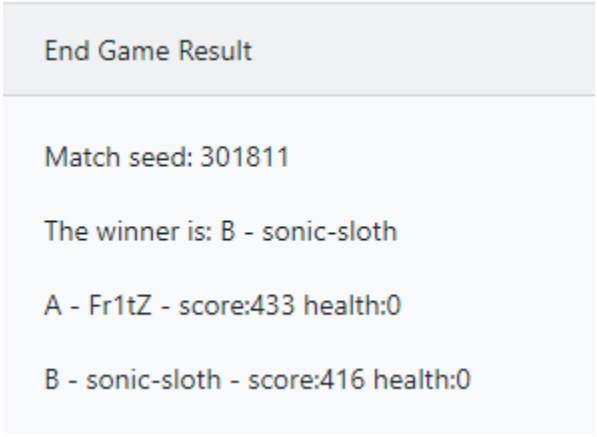
Jarak antar mobil di posisi terakhir: 11 *block*

- Match 2



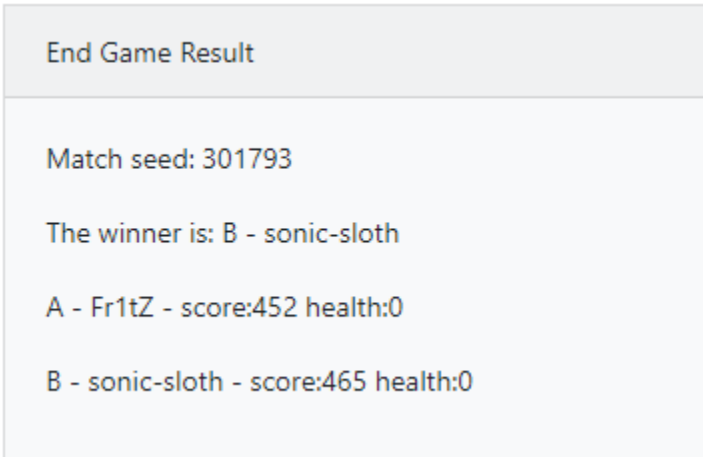
Jarak antar mobil di posisi terakhir: 28 *block*

- Match 3



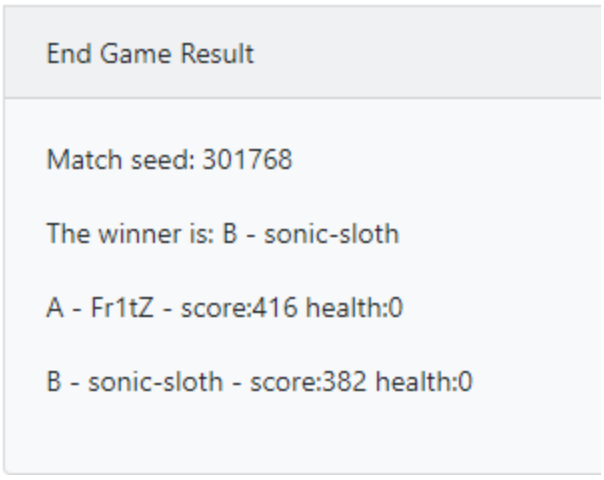
Jarak antar mobil di posisi terakhir: 42 *block*

- Match 4



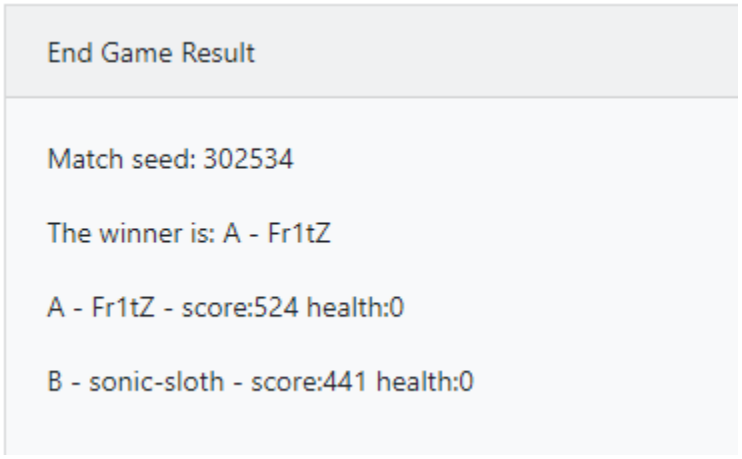
Jarak antar mobil di posisi terakhir: 27 *block*

- Match 5



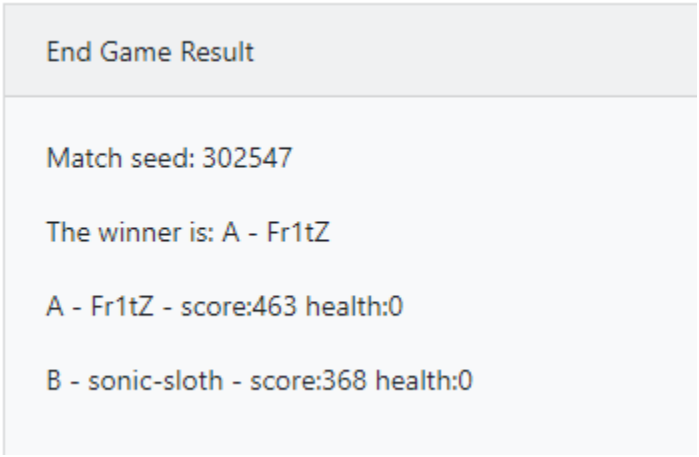
Jarak antar mobil di posisi terakhir: 14 *block*

- Match 6



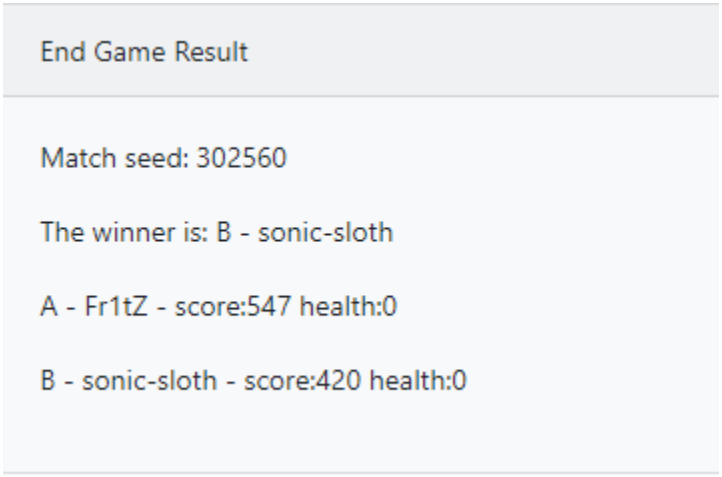
Jarak antar mobil di posisi terakhir: 20 *block*

- Match 7



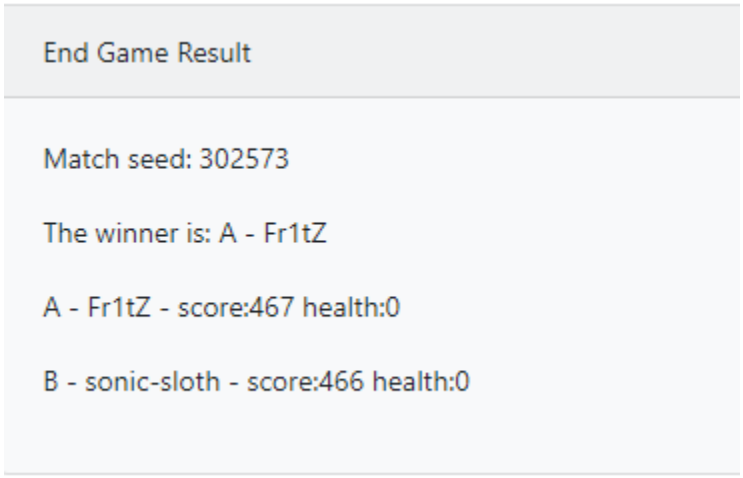
Jarak antar mobil di posisi terakhir: 16 *block*

- Match 8



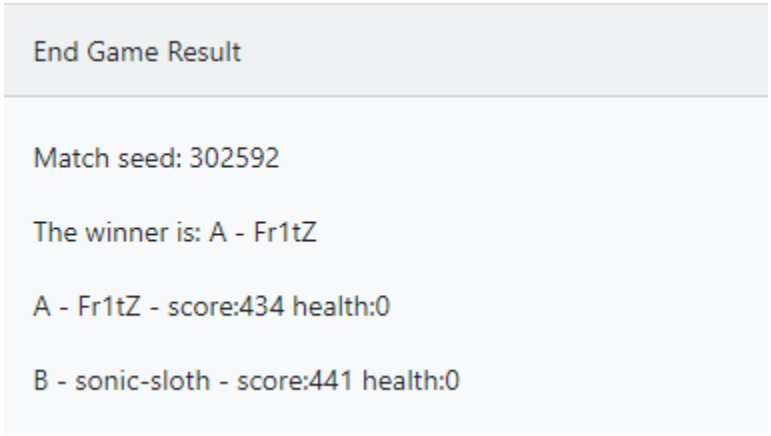
Jarak antar mobil di posisi terakhir: 19 *block*

- Match 9



Jarak antar mobil di posisi terakhir: 5 *block*

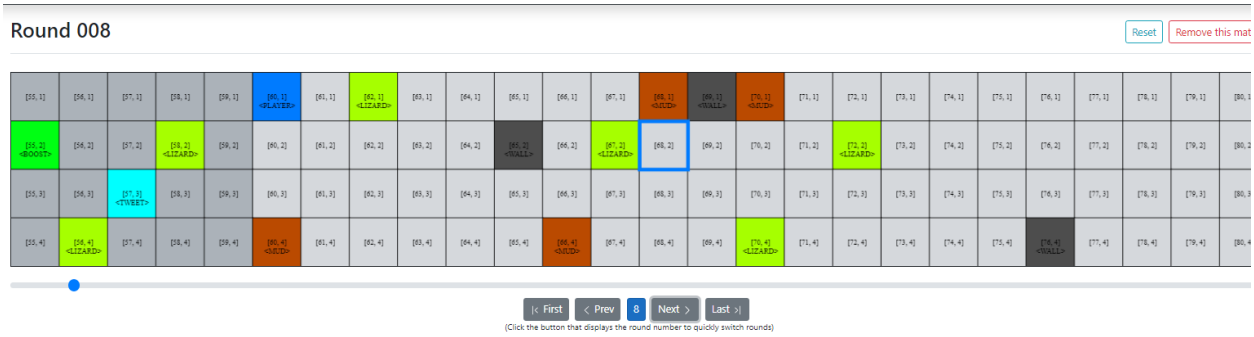
- Match 10



Jarak antar mobil di posisi terakhir: 14 *blocks*

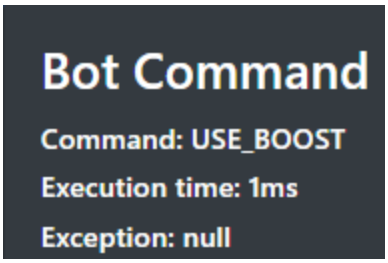
Setelah 10 pengujian, hasil yang didapat adalah bot kami berhasil memenangkan permainan 6 dari 10 kali pengujian. Dapat dilihat juga pada posisi akhir, jarak antara kedua mobil tidak pernah jauh sehingga bisa disimpulkan bahwa pertandingan antara kedua *bot* sangat sengit. Dari sini didapat bahwa algoritma *Greedy* yang kami buat cukup optimal dan dapat mengalahkan bot – bot lain yang tangguh, dilihat dengan persentase kemenangan sebesar 60%.

Berikut akan coba dilakukan analisis lebih mendalam pada match 9 untuk melihat apakah algoritma *greedy* yang kami terapkan benar dilakukan oleh *bot*.

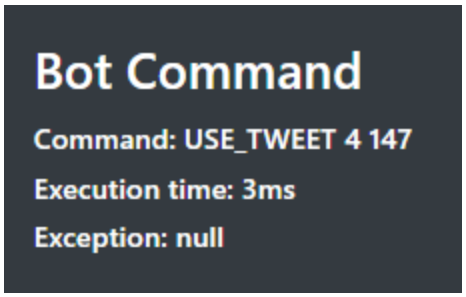


Disini dapat terlihat, saat bot kami terhalang oleh beberapa *obstacle*, dia akan mengecek apakah di kanan ada *obstacle*. Karena ada, *bot* terpaksa memilih *lane* yang menghasilkan damage paling kecil (*greedy by damage*) oleh karena itu *bot* belok kanan. *Bot* tidak menggunakan *lizard* karena apabila menggunakan *lizard*, *bot* akan mendarat pada block yang memiliki *obstacle*.

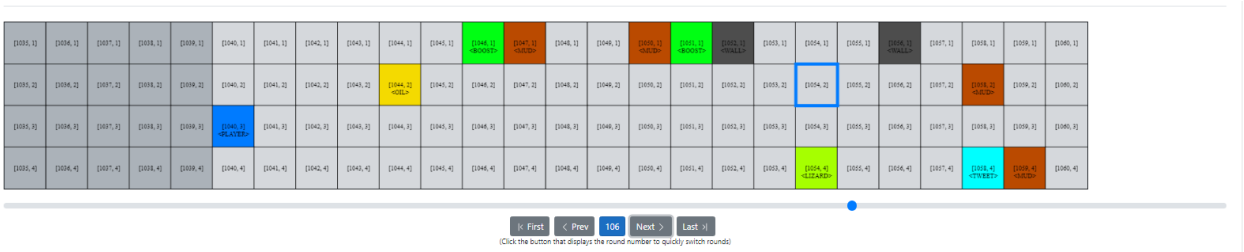
Pada ronde 19, *bot* terhalang obstacle *wall*, saat *bot* mencoba menggunakan *lizard*, ternyata apabila digunakan bot akan mendarat di *obstacle*, sehingga dicoba melihat pilihan lane kanan dan kiri. Setelah itu, *bot* akan melakukan pengecekan *damage* (*greedy by damage*) karena kedua *lane* sama sama kosong, *bot* akan masuk ke algoritma *greedy* berikutnya, yaitu *greedy by powerup*. Disini teruji benar bahwa bot belok kiri, karena lane kiri memiliki powerup yang lebih banyak



Saat didepan tidak ada apa – apa, bot akan mencoba melakukan beberapa hal. Disini seharusnya *bot* melakukan *boost* karena algoritma kami membuat *bot* selalu memprioritaskan *boost* (*greedy by speed*). Hasil ini menunjukkan bahwa algoritma berhasil dan *bot* melakukan *boost*.



Ronde berikutnya, *bot* masih dalam kondisi *boosting* dan *lane* kosong . Karena sedang *max speed*, *lane* kiri dan kanan terdapat *obstacle*, *bot* akan masuk pada algoritma Greedy*greedy* heuristik kami dimana *bot* akan menggunakan *powerup*. Disini, kami meranking *offensive powerup* dari terkuat ke terlemah EMP > TWEET > OIL (Oil bisa diprioritaskan pada kasus – kasus tertentu). Bot akan selalu melakukan emp apabila melihat ada lawan didepan, akan selalu menggunakan oli apabila posisi lawan pas dibelakang. Karena kedua kondisi tidak terpenuhi, *bot* akan melakukan *tweet*, dan disini terbukti algoritma berhasil.



Terakhir, dilihat saat mobil tidak terhalang apa - apa, sedang *boosting*, melakukan lagi *greedy by powerup*. Bisa dilihat dari ronde ini, *bot* memilih belok kiri karena terdapat *powerup oil*.

Dari analisis detail *match* ronde 9 tersebut, bisa dilihat bahwa *bot* berhasil mengikuti perintah dari algoritma *greedy* yang kami implementasikan, dan dilihat dari bagaimana kami berhasil memenangkan pertadingan, membuktikan bahwa algoritma *greedy* yang kami buat cukup baik untuk bisa mengalahkan *bot* lain yang cukup tangguh

Bab 5: Kesimpulan dan Saran

5.1. Kesimpulan

Kelompok kami berhasil mengimplementasikan algoritma *greedy* untuk membuat bot permainan Overdrive yang bisa mencapai tujuan objektif yaitu mencapai garis *finish*. Dari hasil yang didapatkan, dapat dilihat bahwa penggunakan strategi *greedy* cukup optimal dalam kasus ini, dikarenakan pada permainan Overdrive pemilihan paling tamak pada tiap langkahnya kemungkinan besar merupakan kemungkinan terbaik untuk permainan secara umum. Untuk mengilustrasikan, misal pada suatu langkah kita melakukan *greedy* dengan mencoba meraih kecepatan sebesar mungkin pada saat itu, langkah yang kita ambil ini sejalan dengan permainan untuk mencoba mencapai garis finish sebelum lawan sehingga merupakan salah satu pilihan terbaik yang bisa bot lakukan saat itu.

Namun, tentu strategi *greedy* ini juga perlu didampingi dengan teknik heuristik, agar strategi *greedy* yang digunakan lebih optimal dan cocok untuk berbagai kondisi. Secara umum, *bot* yang kami buat berhasil membuktikan bahwa strategi *greedy* cukup baik digunakan dalam pembuatan bot, didukung dengan keberhasilan pengujian melawan bot – bot lain yang ada, salah satunya bot milik pemenang ke – 4 pada turnamen overdrive silam.

5.2. Saran

Terkait dengan topik ini, berikut beberapa saran yang bisa kita ajukan untuk selanjutnya:

- Alangkah baiknya dilakukan pembagian tugas terlebih dahulu agar *workload* masing – masing anggota jelas dan pengerjaan lebih terstruktur
- Penulis menyarankan untuk berikutnya, pembuatan laporan dapat dilakukan lebih cepat lagi dan tidak terlalu mepet dengan tanggal pengumpulan
- Terakhir, pengujian yang dilakukan melawan bot lain bisa dibuat lebih adil lagi. Salah satunya dengan membuat map statis, dan kedua pemain bergantian posisi. Dilihat dari banyaknya faktor *luck* yang lumayan berpengaruh pada permainan

Daftar Pustaka

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

https://ksn.toki.id/data/pemrograman-kompetitif-dasar.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Greedy-(2022)-Bag3.pdf