Tugas Besar 1 IF2211 Strategi Algoritma: Pemanfaatan Algoritma *Greedy* dalam Aplikasi Permainan "Overdrive"

Anggota Kelompok "Sunlight Yellow Overdrive":

Gede Prasidha Bhawarnawa	13520004
Aditya Prawira N	13520049
Owen Christian Wijaya	13520124

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2021

DAFTAR ISI

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.

Pada tugas besar pertama Strategi Algoritma ini, digunakan sebuah game engine yang mengimplementasikan permainan Overdrive. Tugas besar kali ini adalah mengimplementasikan bot mobil dalam permainan Overdrive dengan menggunakan strategi *greedy* untuk memenangkan permainan. Untuk mengimplementasikan bot tersebut, mahasiswa disarankan melanjutkan program yang terdapat pada *starter-bots*.

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

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

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

BAB 2
LANDASAN TEORI

A. Dasar Teori

Algoritma *greedy* atau "rakus" adalah algoritma yang umum dipakai untuk menyelesaikan permasalahan optimasi. Permasalahan optimasi adalah permasalahan dimana harus ditemukan suatu solusi optimal dari semua kemungkinan solusi. Terdapat dua jenis permasalahan optimasi, yaitu permasalahan yang dicari solusi maksimalnya atau maksimasi (*maximization problems*) dan permasalahan yang dicari solusi minimalnya atau minimasi (*minimization problems*).

Dasar dari algoritma *greedy* adalah membentuk solusi langkah per langkah menilai dari keputusan terbaik yang bisa diambil per langkah tersebut. Dengan kata lain, pada setiap langkah algoritma *greedy*, akan dipilih solusi optimum lokal. Penggunaan algoritma *greedy* dilakukan dengan harapan setiap optimum lokal yang dipilih akan menghasilkan solusi optimum global.

Pada setiap pemilihan solusi optimum lokal, terdapat tiga syarat yang perlu dipenuhi. Syarat pertama adalah solusi harus memenuhi kebutuhan dan batasan yang ada pada permasalahan. Lalu, solusi haruslah merupakan solusi optimum lokal atau solusi terbaik diantara semua kemungkinan solusi lainnya. Terakhir, tidak dapat dilakukan *backtracking* atau mundur langkah setelah mengambil suatu solusi optimum lokal.

Terdapat enam elemen dari algoritma *greedy*. Elemen pertama adalah sebuah himpunan kandidat solusi yang akan dipilih pada setiap langkah. Kedua adalah sebuah himpunan solusi yang sudah dipilih, terurut berdasarkan langkah-langkahnya dari awal sampai akhir algoritma. Elemen ketiga adalah fungsi solusi, fungsi yang akan menentukan apakah anggota himpunan kandidat yang dipilih sudah memberikan solusi. Elemen keempat adalah fungsi seleksi, fungsi yang memilih kandidat solusi berdasarkan strategi *greedy* tertentu, umumnya pemilihan kandidat didasarkan pada heuristik. Kelima adalah fungsi kelayakan, atau fungsi yang akan memeriksa apakah kandidat solusi yang dipilih layak atau *feasible* dimasukkan ke dalam himpunan solusi atau tidak. Terakhir adalah fungsi objektif, berfungsi untuk memaksimumkan atau meminimumkan solusi sesuai dengan permasalahan yang ingin diselesaikan. Keenam elemen tersebut bekerja sama secara harmoni dengan alur dimulai dari pencarian elemen solusi dari himpunan kandidat. Elemen solusi tersebut harus memenuhi fungsi solusi, seleksi, dan kelayakan. Setelah itu, sebelum dimasukkan ke dalam himpunan solusi, elemen solusi dioptimisasi oleh fungsi objektif.

Secara umum, algoritma *greedy* jarang digunakan untuk menghasilkan solusi optimum global. Algoritma *greedy* lebih sering digunakan untuk permasalahan yang membutuhkan hampiran atau aproksimasi solusi optimum global, atau solusi yang tidak berbeda jauh dengan solusi optimum globalnya.

B. Cara Kerja Program

Program yang diimplementasikan dan dimodifikasi untuk tugas besar ini adalah permainan *Overdrive*. Pada source file yang dapat diunduh melalui link https://github.com/EntelectChallenge/2020-Overdrive/releases/tag/2020.3.4, terdapat beberapa pilihan bahasa pemrograman yang dapat digunakan untuk mendesain bot mobil. Pada tugas besar ini, bahasa pemrograman dilimitasi pada Java dengan file konfigurasi menggunakan bahasa Javascript. Dari source file telah disediakan dua contoh bot, starter-bot dan reference-bot. Perubahan program dapat dilakukan di file *bot.java*, spesifik pada fungsi run() pada starter-bot. Setelah dilakukan modifikasi program, bot dapat di-*build* dengan menggunakan Maven di Intellij IDEA Jetbrains. Setelah di-*build*, akan dibuat folder baru bernama "target" berisikan file "java-starter-bot-jar-with-dependencies.jar". Untuk memulai permainan, diperlukan dua buah file .jar yang dapat dicompile secara bersamaan menggunakan file "run.bat". Lalu, akan terbuka sebuah command prompt yang menampilkan proses keberjalanan match permainan. Setelah selesai, akan disimpan log match yang telah dilakukan di folder "match-logs".

BAB 3
APLIKASI STRATEGI GREEDY

A. Proses Mapping Persoalan Overdrive Menjadi Elemen-Elemen Algoritma Greedy

Permainan Overdrive dimainkan secara otomatis menggunakan fungsi "run()" pada file "bot.java". Fungsi run() akan mengembalikan output Command. Command adalah custom I/O yang digunakan untuk mengrender perintah-perintah untuk mengendalikan mobil. Bila command-nya valid, bot akan menggerakkan mobil sesuai dengan command-nya.

Jika diturunkan ke dalam elemen-elemen algoritma greedy, maka himpunan kandidat (C), atau himpunan yang berisikan kandidat pilihan yang dapat dipilih pada setiap langkah, dapat dinyatakan sebagai berikut $C = \{NOTHING, ACCELERATE, TURN\_LEFT, TURN\_RIGHT, DECELERATE, USE\_BOOST, USE\_OIL, USE\_LIZARD, USE\_TWEET, USE\_EMP, FIX\}$

Himpunan solusi (S) pada algoritma greedy berisikan semua solusi dari awal program dijalankan sampai program berakhir atau semua kandidat solusi yang memenuhi fungsi solusi, dimana solusi merupakan optimum lokal atau solusi terbaik yang bisa diambil pada setiap langkah. Maka dari itu, isi dari S sangat bergantung pada map tempat balapan dijalankan, namun konsisten terurut isinya dari ronde pertama hingga ronde terakhir.

Fungsi kelayakan (feasibility) adalah fungsi yang pada algoritma greedy adalah fungsi yang memeriksa apakah kandidat solusi yang terbaik merupakan solusi yang paling *feasible* pada suatu langkah. Dikarenakan tujuan yang ingin dicapai secara keseluruhan (optimum global) adalah mencapai garis finish terlebih dahulu dengan jarak sejauh mungkin dari lawan, damage sesedikit mungkin, dan skor setinggi mungkin. Maka dari itu, kandidat solusi yang memenuhi fungsi kelayakan harus memenuhi paling tidak satu dari syarat berikut:

a. Langkah yang diambil haruslah untuk memperjauh jarak antara mobil kita dengan mobil lawan jika kita berada di depan lawan. Jika berada di belakang lawan, langkah yang diambil haruslah untuk mempersempit jarak antara mobil kita dan lawan hingga akhirnya dapat menyalip mobil lawan.
b. Jika langkah (a) tidak dapat dilakukan karena mobil terhalang oleh hambatan seperti oil spill, mud, wall, atau cybertruck, langkah yang diambil haruslah untuk menghindari tabrakan karena tabrakan dapat mengakibatkan kehilangan kecepatan dan bertambahnya damage.
c. Jika langkah (b) tidak dapat dilakukan karena semua jalur di depan mobil, baik itu lurus ke depan, belok ke kiri, atau belok ke kanan, maka mobil akan memilih jalur yang akan memberikan damage minimum
d. Jika mobil memiliki power up yang bersifat ofensif atau menjatuhkan lawan, mobil dapat menggunakan power up sesuai dengan cara pakainya dengan tujuan menghentikan atau menghambat lawan.
e. Jika mobil memiliki power up yang dapat dipakai untuk meningkatkan kecepatan atau membantu untuk menghindari hambatan, mobil dapat menggunakan power up tersebut sesuai dengan cara pakainya dengan tujuan mencapai garis finish terlebih dahulu.

Fungsi seleksi pada algoritma greedy adalah fungsi yang memilih kandidat berdasarkan strategi greedy tertentu yang bersifat heuristik. Penerapan fungsi seleksi pada permainan Overdrive tidak memiliki satu fungsi seleksi, karena setiap tim memiliki prioritas yang berbeda-beda dalam perancangan strategi optimum mereka. Beberapa contoh prioritas yang dapat digunakan sebagai fungsi seleksi adalah prioritas pengambilan powerup dan prioritas penggunaan powerup.

Fungsi obyektif yang digunakan pada permainan Overdrive ini adalah memaksimumkan kecepatan sehingga dapat mencapai garis finish terlebih dahulu. Selain memaksimumkan kecepatan, fungsi obyektif juga dapat diaplikasikan sehingga dapat memaksimumkan skor dan meminimumkan damage.

B. Eksplorasi Alternatif Solusi Greedy yang Mungkin Dipilih Dalam Persoalan Overdrive

Dalam merancang algoritma sebagai solusi dari permainan Overdrive, terdapat beberapa rancangan algoritma yang dijelaskan lebih lanjut sebagai berikut. Sebagai simplifikasi, setiap penulisan “mobil” memiliki arti “mobil yang dikendalikan bot penulis”.

1. Alternatif Solusi 1

   Alternatif 1 ini tergolong algoritma yang cukup pasif defensif karena perilakunya yang hanya mencari segala cara untuk secepat mungkin dan dengan resiko sekecil mungkin untuk mencapai garis finish tanpa atau dengan usaha yang sangat minimum untuk menghalangi lawan.

Algoritma ini didesain dengan urutan eksekusi Command sebagai berikut :

i. Melakukan pengecekan damage mobil. Bila sama dengan 5, maka akan dilakukan fix. Bila lebih kecil atau sama dengan 4, maka berlanjut ke baris selanjutnya.
ii. Melakukan pengecekan terhadap lane yang akan dilewati. Bila terdapat obstacle seperti oil spill, mud, atau wall, maka mobil akan secara random berbelok ke kiri atau kanan, tentunya dengan memperhatikan kasus out-of-bounds seperti mobil di lane 1 tidak dapat belok kiri dan mobil di lane 4 tidak dapat belok kanan. Namun bila memiliki powerup Lizard, mobil akan menggunakan Lizard.
iii. Bila mobil memiliki boost, digunakan boost
iv. Bila mobil ada pada mode kecepatan tertinggi dan memiliki powerup Oil, maka mobil akan menggunakan Oil. Bila mobil memiliki EMP, maka mobil akan menggunakan EMP.
v. Bila dari kode i sampai iv tidak ada yang dijalankan, maka mobil akan Accelerate.

Alternatif 1 adalah alternatif pertama yang diimplementasikan oleh penulis.

2. Alternatif Solusi 2

Alternatif 2 ini merupakan pengembangan dari alternatif 1, dimana algoritmanya masih tergolong pasif namun jauh lebih agresif jika dibandingkan dengan alternatif 1. Bentuk agresivitasnya juga lebih terukur dan lebih akurat ketimbang alternatif 1.

Algoritma ini didesain dengan urutan eksekusi Command sebagai berikut :

i. Melakukan pengecekan damage mobil. Bila lebih besar atau sama dengan 4, maka akan dilakukan fix. Bila lebih kecil atau sama dengan 3, maka berlanjut ke baris selanjutnya.
ii. Bila mobil memiliki boost, digunakan boost
iii. Melakukan pengecekan terhadap lane yang akan dilewati. Bila terdapat obstacle seperti oil spill, mud, atau wall dan mobil memiliki powerup Lizard, maka mobil akan menggunakan Lizard. Bila tidak, maka mobil akan membandingkan lane di depan mobil dan di sebelah kiri kanannya. Mobil lalu akan memilih lane dengan jumlah obstacles paling sedikit.
iv. Bila mobil memiliki boost, digunakan boost
v. Bila mobil ada pada lane yang sama dengan lawan, berada di depan lawan, dan memiliki powerup Oil, maka mobil akan menggunakan Oil.
vi. Bila mobil memiliki EMP, berada di belakang lawan, dan lawan berada dalam range efektif dari EMP, maka mobil akan menggunakan EMP.
vii. Bila dari kode i sampai vi tidak ada yang dijalankan, maka mobil akan Accelerate.

3. Alternatif Solusi 3

   Alternatif solusi 3 dirancang setelah menetapkan fungsi seleksi berupa prioritas pengambilan keputusan pada setiap langkah. Fungsi seleksi ini didasarkan pada tiga *conditionals* yang mungkin terjadi di dalam balapan: apakah jalur di depan mobil kosong atau tidak, apakah mobil berada di jalur yang sama atau berbeda dengan lawan, dan apakah mobil berada di depan atau di belakang lawan. Penerapan fungsi seleksi ini menghasilkan $2^3 = 8$ kasus yang perlu dianalisa satu per satu. Setelah melakukan pengamatan pada implementasi alternatif solusi 1 dan 2, maka dirancang satu algoritma khusus untuk setiap kasus yang dapat ditemukan oleh mobil.

   Algoritma ini didesain dengan urutan Command sesuai dengan kasus yang ditemukan oleh mobil sebagai berikut:

   i. Jika mobil memiliki damage sama dengan 2 dan tidak sedang dekat garis finish atau memiliki boost, mobil akan melakukan fix. Jika damagenya lebih dari itu, mobil secara otomatis melakukan fix. Bila langkah ini tidak dilakukan, maka dilanjutkan dengan analisa per kasus.

   ii. Kasus jalur di depan mobil kosong, mobil berada di depan lawan, dan mobil berada di lane yang sama dengan lawan

      a. Jika mobil memiliki Oil, mobil akan menjatuhkan Oil untuk menghambat lawan.

      b. Jika mobil memiliki Tweet, mobil akan menggunakan Tweet pada block persis di depan posisi estimasi lawan setelah ronde berakhir (asumsi lawan tidak menggunakan boost atau powerup.

      c. Jika mobil memiliki Boost dengan kondisi mobil tidak dalam kondisi boosting dan tidak ada Wall di lane, mobil akan menggunakan Boost.

      d. Jika langkah a sampai c tidak ada yang memenuhi, secara default program akan menggunakan fungsi antara “avoiding”.

   iii. Kasus jalur di depan mobil kosong, mobil berada di belakang lawan, dan mobil berada di lane yang sama dengan lawan

      a. Jika mobil memiliki EMP dan jarak antar mobil lebih besar dari kecepatan mobil, mobil akan menggunakan EMP. Jika tidak, mobil akan menggunakan fungsi “avoiding” untuk menghindari tabrakan.

      b. Jika mobil memiliki Tweet, mobil akan menggunakan Tweet pada block persis di depan posisi estimasi lawan setelah ronde berakhir (asumsi lawan tidak menggunakan boost atau powerup.

c. Jika mobil memiliki Boost dengan kondisi mobil tidak dalam kondisi boosting, tidak ada Wall di lane, dan tidak akan bertabrakan dengan lawan, mobil akan menggunakan Boost.
d. Jika lajur di depan lane jika mobil melakukan accelerate tidak terdapat Wall, mobil akan Accelerate.
e. Jika langkah a sampai d tidak ada yang memenuhi, secara default program akan menggunakan fungsi antara "avoiding".

iv. Kasus jalur di depan mobil kosong, mobil berada di depan lawan, dan mobil berada di lane yang berbeda dengan lawan
a. Jika mobil memiliki Tweet, mobil akan menggunakan Tweet pada block persis di depan posisi estimasi lawan setelah ronde berakhir (asumsi lawan tidak menggunakan boost atau powerup.
b. Jika lajur mobil dengan lajur lawan berselisih indeks 2, mobil akan membandingkan jalur depan mobil dan jalur yang menjauhi lawan sebagai antisipasi EMP dari lawan.
c. Jika langkah a dan b tidak dapat dikerjakan, secara default program akan menggunakan fungsi antara "avoiding".

v. Kasus jalur di depan mobil kosong, mobil berada di belakang lawan, dan mobil berada di lane yang berbeda dengan lawan
a. Jika mobil memiliki EMP, lawan berada dalam range efektif EMP, dan jarak antar mobil lebih besar dari kecepatan mobil, mobil akan menggunakan EMP. Jika tidak, mobil akan menggunakan fungsi "avoiding" untuk mencoba menyalip mobil lawan.
b. Jika mobil tidak dalam keadaan boosting, memiliki Boost dan jalur mobil ke depan tidak terdapat Wall, mobil akan menggunakan Boost.
c. Jika jalur mobil ke depan tidak terdapat Wall, mobil akan melakukan accelerate.
d. Jika langkah a sampai c tidak dapat dikerjakan, secara default program akan menggunakan fungsi antara "avoiding".

vi. Kasus jalur di depan mobil tidak kosong, tanpa memedulikan posisi dan jalur/lane mobil terhadap lawan
a. Jika damage mobil di bawah 3, lajur mobil di depan tidak terdapat Wall dan lawan berada di depan mobil dan dalam range EMP yang dimiliki mobil, mobil akan menggunakan EMP.

b. Jika mobil memiliki boost namun memungkinkan untuk pindah jalur, mobil akan pindah terlebih dahulu untuk melakukan Boost tanpa gangguan. Bila tidak bisa pindah, mobil akan mengaktifkan Boost di tempat.
c. Jika di depan mobil terdapat halangan, mobil memiliki dua opsi. Menggunakan Lizard bila dimiliki atau menghindari dengan fungsi "avoiding".

vii. Jika dari kasus i sampai vii tidak ada yang dikerjakan, secara default mobil akan melakukan Accelerate.

Dalam perancangan alternatif 3 ini, dibutuhkan beberapa fungsi antara agar rancangan algoritma yang diimplementasikan dapat bekerja dengan baik. Beberapa fungsi tersebut adalah :

a) Avoiding, sebuah fungsi yang membandingkan lane yang dapat dilalui oleh mobil untuk mengambil keputusan terbaik. Perbandingan dilakukan dengan prioritas rintangan dan jumlah power-up. Apabila suatu lane mempunyai jumlah rintangan yang lebih sedikit dibandingkan lane lainnya, maka lane tersebut yang diambil. Bila terdapat dua atau lebih lane yang memiliki jumlah rintangan minimum, maka akan dibandingkan jumlah powerup yang terdapat pada lane-lane tersebut. Lane dengan powerup terbanyak akan diambil.
b) getBlocksInFront, sebuah fungsi untuk mengembalikan semua block yang terdapat pada sebuah lane.
c) hasPowerUp, sebuah fungsi untuk mengecek apakah sebuah powerup dimiliki mobil atau tidak
d) Obstacles, sebuah fungsi yang digunakan untuk menghitung jumlah rintangan pada sebuah lane. Karena Wall dan Cybertruck berdampak besar pada kecepatan dan damage mobil, maka untuk setiap wall atau truck yang terdapat pada lane, counter akan ditambah dengan dua.
e) compareLanes, sebuah fungsi yang dapat memberikan informasi mengenai lane mana yang dapat dilewati dan mana yang invalid seperti mobil di lane 1 tidak dapat belok kiri dan mobil di lane 4 tidak dapat belok kanan.
f) obstacleLandingBlock, sebuah fungsi untuk memprediksi lokasi berhentinya mobil, digunakan sebagai pertimbangan penggunaan Lizard

g) countPowerUps, sebuah fungsi yang menghitung jumlah power up pada sebuah lane.
h) hasCyberTruck, fungsi yang mendeteksi ada atau tidaknya cybertruck pada sebuah lane.
i) compareTwoLanes, fungsi untuk membandingkan dua lane secara murni dari jumlah rintangan pada lane masing-masing.
j) currentMaxSpeed, fungsi untuk mencari kecepatan maksimum sebuah mobil dengan informasi damage yang diterima mobil.
k) nextSpeedState, fungsi untuk mencari speed mobil selanjutnya apabila melakukan accelerate

Fungsi-fungsi antara ini diimplementasikan dan digunakan pada langkah i sampai vii.

C. Analisis Efisiensi dari Kumpulan Alternatif Solusi Greedy yang Dirumuskan

Efisiensi dapat didefinisikan sebagai suatu parameter pada pemrograman yang menggambarkan seberapa cepat dan ringan program tersebut. Efisiensi dapat diukur dengan menggunakan dua metode, kompleksitas waktu (time complexity) dan kompleksitas ruang (space complexity). Kompleksitas waktu dapat dideskripsikan sebagai waktu yang dibutuhkan untuk menjalankan suatu algoritma dengan mempertimbangkan ukuran inputnya. Sementara itu, kompleksitas ruang dapat dideskripsikan sebagai ukuran memori yang dibutuhkan sebuah algoritma untuk berjalan sesuai dengan ukuran inputnya.

Kompleksitas waktu maupun ruang lebih sering dinyatakan dalam notasi Big O. Notasi Big O menggambarkan kasus terbaruk atau worst-case scenario dari kebutuhan memori/waktu suatu algoritma. Beberapa notasi dasar yang umum dipakai adalah O(1) yang menggambarkan kebutuhan waktu/memori yang konstan, O(log N) yang menggambarkan kebutuhan waktu/memori yang berbentuk grafik logaritmik terhadap jumlah input data, O(N) yang menggambarkan kebutuhan waktu/memori yang linear terhadap jumlah input data, O(N log N) yang bertumbuh proporsional dengan ukuran input dan suatu faktor logaritmik, serta O(N^2) yang bertumbuh secara eksponensial berpangkat dua beriringan dengan ukuran input.

Pada semua alternatif solusi algoritma, kompleksitas waktu dapat dilihat dari jenis algoritma yang dipakai. Karena bentuk map adalah array of blocks, maka segala bentuk pertimbangan kompleksitas waktu didasarkan pada cara algoritma memproses array. Berikut adalah tabel yang berisikan kompleksitas waktu setiap operasi sederhana array, asumsi pencarian array menggunakan linear search dan pengurutan array menggunakan quick sort:

No.	Operasi Array	Kompleksitas Waktu Real-time dalam Big O	Kompleksitas Waktu yang Diasumsikan dalam Big O
1	Mengakses elemen ke-i	$O(\sqrt{N})$	$O(1)$
2	Traversal semua elemen array	$O(N + \sqrt{N})$	$O(N)$
3	Mengubah elemen pada indeks ke-i	$O(\sqrt{N})$	$O(1)$
4	Penyisipan elemen	$O(N + \sqrt{N})$	$O(N)$
5	Penghapusan elemen	$O(N + \sqrt{N})$	$O(N)$
6	Linear Search	$O(N + \sqrt{N})$	$O(N)$
7	Quicksort	$O(N \log N + \sqrt{N})$	$O(N \log N)$

Pada alternatif algoritma 1, kompleksitas waktu (yang diasumsikan) untuk setiap langkah, dengan catatan kondisi worst-case selalu digunakan, dijelaskan dalam tabel berikut:

No.	Langkah yang dilakukan	Proses array yang dilakukan	Kompleksitas waktu	Sub-total Kompleksitas Waktu
1	Pengecekan damage mobil	None, pengecekan nilai variabel	O(1)	O(1)
2	Pencarian powerup boost dan menggunakannya bila ada	Linear search untuk mencari powerup, lalu menggunakan boost (penghapusan elemen)	O(N + N)	$O(1 + 2N)$
3	Pengecekan rintangan pada lane, lalu secara random memilih lane bila ada rintangan	Linear search untuk mencari obstacle, integer randomizer	$O(N + 1)$	$O(2 + N)$
4	Mencari dan menggunakan Oil (atau EMP) jika punya dan jika pada kecepatan maksimum	Pengecekan variabel, linear search sebanyak dua kali dan delesi elemen satu kali	$O(2N + N)$	$O(2+8N)$
5	Mobil akan Accelerate	None	O(1)	O(3 + 8N)
Total Kompleksitas Waktu			$O(3+8N)$ = O(N)	

Pada nomor 4, dilakukan linear search sebanyak dua kali karena digunakan asumsi worst-case, dimana worst case pada tahap ini adalah hanya ada EMP dan tidak ada oil, sehingga

iterasi pertama akan menghasilkan false. Iterasi kedua akan mengembalikan true dan powerup EMP akan didelesi.

Secara total, kompleksitas waktu untuk alternatif solusi algoritma 1 adalah O(N).

Pada alternatif algoritma 2, kompleksitas waktu (yang diasumsikan) untuk setiap langkah, dengan catatan kondisi worst-case selalu digunakan, dijelaskan dalam tabel berikut:

No.	Langkah yang dilakukan	Proses array yang dilakukan	Kompleksitas waktu	Sub-total Kompleksitas Waktu
1	Pengecekan damage mobil	None, pengecekan nilai variabel	O(1)	O(1)
2	Mencari boost dan menggunakan boost bila ada	Linear search untuk mencari powerup boost, lalu menggunakan boost (penghapusan elemen)	O($N + N$)	O($1 + 3N$)
3	Pencarian rintangan pada lane. Jika ada rintangan, maka akan membandingkan jumlah rintangan pada lane kanan dan kiri.	Linear search untuk mencari rintangan pada lane, maksimum dilakukan tiga kali)	O($3N$)	O($1 + 5N$)
4	Membandingkan lane mobil dengan mobil lawan, jika sama akan menjatuhkan Oil jika punya powerup tersebut	Pengecekan variabel, linear search sebanyak satu kali dan delesi elemen satu kali	O($N + N$)	O($1 + 5N$)
5	Membandingkan posisi mobil dengan mobil lawan, jika memiliki EMP dan memungkinkan untuk memakai EMP, akan digunakan	Pengecekan variabel, linear search sebanyak satu kali dan delesi elemen satu kali	O($N + N$)	O($1 + 7N$)
6	Mobil akan Accelerate	None	O(1)	O($2 + 7N$)
Total Kompleksitas Waktu			O($2 + 7N$) = O(N)	

Secara total, kompleksitas waktu untuk alternatif solusi algoritma 1 adalah O(N).

Pada alternatif algoritma 3, kompleksitas waktu (yang diasumsikan) untuk setiap langkah, dengan catatan kondisi worst-case selalu digunakan, sangat bergantung pada fungsi antara yang digunakan. Berikut adalah kompleksitas waktu untuk masing-masing fungsi antara:

No.	Fungsi Antara	Aksi yang Dilakukan	Kompleksitas Waktu
1	Avoiding	Memilih lane yang akan dilewati berdasarkan jumlah rintangan dan jumlah powerup (2 x 5 iterasi lane, array of blocks)	O(2 x 5N) = O(10N) = O(N)
2	getBlocksInFront	Mengembalikan semua block pada lane (iterasi lane)	O(N)
3	hasPowerUp	Mengecek apakah suatu powerup tersedia atau tidak pada mobil (iterasi array)	O(N)
4	Obstacles	Mengecek jumlah rintangan pada suatu lane (iterasi lane)	O(N)
5	compareLanes	Mengecek mobil bisa berbelok ke kiri/kanan	O(1)
6	obstacleLandingBlock	Mengecek terrain tempat mobil mendarat setelah bergerak pada satu ronde	O(1)
7	countPowerUps	Menghitung jumlah powerup pada sebuah lane (iterasi lane)	O(N)
8	hasCyberTruck	Mendeteksi ada tidaknya cybertruck pada sebuah lane (linear search)	O(N)
9	compareTwoLanes	Membandingkan dua lane secara murni dari jumlah rintangan (dua iterasi lane dan satu operasi perbandingan)	O(2N + 1)
10	currentMaxSpeed	Mencari kecepatan maksimum sebuah mobil dari informasi damage (operasi perbandingan)	O(1)
11	nextSpeedState	Mencari speed mobil selanjutnya bila melakukan accelerate (operasi perbandingan)	O(1)

Dari fungsi-fungsi antara tersebut, dapat diprediksi dan dihitung kompleksitas waktu dari alternatif solusi 3 sebagaimana berikut:

No.	Langkah yang dilakukan	Proses array yang dilakukan atau fungsi antara	Kompleksitas waktu	Sub-total Kompleksitas Waktu
1	Pengecekan damage mobil	None, pengecekan nilai variabel	O(1)	O(1)
2	Pengecekan kasus depan mobil kosong atau ada rintangan	Obstacles	O(N)	O(1 + N)
3	Pengecekan kasus apakah mobil berada di depan atau di belakang lawan	Operasi perbandingan	O(1)	O(2 + N)
4	Pengecekan kasus apabila mobil berada di lane yang sama atau berbeda dengan lawan	Operasi perbandingan	O(1)	O(3 + N)
5	Validasi penggunaan Oil	hasPowerUp, operasi perbandingan, penghapusan elemen	O(1 + N)	O(4 + 2N)
6	Validasi penggunaan EMP	hasPowerUp, operasi perbandingan, penghapusan elemen	O(1 + N)	O(5 + 3N)
7	Validasi penggunaan Boost tanpa Wall di lane depan	hasPowerUp, operasi perbandingan (2), Obstacles, penghapusan elemen	O(2 + 3N)	O(7 + 6N)
8	Fungsi menghindari rintangan atau Avoiding	Avoiding	O(N)	O(7 + 7N)
9	Validasi penggunaan EMP	hasPowerUp, operasi perbandingan, penghapusan elemen	O(2N + 1)	O(8 + 9N)
10	Validasi penggunaan	Operasi perbandingan,	O(2N + 1)	O(9 + 11N)

	Accelerate tanpa Wall	Obstacles, penghapusan elemen		
11	Validasi penggunaan Lizard	Obstacles, hasPowerUp, penghapusan elemen	O(3N)	O(14N + 9)
Total Kompleksitas Waktu			O(9+14N) = O(N)	

Maka, alternatif solusi tiga memiliki kompleksitas waktu O(N) dalam notasi Big O.

Perlu diperhatikan bahwa penjabaran kompleksitas waktu pada alternatif solusi tiga hanya "daun" atau langkah terakhir yang memberikan output Command dan bukan conditionals. Arti "daun" di sini umum dipakai dalam analisis pohon sebagai graf berarah. Bila algoritma alternatif ketiga ini dibuat menjadi sebuah pohon, pohon memiliki tinggi yang tetap sepanjang keberjalanan permainan, atau setinggi *k*. Maka, kompleksitas waktunya menjadi O(*k*N). Karena *k* adalah sebuah bilangan koefisien, maka hasil akhirnya adalah O(N).

Untuk alternatif 1, 2, atau 3, semuanya memiliki kompleksitas ruang O(N). Ini diakibatkan dari awal sampai akhir permainan, program akan memproses map dengan ukuran tetap berukuran 4 x N. Karena ukuran map tidak berubah selama keberjalanan program dan ukuran map bergantung dari seed dan game-config, maka kompleksitas ruang bot.java secara keseluruhan adalah O(N).

Dengan kata lain, secara garis besar ketiga alternatif algoritma menggunakan notasi Big O yang sama, yaitu O(N) dengan N adalah ukuran inputnya. Satu-satunya perbedaan adalah pada koefisien, dimana algoritma alternatif 3 adalah algoritma dengan koefisien terbesar. Besar kecilnya nilai koefisien ini tidak terlalu membedakan seperti berbeda notasi Big O, namun tetap dapat mengurangi efisiensi meskipun hanya sebagian kecil.

D. Analisis Efektivitas dari Kumpulan Alternatif Solusi Greedy yang Dirumuskan

Efektivitas dapat dideskripsikan sebagai seberapa akurat dan presisi suatu algoritma menyelesaikan persoalan yang sebelumnya telah didefinisikan. Salah satu metode untuk mengukur efektivitas suatu algoritma adalah dengan menggunakan parameter kesuksesan solusi permasalahan.

Pada algoritma greedy, parameter ini adalah fungsi obyektif. Fungsi obyektif ini bertujuan untuk memaksimumkan kecepatan, meminimumkan waktu yang diperlukan untuk mencapai garis finish dan memaksimumkan skor. Maka dari itu, untuk menguji efektivitas algoritma yang sebelumnya telah didesain, dibutuhkan suatu test run untuk melihat kemampuan masing-masing alternatif.

Berikut adalah tabel yang berisikan data kemenangan, skor, dan kecepatan akhir mobil implementasi alternatif algoritma 1 sampai 3. Data didapatkan dengan mengadu mobil implementasi algoritma dengan reference-bot dari starter-pack.

Seed	Alternatif 1		Alternatif 2		Alternatif 3	
	Speed	Skor Akhir	Speed	Skor Akhir	Speed	Skor Akhir
69	3	13	6	92	9	130
420	3	-17	6	26	9	88
69420	3	-25	6	23	9	117
42069	0	-5	3	14	6	97
11111	6	-11	6	35	15	102
Winrate	100%		100%		100%	
Rata - rata	3	-9	5.4	38	9.6	106.8

Dapat dilihat bahwa alternatif 3 adalah alternatif terbaik karena selain memiliki win rate yang sempurna seperti alternatif lainnya, alternatif 3 juga memiliki rata-rata kecepatan akhir dan rata-rata skor akhir tertinggi.

Berikut adalah kelemahan algoritma alternatif 1 yang menyebabkannya menjadi tidak efektif:

i. Rendahnya penggunaan powerup seperti Tweet dan EMP, karena selain dapat meningkatkan kemungkinan lawan terhambat, mobil akan mendapatkan skor tambahan juga.
ii. Rendahnya penggunaan Command Accelerate, yang kemungkinan besar berkontribusi terhadap rendahnya kecepatan akhir
iii. Fungsi mengelak dari rintangan di depan mobil yang bersifat random. Fungsi ini dapat menjadi boomerang karena bisa saja mobil berkelok ke lane yang lebih banyak rintangannya.

Berikut adalah kelemahan algoritma alternatif 2 yang menyebabkannya menjadi tidak efektif:

i. Penggunaan Boost dan Accelerate tidak mempertimbangkan apabila lane di depannya memungkina terjadinya tabrakan atau tidak
ii. Belum lengkapnya parameter penilaian untuk fungsi berkelit menyebabkan adanya kemungkinan mobil melewati lane yang memiliki banyak powerup.
iii. Belum ada pembobotan rintangan dan powerup. Seperti Wall yang damagenya lebih fatal jika bertabrakan ketimbang bertabrakan dengan Mud atau Oil Spill.

Berikut adalah kelebihan algoritma alternatif 2 jika dibandingkan dengan alternatif 1:

i. Lebih dimanfaatkan powerup agresif seperti EMP dan Tweet
ii. Fungsi mengelak yang lebih banyak parameternya sehingga tidak akan secara random menabrak rintangan yang tidak terdeteksi

Berikut adalah kelebihan algoritma alternatif 3 secara keseluruhan dan jika dibandingkan dengan alternatif 1 dan 2:

i. Lebih banyak parameter pada algoritma sehingga memungkinkan bot mobil untuk mengambil keputusan terbaik setiap saat.
ii. Lebih banyak opsi untuk berkelit, didasarkan pada banyaknya rintangan dan banyaknya powerup pada lane yang dibandingkan
iii. Terdapat algoritma untuk antisipasi serangan EMP dari lawan dengan cara menjauh satu lane dari posisi lane lawan.

iv. Terdapat algoritma untuk menghindari serangan Cybertruck dari lawan
v. Penggunaan Boost dan Accelerate hanya valid bila setelah pemakaian, mobil tidak akan menabrak rintangan apapun.
vi. Penggunaan Lizard juga mempertimbangkan block tempat mobil akan mendarat. Alasannya adalah setelah mobil mendarat, mobil tidak akan terkena damage.

E. Strategi Greedy yang Dipilih untuk Permainan Overdrive

Menilai dari efisiensi, tidak ada perbedaan signifikan antara alternatif 1, 2, maupun 3. Semunya sama-sama memproses array dengan ukuran yang sama saat permainan dimulai. Namun, jika melihat dari efektivitas, terlihat jelas bahwa alternatif 3 unggul secara skor, tingkat kemenangan, maupun kecepatan akhir saat melewati garis finish. Maka dari itu, penulis memilih alternatif solusi 3 untuk diimplementasikan dan dikumpulkan sebagai solusi akhir dari permainan Overdrive.

BAB 4
IMPLEMENTASI DAN PENGUJIAN

A. Pseudocode Algoritma *Greedy*

```
/*
    Fungsi run dipanggil di Main.java sebagai basis pengambilan keputusan. Keputusan yang
diambil adalah
    keputusan terbaik yang dapat diambil di dalam suatu ronde, tanpa mempertimbangkan keadaan
ronde sebelumnya
    atau ronde setelahnya. Hal ini dilakukan sebagai implementasi algoritma greedy, yang
mengambil keputusan
    yang bersifat optimum lokal (keputusan terbaik pada suatu saat)
     */

    public Command run() {
        // inisialisasi variabel dari gameState
        Car myCar = gameState.player;
        Car opponent = gameState.opponent;

        /*
        Pemanggilan fungsi getBlocksInFront untuk mendapatkan list of lanes pada lane kiri,
kanan, dan tengah.
        Inisialisasi lane kiri dan/atau kanan dilakukan apabila memungkinkan, yaitu pada
kondisi apabila
        lokasi lane mobil di antara 1 dan 4 (tidak di kiri ataupun di kanan).
         */

        List <Object> currentLane = h.getBlocksInFront(myCar.position.lane,
myCar.position.block, myCar.speed);
        List <Object> leftLane = currentLane;
        List <Object> rightLane = currentLane;
        if (myCar.position.lane > 1){
            leftLane = h.getBlocksInFront(myCar.position.lane - 1, myCar.position.block,
myCar.speed);
            leftLane.remove(leftLane.size() - 1);
        }
        if (myCar.position.lane < 4){
            rightLane = h.getBlocksInFront(myCar.position.lane + 1, myCar.position.block ,
myCar.speed);
            rightLane.remove(rightLane.size() - 1);
        }
        if (currentLane.size() != 0) {
            currentLane.remove(0);
        }


        /*
        Variabel choice digunakan sebagai penanda kemungkinan lane yang dapat diakses mobil.
         */

        String choice = h.compareLanes();
        /*
        Algoritma mengutamakan perbaikan mobil apabila damage mobil sama dengan atau lebih
besar dari 2
        dan ada power-up boost, sehingga mobil dapat mencapai kecepatan tertinggi. Fix juga
akan dilakukan
        apabila mobil mempunyai damage di atas 2.
        Selain itu, mobil juga akan mengutamakan switching lane
         */

        if (myCar.damage >= 2 && (!nearFinish(currentLane, myCar) ||
h.hasPowerUp(PowerUps.BOOST, myCar.powerups))){
            return FIX;
        } else if (myCar.damage > 2) {
            return FIX;
        }

        if (myCar.speed <= 3 && !nearFinish(currentLane, myCar)){
            if (h.Obstacles(currentLane, 0) <= 1 && myCar.damage == 0) {
                return avoiding(choice, currentLane, leftLane, rightLane);
            }
        }
```

```
        /*
        Algoritma dibagi menjadi dua segmen, yaitu apabila lane sedang kosong dan tidak
kosong.
        Algoritma pada lane kosong dibagi menjadi algoritma saat musuh berada dalam lane yang
sama
        atau berbeda.
         */

        if (h.Obstacles(currentLane, 0) == 0){
            if (myCar.position.lane == opponent.position.lane){
                // pemanggilan fungsi sameLaneCommand(...) untuk lane musuh == lane mobil
                return sameLaneCommand(choice, myCar, opponent, currentLane, leftLane,
rightLane);
            } else {
                // pemanggilan fungsi diffLaneCommand(...) untuk lane musuh != lane mobil;=
                return diffLaneCommand(choice, myCar, opponent, currentLane, leftLane,
rightLane);
            }
        } else {
            /*
            Mobil akan menggunakan EMP apabila memungkinkan. Untuk menghindari collison, mobil
dapat melakukan
            manuver berpindah ke kiri / kanan.
             */
            if (myCar.damage < 3 && h.Obstacles(currentLane, 0) < 3
                    && h.hasPowerUp(PowerUps.EMP, myCar.powerups)
                    && Math.abs(myCar.position.lane - opponent.position.lane) <= 1
                    && opponent.position.block > myCar.position.block) {
                if (Math.abs(myCar.position.block - opponent.position.block) > myCar.speed) {
                    return USE_EMP;
                } else {
                    return avoiding("LEFT_RIGHT", currentLane, leftLane, rightLane);
                }
            }

            /*
            Apabila ada powerup boost namun memungkinkan untuk pindah, mobil akan pindah
terlebih dahulu. Jika tidak
            baru boost digunakan.
             */

            if (h.hasPowerUp(PowerUps.BOOST, myCar.powerups)
                    && h.Obstacles(currentLane, 0) < 2
                    && !myCar.boosting) {
                if (h.Obstacles(leftLane, -1) <= h.Obstacles(currentLane, 0)
                        || h.Obstacles(rightLane, 1) <= h.Obstacles(currentLane, 0)) {
                    return avoiding(choice, currentLane, leftLane, rightLane);
                }
                return USE_BOOST;
            }

            /*
            Algoritma juga akan menangani kasus apabila ada rintangan. Apabila memungkinkan
untuk menggunakan
            power-up LIZARD dan blok tujuan bukanlah blok rintangan, maka mobil akan
menggunakan perintah
            USE_LIZARD. Sebaliknya, mobil akan berusaha menghindar.
             */

            if (currentLane.contains(Terrain.MUD) || currentLane.contains(Terrain.WALL)
                    || currentLane.contains(Terrain.OIL_SPILL) || h.hasCyberTruck(0) != -1) {
                if (h.hasPowerUp(PowerUps.LIZARD, myCar.powerups) &&
h.obstacleLandingBlock(currentLane) == 0) {
                    return USE_LIZARD;
                } else {
                    return avoiding(choice, currentLane, leftLane, rightLane);
                }
            }

        }
        return ACCELERATE;
    }



    // Fungsi validasi apakah mobil berada di dekat garis finis
```

```
    private boolean nearFinish (List<Object> CurrLane, Car myCar) {
        int i = 0;
        boolean finish = false;
        while (i < CurrLane.size() && !finish) {
            if (CurrLane.get(i).equals(Terrain.FINISH)) {
                finish = true;
            } else {
                i++;
            }
        }
        return finish;
    }

    // fungsi untuk melakukan algoritma jika mobil musuh berada di lane yang sama
    private Command sameLaneCommand(String choice, Car myCar, Car opponent, List<Object>
currentLane,
                                    List <Object> pNextBlockLeft, List <Object>
pNextBlockRight){
        int with_accelerate = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block,  h.nextSpeedState(myCar)), 0);

        /*
        Apabila mobil berada di depan mobil musuh, mobil akan berusaha untuk bermain agresif
        dengan menggunakan power-ups yang dimiliki. Jika tidak, mobil akan menggunakan power-
up yang dimiliki
        untuk memperlambat mobil musuh
         */
        if (myCar.position.block >= opponent.position.block) {
            if (h.hasPowerUp(PowerUps.OIL, myCar.powerups)) {
                return USE_OIL;
            }
            if (h.hasPowerUp(PowerUps.TWEET, myCar.powerups)){
                return new TweetCommand(opponent.position.lane, opponent.position.block +
opponent.speed + 1);
            }

            if (h.hasPowerUp(PowerUps.BOOST, myCar.powerups) && h.Obstacles(currentLane, 0) <
2 && !myCar.boosting) {
                return USE_BOOST;
            }
        } else {
            if (h.hasPowerUp(PowerUps.EMP, myCar.powerups) && (opponent.position.block >
myCar.position.block)) {
                if (Math.abs(myCar.position.block - opponent.position.block) > myCar.speed) {
                    return USE_EMP;
                } else {
                    return avoiding("LEFT_RIGHT", currentLane, pNextBlockLeft,
pNextBlockRight);
                }
            }

            if (h.hasPowerUp(PowerUps.TWEET, myCar.powerups)){
                return new TweetCommand(opponent.position.lane, opponent.position.block +
opponent.speed + 1);
            }

            if (h.hasPowerUp(PowerUps.BOOST, myCar.powerups) && h.Obstacles(currentLane, 0) <
2
                    && (opponent.position.block - myCar.position.block) > 15 &&
!myCar.boosting) {
                return USE_BOOST;
            }

            if (with_accelerate < 2) {
                return ACCELERATE;
            }
        }
        return avoiding(choice, currentLane, pNextBlockLeft, pNextBlockRight);
    }


    private Command diffLaneCommand(String choice, Car myCar, Car opponent, List<Object>
currentLane,
                                    List <Object> pNextBlockLeft, List <Object>
pNextBlockRight){
        int with_accelerate = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block,  h.nextSpeedState(myCar)), 0);
```

```
        int with_boost = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block, h.currentMaxSpeed(myCar)), 0);

        /*
        Apabila mobil berada di depan musuh, mobil akan mengantisipasi EMP yang akan
ditembakkan musuh dan mencoba
        menghindar dari obstacle. Sebaliknya, mobil juga akan berusaha untuk memperlambat
musuh dengan
        power-up yang dimiliki.
         */
        if (myCar.position.block >= opponent.position.block) {
            if (h.hasPowerUp(PowerUps.TWEET, myCar.powerups)){
                return new TweetCommand(opponent.position.lane, opponent.position.block +
opponent.speed + 1);
            }
            // buat antisipasi EMP
            if (Math.abs(myCar.position.lane - opponent.position.lane) == 2 &&
(myCar.position.lane != 1 || myCar.position.lane != 4)) {
                switch (myCar.position.lane) {
                    case 2:
                        return h.compareTwoLanes(currentLane, pNextBlockLeft, -1,
trackLength);
                    case 3:
                        return h.compareTwoLanes(currentLane, pNextBlockRight, 1,
trackLength);
                    default:
                        break;
                }
            }

            if (currentLane.contains(Terrain.MUD) || currentLane.contains(Terrain.WALL)
                    || currentLane.contains(Terrain.OIL SPILL) || h.hasCyberTruck(0) != -1) {
                if (h.hasPowerUp(PowerUps.LIZARD, myCar.powerups) &&
h.obstacleLandingBlock(currentLane) == 0) {
                    return USE_LIZARD;
                } else {
                    return avoiding(choice, currentLane, pNextBlockLeft, pNextBlockRight);
                }
            }
        } else {
            if (h.hasPowerUp(PowerUps.EMP, myCar.powerups)){
                if (opponent.position.lane <= myCar.position.lane + 1 &&
opponent.position.lane >= myCar.position.lane - 1
                        && opponent.position.block > myCar.position.block){
                    if (Math.abs(myCar.position.block - opponent.position.block) >
myCar.speed) {
                        return USE_EMP;
                    } else {
                        return avoiding("LEFT_RIGHT", currentLane, pNextBlockLeft,
pNextBlockRight);
                    }
                }
            }
            if (h.hasPowerUp(PowerUps.BOOST, myCar.powerups) && with_boost < 2 &&
!myCar.boosting && Math.abs(myCar.position.block - opponent.position.block) <
h.currentMaxSpeed(myCar)) {
                return USE_BOOST;
            }
            if (with_accelerate < 2) {
                return ACCELERATE;
            }
        }
        return avoiding(choice, currentLane, pNextBlockLeft, pNextBlockRight);
    }

    /*
        Algoritma pembanding lane untuk mengambil keputusan terkait lane terbaik yang dapat
diambil.
        Perbandingan dilakukan dengan prioritas rintangan dan jumlah power-up. Apabila suatu
lane mempunyai
        jumlah rintangan yang lebih sedikit dibandingkan lane lainnya, maka lane tersebut akan
diambil. Apabila
        lane tersebut mempunyai jumlah rintangan yang sama dengan lane lainnya, maka lane
dengan jumlah power-up
        terbanyak yang akan diambil. Perbandingan dilakukan berdasarkan string choice dengan
aturan:
        - CURR_LEFT = membandingkan lane yang ditempati dengan lane kiri
```

```
        - CURR_RIGHT = membandingkan lane yang ditempati dengan lane kanan
        - ALL = membandingkan lane kiri, kanan, dan lane yang ditempati
        - LEFT_RIGHT = membandingkan lane kiri dan kanan (dipanggil untuk menghindari
collision saat EMP
        - STAY = tetap di lane
     */
    private Command avoiding(String choice, List <Object> pNextBlock, List <Object>
pNextBlockLeft, List <Object> pNextBlockRight){
        int no_accelerate = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block, myCar.speed), 0);
        int with_accelerate = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block,  h.nextSpeedState(myCar)), 0);
        int with_boost = h.Obstacles(h.getBlocksInFront(myCar.position.lane,
myCar.position.block, h.currentMaxSpeed(myCar)), 0);
        int leftObstacleCount = 100;
        int leftPowerUpCount = 0;

        int rightObstacleCount = 100;
        int rightPowerUpCount = 0;

        int currObstacleCount = h.Obstacles(pNextBlock, 0);
        int currPowerUpCount = h.countPowerUps(pNextBlock);

        if (myCar.position.lane > 1) {
            leftObstacleCount = h.Obstacles(pNextBlockLeft, -1);
            leftPowerUpCount = h.countPowerUps(pNextBlockLeft);
        }
        if (myCar.position.lane < 4) {
            rightObstacleCount = h.Obstacles(pNextBlockRight, 1);
            rightPowerUpCount = h.countPowerUps(pNextBlockRight);
        }

        switch (choice) {
            case "CURR_LEFT":
                if ((currObstacleCount < leftObstacleCount)
                        || ((currObstacleCount == leftObstacleCount) && (currPowerUpCount >=
leftPowerUpCount))){
                    if (with_accelerate <= no_accelerate) {
                        if (!myCar.boosting && h.hasPowerUp(PowerUps.BOOST, myCar.powerups) &&
with_boost <= with_accelerate) {
                            return USE_BOOST;
                        } else {
                            return ACCELERATE;
                        }
                    } else {
                        if (myCar.speed == 0) {
                            return ACCELERATE;
                        } else {
                            return NOTHING;
                        }
                    }
                } else {
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_LEFT;
                    }
                }
            case "CURR_RIGHT":
                if ((currObstacleCount < rightObstacleCount)
                        || ((currObstacleCount == rightObstacleCount)
                        && (currPowerUpCount >= rightPowerUpCount))){
                    if (with_accelerate <= no_accelerate) {
                        if (!myCar.boosting && h.hasPowerUp(PowerUps.BOOST, myCar.powerups) &&
with_boost <= with_accelerate) {
                            return USE_BOOST;
                        } else {
                            return ACCELERATE;
                        }
                    } else {
                        if (myCar.speed == 0) {
                            return ACCELERATE;
                        } else {
                            return NOTHING;
                        }
                    }
                } else {
```

```
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_RIGHT;
                    }
                }
            case "LEFT_RIGHT":
                if (leftObstacleCount < rightObstacleCount
                || ((leftObstacleCount == rightObstacleCount) && (leftPowerUpCount >=
rightPowerUpCount))){
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_LEFT;
                    }
                } else {
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_RIGHT;
                    }
                }
            case "ALL":
                if ((currObstacleCount < Math.min(leftObstacleCount, rightObstacleCount)
                        || ((currObstacleCount == Math.min(leftObstacleCount,
rightObstacleCount))
                        && currPowerUpCount >= Math.max(leftPowerUpCount,
rightPowerUpCount)))) {
                    if (with_accelerate <= no_accelerate) {
                        if (!myCar.boosting && h.hasPowerUp(PowerUps.BOOST, myCar.powerups) &&
with_boost <= with_accelerate) {
                            return USE BOOST;
                        } else {
                            return ACCELERATE;
                        }
                    } else {
                        if (myCar.speed == 0) {
                            return ACCELERATE;
                        } else {
                            return NOTHING;
                        }
                    }
                }

                if ((leftObstacleCount < Math.min(currObstacleCount, rightObstacleCount)
                        || ((leftObstacleCount == Math.min(currObstacleCount,
rightObstacleCount))
                        && (leftPowerUpCount >= Math.max(currPowerUpCount,
rightPowerUpCount))))) {
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_LEFT;
                    }
                }

                if ((rightObstacleCount < Math.min(currObstacleCount, leftObstacleCount))
                        || ((rightObstacleCount == Math.min(currObstacleCount,
leftObstacleCount))
                        && (rightPowerUpCount >= Math.max(currPowerUpCount,
leftPowerUpCount)))) {
                    if (myCar.speed == 0) {
                        return ACCELERATE;
                    } else {
                        return TURN_RIGHT;
                    }
                }
            case "STAY":
                if (with_accelerate <= no_accelerate) {
                    if (!myCar.boosting && h.hasPowerUp(PowerUps.BOOST, myCar.powerups) &&
with_boost <= with_accelerate) {
                        return USE_BOOST;
                    } else {
                        return ACCELERATE;
                    }
                } else {
                    if (myCar.speed == 0) {
```

```
                        return ACCELERATE;
                    } else {
                        return NOTHING;
                    }
                }
            default:
                if (!myCar.boosting && h.hasPowerUp(PowerUps.BOOST, myCar.powerups) &&
with_boost <= with_accelerate) {
                    return USE_BOOST;
                } else {
                    return ACCELERATE;
                }
        }
    }
}

// fungsi untuk mengembalikan list of blocks
    public List<Object> getBlocksInFront(int lane, int block, int speed) {
        List<Lane[]> map = gameState.lanes;
        List<Object> blocks = new ArrayList<>();
        Lane[] laneList = map.get(lane - 1);

        int startBlock = map.get(0)[0].position.block;

        for (int i = max(block - startBlock, 0); i <= block - startBlock + 15 ; i++) {
            if (laneList[i] == null || laneList[i].terrain == Terrain.FINISH) {
                break;
            }
            blocks.add(laneList[i].terrain);
        }
        return blocks;
    }

    // fungsi validasi power-up ada di list power-up mobil
    public Boolean hasPowerUp(PowerUps powerUpToCheck, PowerUps[] available) {
        for (PowerUps powerUp: available) {
            if (powerUp.equals(powerUpToCheck)) {
                return true;
            }
        }
        return false;
    }

    // fungsi untuk menghitung damage total dari rintangan

    public int Obstacles(List<Object> Lane, int flag) {
        int count = 0;
        for (int i = 0; i < Lane.size(); i++) {
            if (Lane.get(i).equals(Terrain.MUD) ||
                    Lane.get(i).equals(Terrain.OIL_SPILL)){
                count++;
            } else if (Lane.get(i).equals(Terrain.WALL)){
                count += 2;
            }
        }
        if (hasCyberTruck(flag) >= 0) {
            count += 2;
        }
        return count;
    }

    // fungsi untuk mengembalikan string untuk pemilihan avoiding
    public String compareLanes(){
        boolean LPos = false; boolean RPos = false;
        if (myCar.position.lane != 1) {
            LPos = true;
        }
        if (myCar.position.lane != 4){
            RPos = true;
        }
        if (LPos){
            return (RPos ? "ALL" : "CURR_LEFT");
        } else {
            return (RPos ? "CURR_RIGHT" : "DEFAULT");
        }
    }
```

```
    // fungsi untuk memprediksi lokasi berhentinya mobil, digunakan
    // sebagai pertimbangan penggunaan fungsi lizard
    public int obstacleLandingBlock(List<Object> pNextBlock) {
        int landingPosition = 0;
        if (myCar.speed == 15 || pNextBlock.size() < myCar.speed){
            landingPosition = pNextBlock.size() - 1;
        } else {
            landingPosition = myCar.speed - 1;
        }
        if (myCar.speed > 0) {
            if (pNextBlock.get(landingPosition).equals(Terrain.OIL_SPILL)) {
                return 1;
            } else if (pNextBlock.get(landingPosition).equals(Terrain.MUD)) {
                return 2;
            } else if (pNextBlock.get(landingPosition).equals(Terrain.WALL)) {
                return 3;
            } else {
                return 0;
            }
        } else {
            return  0;
        }
    }

    // fungsi untuk menghitung jumlah power-ups dalam suatu lane, digunakan untuk
    // membandingkan jumlah power-ups antar lane

    public int countPowerUps(List<Object> laneList){
        int count = 0;
        for (int i = 0; i < laneList.size(); i++) {
            if (laneList.get(i).equals(Terrain.OIL_POWER) ||
                    laneList.get(i).equals(Terrain.EMP) ||
                    laneList.get(i).equals(Terrain.BOOST) ||
                    laneList.get(i).equals(Terrain.LIZARD) ||
                    laneList.get(i).equals(Terrain.TWEET)) {
                count++;
            }
        }
        return count;
    }

    // fungsi yang mengembalikan integer yang menandai adanya cybertruck dalam lane
    // return -1 = tidak ada, return -999 = lane tidak bisa diakses, lainnya = ada cybertruck
    public int hasCyberTruck(int flag) {
        List<Lane[]> map = gameState.lanes;
        List<Object> blocks = new ArrayList<>();
        int startBlock = map.get(0)[0].position.block;
        int lane = this.myCar.position.lane + flag;
        int block = this.myCar.position.block;

        if (lane + flag <= 4 && lane + flag >= 1) {
            Lane[] laneList = map.get(lane - 1);
            for (int i = max(block - startBlock, 0); i <= block - startBlock + 15; i++) {
                if (laneList[i] == null || laneList[i].terrain == Terrain.FINISH) {
                    break;
                }
                if (laneList[i].isOccupiedByCyberTruck) {
                    return i;
                }
            }
            return -1;
        } else {
            return -999;

        }
    }

    // fungsi untuk membandingkan dua lane murni dari jumlah obstaclenya
    public Command compareTwoLanes(List<Object> CenterLane, List<Object> CompLane, int flag,
int trackLength) {
        int Ccount = Obstacles(CenterLane, 0);
        int Pcount = Obstacles(CompLane, flag);

        if (Pcount < Ccount) {
            if (flag == -1) {
                return TURN LEFT;
            } else {
```

```
                return TURN_RIGHT;
            }
        } else {
            if (hasPowerUp(PowerUps.BOOST, myCar.powerups)) {
                return USE_BOOST;
            } else {
                return ACCELERATE;
            }
        }
    }

    // fungsi untuk melihat speed tertinggi yang dapat diakses
    public int currentMaxSpeed (Car myCar) {
        switch (myCar.damage) {
            case 0:
                return 15;
            case 1:
                return 9;
            case 2:
                return 8;
            case 3:
                return 6;
            case 4:
                return 3;
            case 5:
                return 0;
            default:
                return myCar.speed;
        }
    }

    // fungsi untuk mengembalikan speed state selanjutnya
    // apabila accelerate / boost
    public int nextSpeedState (Car targetCar) {
        switch (targetCar.speed) {
            case 0:
                return 3;
            case 3:
            case 5:
                return 6;
            case 6:
                return 8;
            case 8:
                return 9;
            default:
                return targetCar.speed;
        }
    }
}
```

B. Struktur Data Program Bot

Dalam program bot Overdrive, digunakan beberapa struktur data:

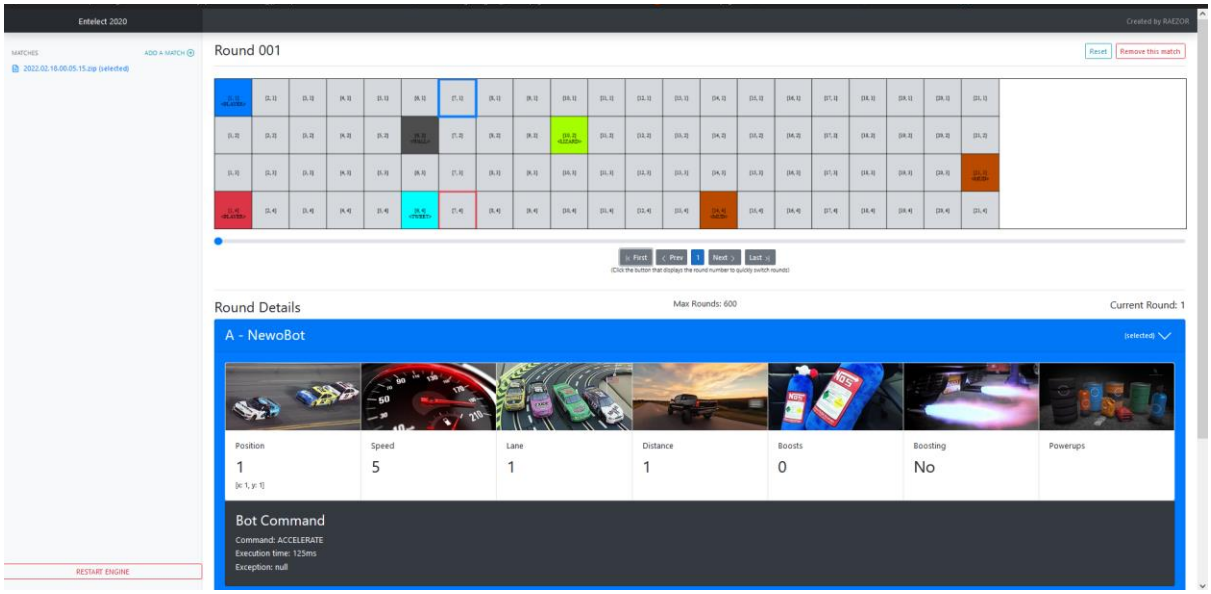
- List <object>
  Sebuah list yang elemennya merupakan objek-objek yang telah didefinisikan. Struktur data ini dapat diperlakukan seperti list yang berisi tipe data primitif, seperti integer, float, dan char. Dalam program kali ini, object yang didefinisikan adalah Car, GameState, Command, dan PowerUps. Beberapa metode yang dimiliki struktur data ini adalah pengaksesan elemen ke-i, penghapusan elemen dalam list, penambahan elemen ke dalam list.
- String
  Struktur data string pada hakikatnya adalah sebuah senarai karakter. String dapat diperlakukan layaknya array atau list biasa. Akan tetapi, terdapat metode yang biasanya

tidak dimiliki array atau list lain, yaitu concatenation. Concatenation atau biasa disebut concat merupakan metode yang menggabungkan dua string.

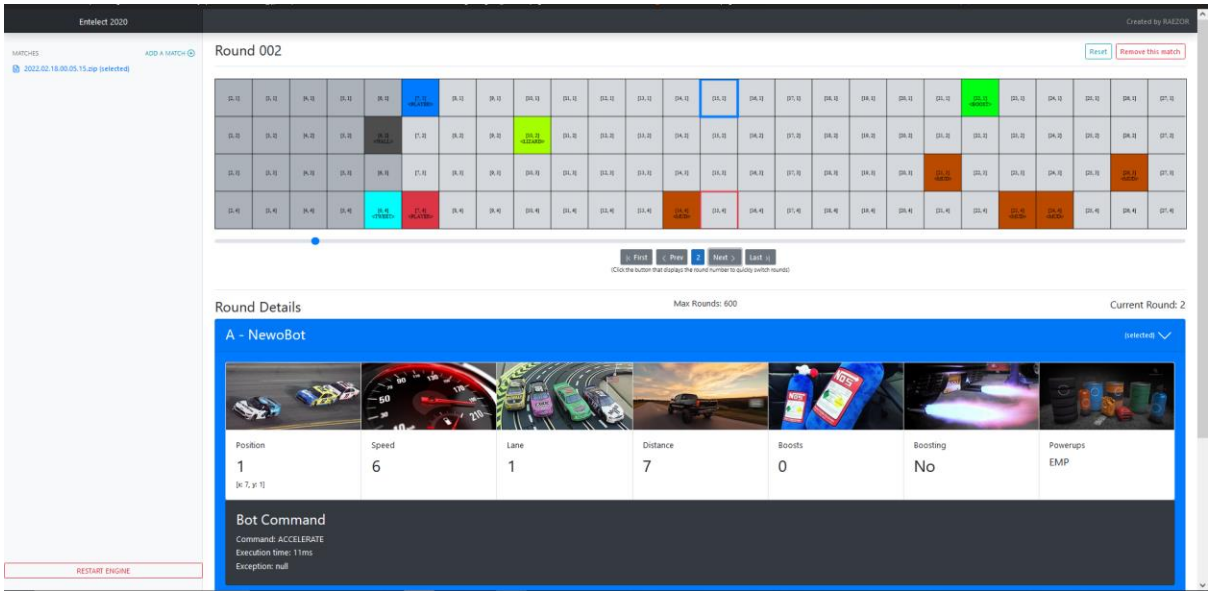
- List <Lane[]>
  Struktur data ini merupakan list yang menyimpan sebuah array yang menyimpan tipe data Lane. Lane dalam program "Overdive" adalah sebuah array yang berisi objek Terrain. Sehingga, struktur data ini mirip dengan matriks, yang merupakan "array of array". Karena struktur data ini mirip matriks, kita perlu memberikan indeks untuk mengakses elemen pada baris ke-i dan kolom ke-j. Fungsi yang dapat diterapkan pada struktur data ini mirip dengan list biasa, hanya saja struktur data ini dua dimensi.

C. Analisis Solusi Algoritma Greedy

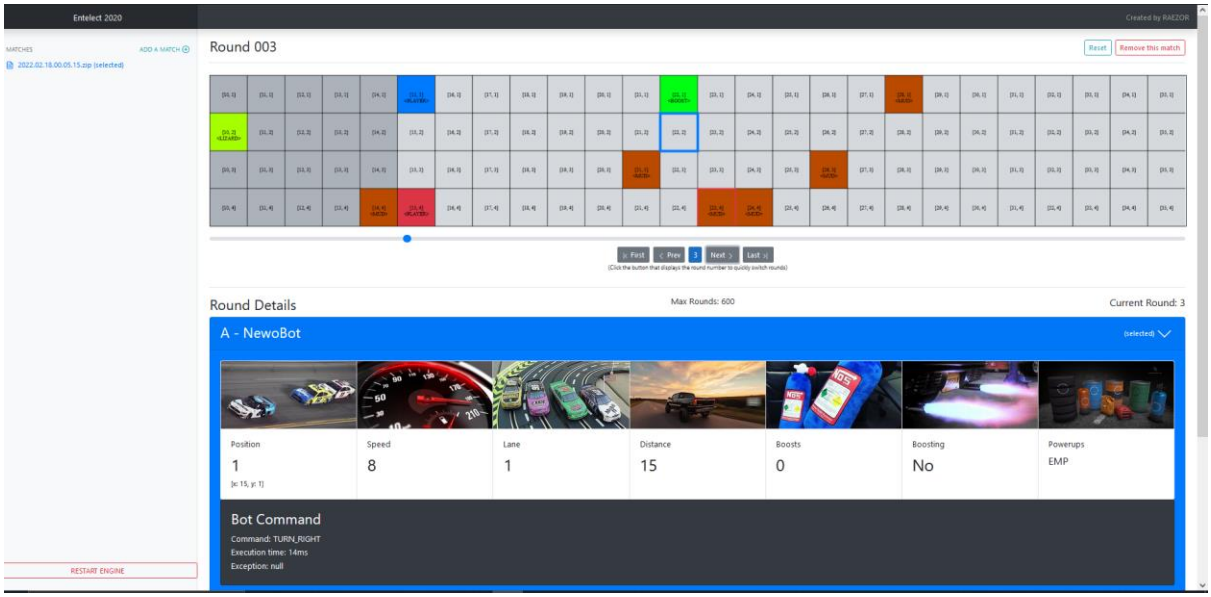
Kami menguji algoritma kami dengan pengaturan panjang trek 100 blok pada seed 18601 dengan pengaturan lain default. Player-a pada game adalah bot kami dan player-b pada game adalah bot referensi. Bot kami beri nama NewoBot. Bot akan mulai dengan kecepatan awal 3. Bot kami akan melakukan pindai pada *lane* yang sedang ditempatinya dan *lane* sebelahnya dengan panjang 15 *block.*



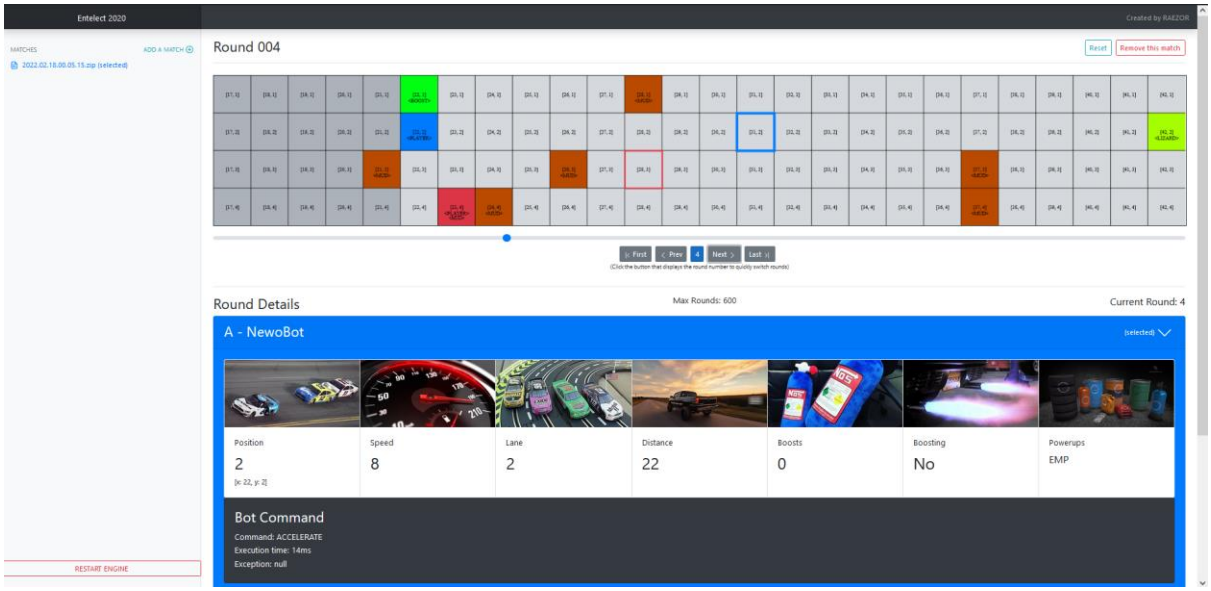
Pada ronde pertama, terlihat bahwa bot kami mengambil keputusan untuk ACCELERATE. Keputusan tersebut merupakan keputusan yang optimal untuk kondisi saat itu. Jika bot TURN_RIGHT, maka bot akan menabrak halangan WALL dan bot akan terkena damage sebesar 2. Akan tetapi, jika bot lurus, tidak ada halangan yang merugikan dan ada dua keputusan yang dapat diambil, yaitu NOTHING atau ACCELERATE. Karena saat itu bot belum berada pada keadaan *maximum speed*, maka keputusan terbaik adalah ACCELERATE supaya kecepatan bertambah dan bot sampai finish lebih cepat.



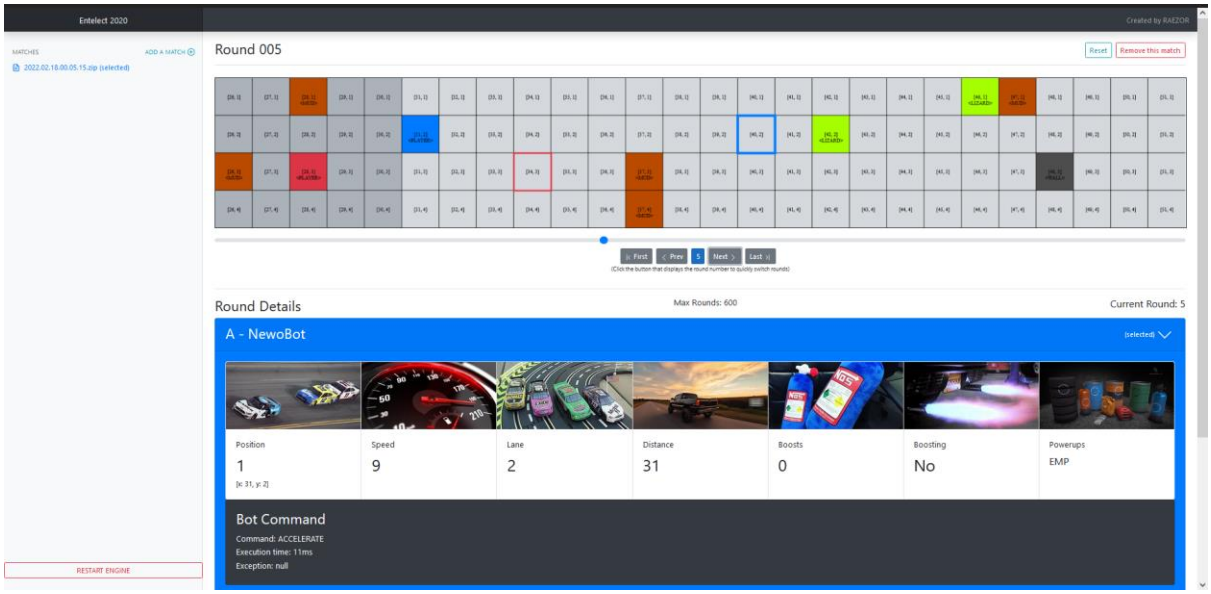
Pada ronde kedua, bot kami masih memilih untuk ACCELERATE. Karena bot kami sedang berada di *lane* paling kiri, maka bot tidak bisa TURN_LEFT. Bot memiliki beberapa pilihan yang dapat diambil, yaitu USE_EMP, ACCELERATE, TURN_RIGHT, NOTHING. Pada kondisi tersebut, musuh sedang tidak berada di depan bot sehingga USE_EMP tidak berguna. Jika bot TURN_RIGHT, bot akan mendapatkan sebuah *power up* LIZARD, tetapi bot hanya akan maju sebanyak kecepatan – 1. Maka dari itu, solusi optimal untuk ronde ini adalah melakukan ACCELERATE untuk menambah kecepatan yang dimiliki.



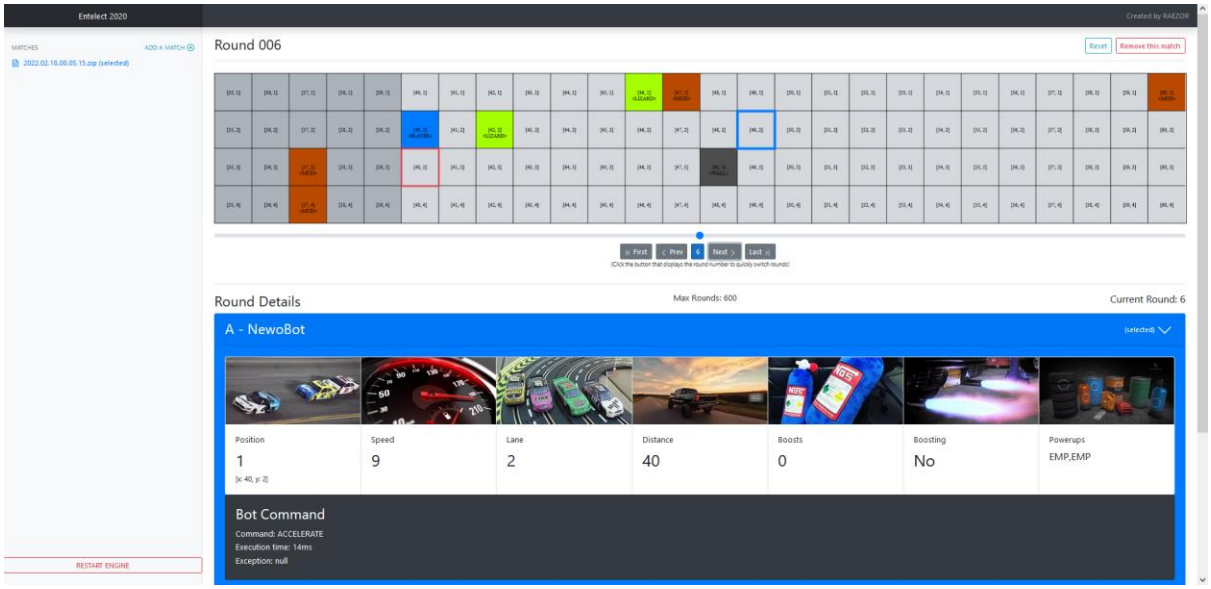
Pada ronde ketiga, keputusan yang diambil bot bukanlah solusi optimal untuk ronde ini. Hal tersebut dikarenakan kami memutuskan untuk melakukan pindai sejauh 15 *block* sehingga bot sudah membaca halangan MUD secara prematur. Padahal, jika bot memilih untuk ACCELERATE, bot akan mencapai keadaan *maximum speed* dan bot akan mendapatkan BOOST. Kemudian bot bisa menghindari MUD dengan melakukan TURN_RIGHT di ronde selanjutnya. Akan tetapi, karena 15 *block* pada *lane* kanan tidak ada rintangan dan bot kami memprioritaskan *lane* dengan jumlah rintangan terkecil, maka bot memutuskan untuk TURN_RIGHT sehingga bot kehilangan satu *block* dan kecepatan tetap di 8.



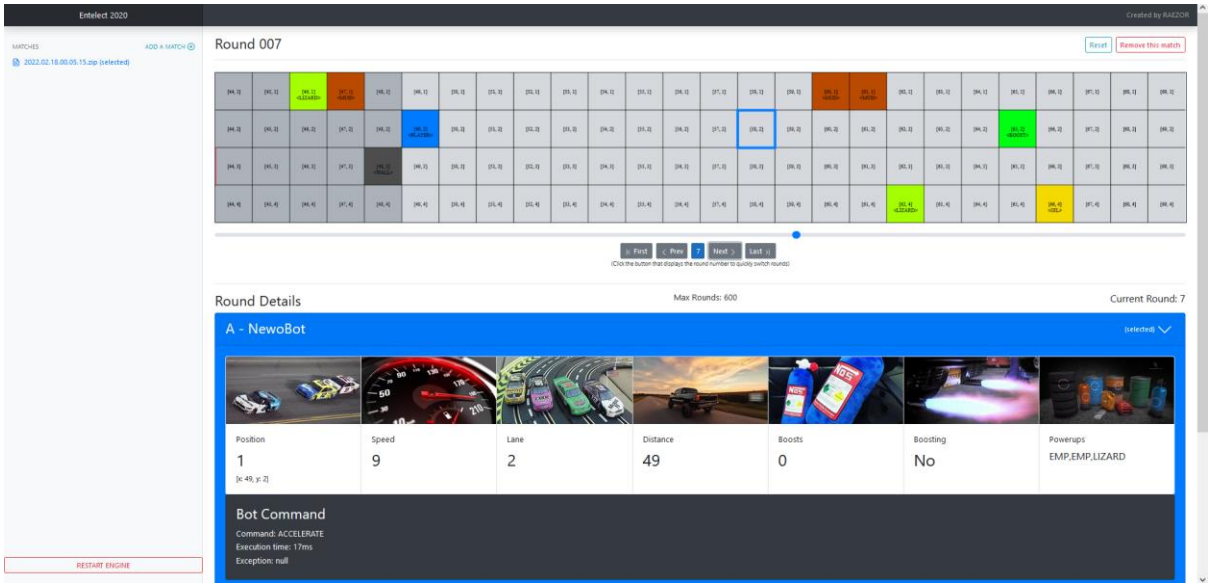
Pada ronde keempat, bot memilih untuk ACCELERATE. Jika bot melakukan TURN_RIGHT, bot akan menabrak MUD dan mendapatkan *damage*. Jika bot TURN_LEFT, bot akan mendapatkan BOOST, tetapi bot akan menabrak MUD. USE_EMP tidak akan berguna karena musuh berada di belakang bot. Oleh karena itu, pilihan terbaik yang dapat dilakukan bot adalah ACCELERATE untuk menambah kecepatan.



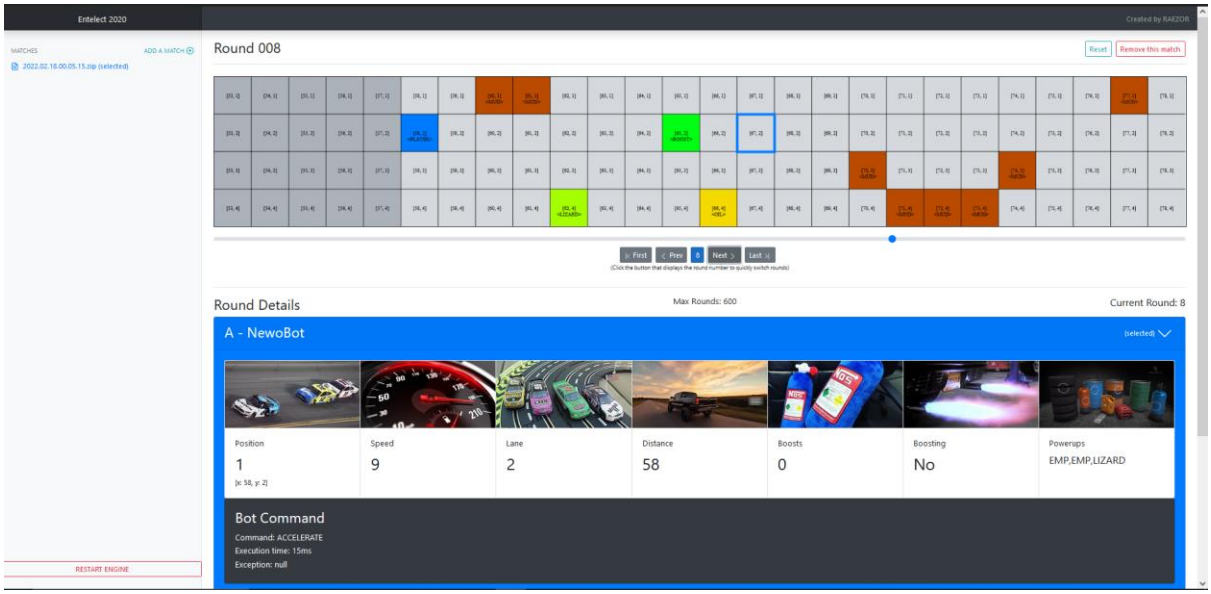
Di ronde ini, untuk 15 *block* ke depan, terdapat satu MUD di kanan, 1 *power up* di *lane* yang ditempati, dan tidak terbaca apa-apa di kiri. Karena lebih menguntungkan untuk tetap di *lane* sekarang, maka bot memiliki dua pilihan yaitu NOTHING atau ACCELERATE. Kami memutuskan untuk tetap memilih ACCELERATE karena tidak ada ruginya meskipun kecepatan tidak akan bertambah. Dapat dikatakan bahwa keputusan yang diambil adalah solusi optimal karena jika ke kiri bot akan kehilangan satu *block* dan jika ke kanan bot akan menabrak MUD.



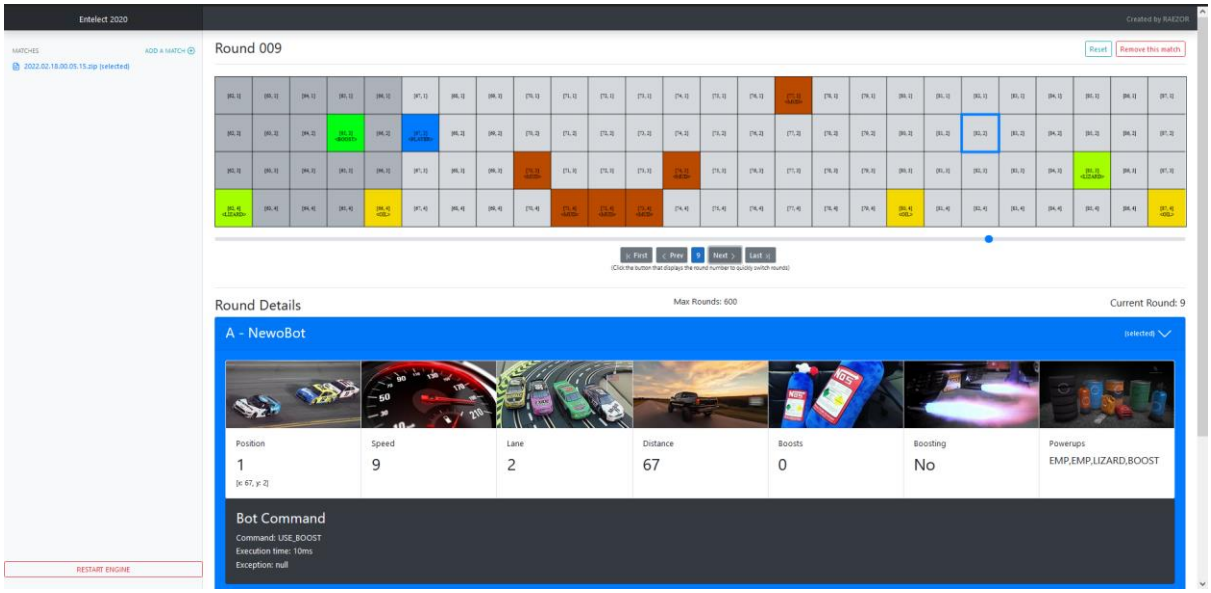
Situasi pada ronde ini hampir mirip dengan ronde sebelumnya, hanya saja kini terdapat satu *power up* di *lane* yang ditempati dan di kiri bot. Akan tetapi, terdapat halangan MUD di kiri bot. Oleh karena itu, bot lebih baik tetap di *lane* yang sedang ditempati dan melakukan ACCELERATE karena bot akan mendapatkan *power up* tanpa menabrak sebuah halangan. Menggunakan EMP tidak akan berguna karena musuh di belakang bot sehingga EMP tidak akan memberikan efek apa-apa.



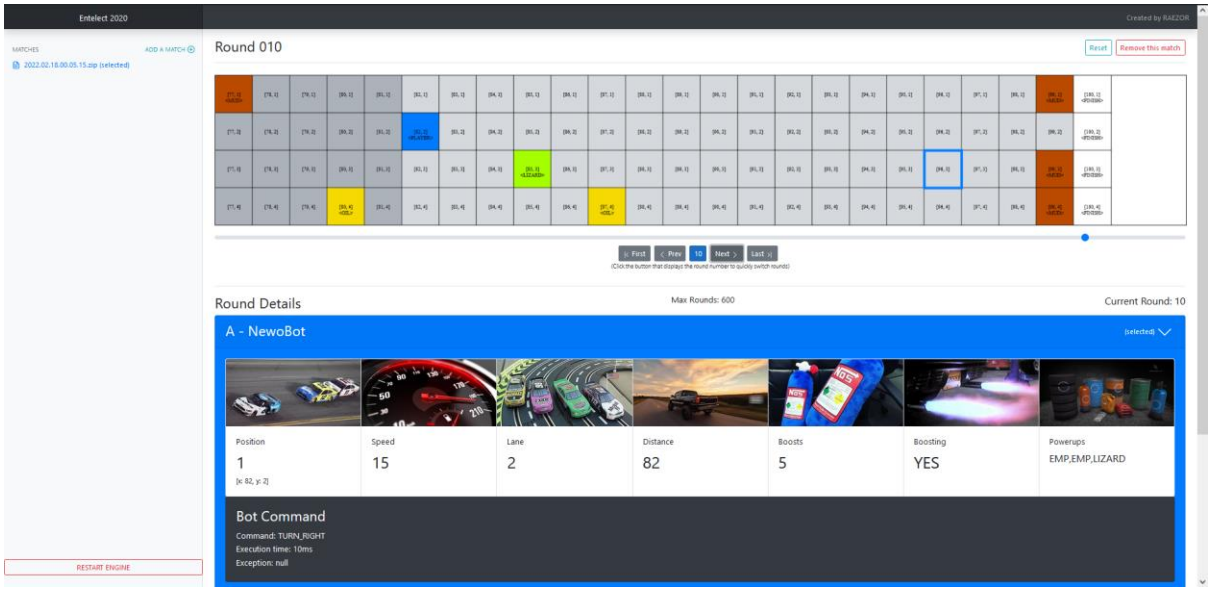
Pada ronde 7, hasil pindai robot akan membaca ada 2 halangan di kiri dan 0 halangan di kanan serta di *lane* sekarang sehingga bot mengeliminasi perintah TURN_LEFT. Karena di kanan tidak ada apa-apa, bot memutuskan untuk ACCELERATE agar tidak kehilangan satu block. Penggunaan *power up* EMP maupun LIZARD tidak berguna karena musuh di belakang dan tidak ada halangan apapun di depan sehingga LIZARD tidak diperlukan.



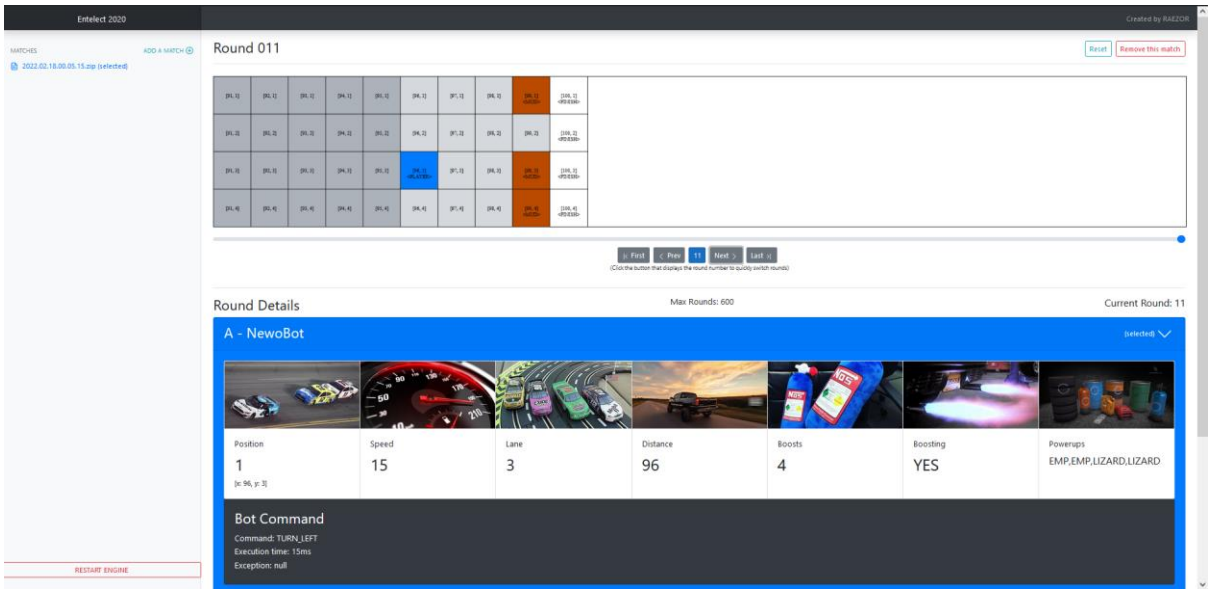
Pada ronde 8, hasil pindai akan menghasilkan 2 halangan di kiri, 1 *power up* di *lane* sekarang, dan 1 halangan di kanan. Berdasarkan informasi tersebut, bot akan memutuskan untuk ACCELERATE karena TURN_RIGHT ataupun TURN_LEFT akan merugikan bot, sedangkan di *lane* sekarang memiliki *power up* BOOST. Sama seperti ronde sebelumnya, penggunaan *power up* tidak berguna karena kondisi tidak memerlukan bot menggunakannya.



Pada ronde sebelumnya, bot mendapatkan *power up* BOOST. Karena di *lane* yang ditempati bot kosong dan terdapat halangan pada lane kanan maupun kiri, bot memutuskan untuk menggunakan BOOST. Keputusan yang diambil adalah solusi optimal untuk ronde ini karena bot akan rugi jika ke kanan maupun kiri dan bot akan mencapai garis akhir lebih cepat karena kecepatan bertambah ketika menggunaakan BOOST.



Pada ronde 10, hasil pindai menunjukkan bahwa terdapat satu *power up* di kiri. Karena tidak ada halangan pada *lane* kiri, tengah, dan kanan, maka akan lebih menguntungkan pada saat itu jika bot melakukan TURN_RIGHT sehingga bot mendapatkan *power up*. Namun, jika kita teliti hingga ke ronde berikutnya, seharusnya bot tetap di *lane* 2 sehingga bot tidak kehilangan satu *block*. Akan tetapi, karena algoritma harus menggunakan algoritma *greedy*, maka tidak diperkenankan untuk melakukan prediksi ronde selanjutnya.



Di ronde terakhir ini, hasil pindai akan menunjukkan satu halangan di *lane* tengah dan kanan. Karena bot kami memprioritaskan *lane* yang kosong, maka bot memutuskan untuk TURN_LEFT. Tentu saja hal itu bukan solusi yang optimal mengingat bot memiliki *power up* LIZARD yang mampu menghindari halangan tanpa harus belok dan kehilangan satu *block*.

Dari beberapa analisis ronde per ronde, dapat disimpulkan bahwa solusi yang didapatkan dari algoritma *greedy* saat ini bukanlah solusi yang optimal untuk memaksimalkan kecepatan. Dapat dilihat bahwa ada kasus dimana bot bisa memilih hal yang lebih optimal, tetapi tidak dilakukan.

BAB 5
KESIMPULAN DAN SARAN

A. Kesimpulan

Setelah melakukan berbagai pengamatan dan eksperimen sebelum menentukan alternatif algoritma terbaik, penulis berpendapat bahwa salah satu cara terbaik untuk mendapatkan solusi optimum lokal (metode greedy) adalah dengan membagi sebuah ronde ke dalam berbagai kemungkinan skenario (berdasarkan posisi mobil dan lawan, lane dan block, serta ada tidaknya rintangan di depan) serta mendesain algoritma sesuai dengan prioritasnya untuk masing-masing skenario. Perancangan algoritma oleh penulis juga memprioritaskan kecepatan, menghindari rintangan, penggunaan powerup, dan sabotase lawan sesuai dengan skenario-skenario yang tersedia.

B. Saran

Dikarenakan tidak adanya akses ke sebuah bot mobil yang bukan referensi atau dengan kata lain, yang belum terimplementasi sepenuhnya, maka sulit untuk melakukan benchmarking yang lebih berbobot. Maka dari itu, perlu diadakan kolaborasi dengan penulis lain pada tugas besar yang sama agar dapat membantu proses debugging.

DAFTAR PUSTAKA

Munir, R. (2022). *Algoritma Greedy (Bagian 1)*. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf

Munir, R. (2022). *Algoritma Greedy (Bagian 2)*. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf

Munir, R. (2022). *Algoritma Greedy (Bagian 3)*. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag3.pdf

LINK PENTING

Video Kelompok YouTube : https://youtu.be/BuAaA_AD4K4

Github Repository : https://github.com/clumsyyyy/TubesStima1