Laporan Tugas Besar 1
IF2211 Strategi Algoritma

Disusun oleh:

AmbatuTank

Sebastian Hung Yansen	13523070
Aramazaya	13523082
Zulfaqqar Nayaka Athadiansyah	13523094

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Bandung

2025

Daftar Isi

Daftar Isi **1**
1. Deskripsi Tugas **2**
2. Landasan Teori **7**
2.1. Algoritma Greedy 7
2.2. Cara Kerja Program 7
3. Penerapan Algoritma **9**
3.1. Identifikasi 9
3.2. Eksplorasi 10
3.2.1. Alternatif Solusi 1: Weakest Energy 10
3.2.2. Alternatif Solusi 2: AramBot 10
3.2.3. Alternatif Solusi 3: Lazy Survival 10
3.2.4. Alternatif Solusi 4: AmbatuTank 11
3.3. Analisis 11
3.3.1. Alternatif Solusi 1 11
3.3.2. Alternatif Solusi 2 12
3.3.3. Alternatif Solusi 3 12
3.3.4. Alternatif Solusi 4 12
3.4. Hasil 12
4. Implementasi dan Pengujian **13**
4.1. Implementasi 13
4.1.1. Alternatif Solusi 1: Weakest Energy 13
4.1.2. Alternatif 2: AramBot 15
4.1.3. Alternatif 3: NayakaBot 15
4.1.4. Alternatif 4: AmbatuTank 16
4.2. Pengujian 17
4.3. Hasil 17
5. Penutup **18**
5.1. Kesimpulan 18
5.2. Saran 18
Lampiran **19**
Daftar Pustaka **20**

1. Deskripsi Tugas

Robocode adalah permainan pemrograman yang bertujuan untuk membuat kode bot dalam bentuk tank virtual untuk berkompetisi melawan bot lain di arena. Pertempuran Robocode berlangsung hingga bot-bot bertarung hanya tersisa satu seperti permainan Battle Royale, karena itulah permainan ini dinamakan Tank Royale. Nama Robocode adalah singkatan dari "Robot code," yang berasal dari versi asli/pertama permainan tersebut. Robocode Tank Royale adalah evolusi/versi berikutnya dari permainan ini, di mana bot dapat berpartisipasi melalui Internet/jaringan. Dalam permainan ini, pemain berperan sebagai programmer bot dan tidak memiliki kendali langsung atas permainan. Pemain hanya bertugas untuk membuat program yang menentukan logika atau "otak" bot. Program yang dibuat akan berisi instruksi tentang cara bot bergerak, mendeteksi bot lawan, menembakkan senjatanya, serta bagaimana bot bereaksi terhadap berbagai kejadian selama pertempuran.

Pada Tugas Besar pertama Strategi Algoritma ini, mahasiswa diminta untuk membuat sebuah bot yang nantinya akan dipertandingkan satu sama lain. Tentunya mahasiswa harus menggunakan strategi greedy dalam membuat bot ini. Komponen-komponen dari permainan ini antara lain:

1. **Rounds dan Turns**

   Pertempuran dapat terdiri dari beberapa *rounds* (giliran). Secara *default*, satu pertempuran berisi 10 *rounds*, di mana setiap *rounds* akan memiliki pemenang dan yang kalah.

   Setiap round dibagi menjadi beberapa turns, yang merupakan unit waktu terkecil. Satu turn adalah satu ketukan waktu dan satu putaran permainan. Jumlah turn dalam satu *round* tergantung pada berapa lama waktu yang dibutuhkan hingga hanya tersisa bot terakhir yang bertahan.

   Pada setiap turn, sebuah bot dapat:

   - Menggerakkan bot, memindai musuh, dan menembakkan senjata.
   - Bereaksi terhadap peristiwa seperti saat bot terkena peluru atau bertabrakan dengan bot lain atau dinding.
   - Perintah untuk bergerak, berputar, memindai, menembak, dan sebagainya dikirim ke server untuk setiap turn.

   Perlu diperhatikan bahwa API (Application Programming Interface) bot resmi secara otomatis mengirimkan niat bot ke server di balik layar, sehingga Anda tidak perlu mengkhawatirkannya, kecuali jika Anda membuat API Bot sendiri. Pada setiap *turn*, bot akan secara otomatis menerima informasi terbaru tentang posisinya dan orientasinya di medan perang. Bot juga akan mendapatkan informasi tentang bot musuh ketika mereka terdeteksi oleh pemindai.

2. **Batas Waktu Giliran**

Penting untuk dicatat bahwa setiap bot memiliki batas waktu untuk setiap *turn* yang disebut *turn timeout*, biasanya antara 30-50 ms (dapat diatur sebagai aturan pertempuran). Ini berarti bahwa bot tidak bisa mengambil waktu sebanyak yang mereka inginkan untuk bergerak dan menyelesaikan turn saat ini.

Setiap kali *turn* baru dimulai, penghitung waktu ulang diatur ulang dan mulai berjalan. Jika batas waktu tercapai dan bot tidak mengirimkan pergerakannya untuk *turn* tersebut, maka tidak ada perintah yang dikirim ke server. Akibatnya, bot akan melewatkan *turn* tersebut. Jika bot melewatkan *turn*, ia tidak akan bisa menyesuaikan gerakannya atau menembakkan senjatanya karena server tidak menerima perintah tepat waktu sebelum *turn* berikutnya dimulai.

3. **Energi**

Semua bot memulai permainan dengan jumlah energi awal sebanyak 100 poin energi.

- Bot akan kehilangan energi jika ditembak atau ditabrak oleh bot musuh.
- Bot juga akan kehilangan energi jika menembakkan meriamnya.
- Bot akan mendapatkan energi jika peluru dari meriamnya mengenai musuh. Energi yang didapat akan lebih banyak 3 kali lipat dari energi yang digunakan untuk menembakkan peluru.
- Bot dengan energi nol akan dinonaktifkan dan tidak bisa bergerak. Jika bot terkena serangan dalam keadaan ini, bot akan hancur.

4. **Peluru**

Semakin banyak energi (daya tembak) yang digunakan untuk menembakkan peluru, semakin berat peluru tersebut dan semakin lambat gerakannya. Namun, peluru yang lebih berat juga menghasilkan lebih banyak kerusakan dan memungkinkan bot mendapatkan lebih banyak energi saat mengenai bot musuh.

Seperti disebutkan sebelumnya, peluru yang lebih berat akan bergerak lebih lambat. Ini berarti akan membutuhkan waktu lebih lama untuk mencapai target, meningkatkan risiko peluru tidak mengenai sasaran. Sebaliknya, peluru yang lebih ringan bergerak lebih cepat, sehingga lebih mudah mengenai target, tetapi peluru ringan tidak memberikan banyak poin energi saat mengenai bot musuh.

5. **Panas Meriam (*Gun Heat*)**

Saat menembakkan peluru, meriam akan menjadi panas. Peluru yang lebih berat menghasilkan lebih banyak panas dibandingkan peluru yang lebih ringan. Ketika meriam terlalu panas, bot tidak dapat menembak hingga suhu meriam turun ke nol. Selain itu, meriam juga sudah dalam keadaan panas di awal round dan perlu waktu untuk mendingin sebelum bisa digunakan untuk pertama kalinya.

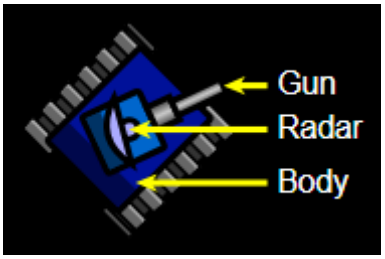
6. **Tabrakan**

Perlu diperhatikan bahwa bot akan menerima kerusakan jika menabrak dinding (batas arena), yang disebut *wall damage*. Hal yang sama juga terjadi jika bot bertabrakan dengan bot

lain. Jika bot menabrak bot musuh dengan bergerak maju, ini disebut *ramming* (menabrak dengan sengaja), yang akan memberikan sedikit skor tambahan bagi bot yang menyerang.

7. **Bagian Tubuh Tank**

   Tubuh tank terdiri dari 3 bagian:



   - *Body* adalah bagian utama dari tank yang digunakan untuk menggerakkan tank.
   - *Gun* digunakan untuk menembakkan peluru dan dapat berputar bersama *body* atau independen dari *body*.
   - Radar digunakan untuk memindai posisi musuh dan dapat berputar bersama *body* atau independen dari *body*.

8. **Pergerakan**

   Bot dapat bergerak maju dan mundur hingga kecepatan maksimum. Dibutuhkan beberapa giliran untuk mencapai kecepatan maksimum. Bot dapat mengalami percepatan maksimum sebesar 1 unit per giliran dan pengereman dengan perlambatan maksimum 2 unit per giliran. Percepatan dan perlambatan maksimum tidak bergantung pada kecepatan bot saat itu.

9. **Berbelok**

   Seperti yang disebutkan sebelumnya, bagian tubuh, turret (meriam), dan radar dapat berputar secara independen satu sama lain. Jika turret atau radar tidak diputar, maka keduanya akan mengarah ke arah yang sama dengan tubuh bot.

   Setiap bagian tubuh memiliki kecepatan putar yang berbeda. Radar adalah bagian tercepat dan dapat berputar hingga 45 derajat per giliran, yang berarti dapat berputar 360 derajat dalam 8 giliran. Turret dan meriam dapat berputar hingga 20 derajat per giliran.

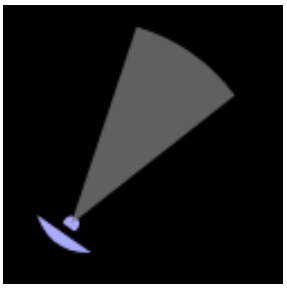
   Bagian paling lambat adalah tubuh tank, yang dalam kondisi terbaik dapat berputar hingga 10 derajat per giliran. Namun, ini bergantung pada kecepatan bot saat ini. Semakin cepat bot bergerak, semakin lambat kemampuannya untuk berbelok.

   Perlu diperhatikan bahwa tidak ada energi yang dikonsumsi saat bot bergerak atau berbelok.

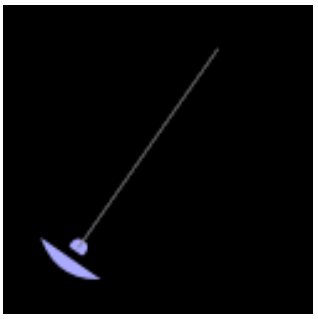
10. **Pemindaian**

    Aspek penting dalam Robocode adalah memindai bot musuh menggunakan radar. Radar dapat mendeteksi bot dalam jangkauan hingga 1200 piksel. Musuh yang berada lebih dari 1200 piksel dari bot tidak dapat terdeteksi atau dipindai oleh radar.

Penting untuk diperhatikan bahwa sebuah bot hanya dapat memindai bot musuh yang berada dalam jangkauan sudut pemindaian (scan arc)-nya. Sudut pemindaian ini merupakan "sapuan radar" dari arah radar sebelumnya ke arah radar saat ini dalam satu giliran.



Jika radar tidak bergerak dalam suatu giliran, artinya radar tetap mengarah ke arah yang sama seperti pada giliran sebelumnya, maka sudut pemindaian akan menjadi nol derajat, dan bot tidak akan dapat mendeteksi musuh.



Oleh karena itu, sangat disarankan untuk selalu mengubah arah radar agar tetap dapat memindai musuh.

11. Skor

Pada akhir pertempuran, setiap bot akan diranking berdasarkan total skor yang diperoleh masing-masing bot selama keseluruhan pertempuran. Tentunya, tujuan utama pada tugas besar ini adalah membuat bot yang memberikan skor setinggi mungkin. Berikut adalah rincian komponen skor pada pertempuran:

- *Bullet Damage*: Bot mendapatkan poin sebesar *damage* yang dibuat kepada bot musuh menggunakan peluru.
- *Bullet Damage Bonus*: Apabila peluru berhasil membunuh bot musuh, bot mendapatkan poin sebesar 20% dari *damage* yang dibuat kepada musuh yang terbunuh.
- *Survival Score*: Setiap ada bot yang mati, bot lainya yang masih bertahan pada ronde tersebut mendapatkan 50 poin.
- *Last Survival Bonus*: Bot terakhir yang bertahan pada suatu ronde akan mendapatkan 10 poin dikali dengan banyaknya musuh.
- *Ram Damage*: Bot mendapatkan poin sebesar 2 kalinya *damage* yang dibuat kepada bot musuh dengan cara menabrak.
- *Ram Damage Bonus*: Apabila musuh terbunuh dengan cara ditabrak, bot mendapatkan poin sebesar 30% dari *damage* yang dibuat kepada musuh yang terbunuh.

Skor akhir bot adalah akumulasi dari 6 komponen di atas. Perlu diperhatikan bahwa game akan menampilkan berapa kali suatu bot meraih peringkat 1, 2, atau 3 pada setiap ronde. Namun, hal ini tidak dihitung sebagai komponen skor maupun untuk perangkingan akhir. Bot yang dianggap menang pertempuran adalah bot dengan akumulasi skor tertinggi.

2. Landasan Teori

2.1. Algoritma Greedy

Algoritma *greedy* adalah salah satu ide perancangan algoritma yang berfokus pada pencarian solusi dari suatu masalah melalui maksimasi (*Maximization*) atau minimasi (*Minimization*). Cikal bakal pembakuan algoritma *greedy* adalah algoritma Dijkstra yang dicetuskan oleh informatikawan Belanda Edsger Dijkstra pada 1959 untuk menyelesaikan persoalan mencari jarak terpendek antara dua titik. Beberapa penerapan algoritma *greedy* mencakup *scheduling* pada sistem operasi (dipelajari pada IF2130 Sistem Operasi), pencarian pohon merentang minimum (*minimum spanning tree*) seperti algoritma Prim dan Kruskal, dan masih banyak lagi.

Ketika diharuskan untuk mengambil keputusan dalam suatu persoalan, algoritma *greedy* akan memilih solusi yang sekilas optimal pada saat itu (solusi optimal lokal) tanpa mempedulikan solusi optimal pada kondisi sebelumnya maupun sesudahnya, yang boleh jadi lebih optimal. Akibatnya, solusi yang didapatkan lewat algoritma *greedy* boleh jadi bukanlah solusi optimal global, yakni solusi paling optimal dari suatu instansiasi persoalan atau sub-optimal.

Kerangka berpikir utama dari algoritma *greedy* dimulai dengan menginisiasi solusi (solusi kosong). Selanjutnya, solusi ini diperbarui dengan memperbarui solusi tersebut dengan solusi optimal lokal yang didapatkan. Hal ini terus diulangi hingga kondisi akhir dari persoalan tercapai.

Algoritma *greedy* umumnya tidak melibatkan *backtracking* sehingga menghemat waktu komputasi dan menghindari redundansi akibat memperhitungkan langkah yang invalid dua kali atau lebih. Artinya, ketika algoritma ini memilih keputusan yang salah, tidak ada cara untuk memperbaikinya. Padahal, tidak semua persoalan bisa diselesaikan dengan cara pikir yang *greedy*.

Di samping itu, algoritma *greedy* terhitung sederhana dan efisien karena jarang melibatkan struktur data yang kompleks. Hal ini membuat solusi yang dihasilkannya cenderung mempunyai kompleksitas waktu yang rendah. Namun, apabila algoritma *greedy* menghasilkan solusi optimal, keoptimalannya harus dapat dibuktikan secara matematis, yang menjadikannya tantangan tersendiri.

2.2. Cara Kerja Program

Sebuah pertandingan Robocode terdiri dari beberapa ronde. Semua *bot* dalam Robocode dapat bergerak, memindai musuh, dan menembakkan senjata. Pada setiap *turn* (giliran), sebuah *bot* dapat melakukan aksi-aksi tersebut atau bereaksi terhadap sebuah peristiwa dalam ronde. Perintah-perintah tersebut dikirim ke server melalui API untuk setiap giliran

secara otomatis di balik layar. Setiap giliran juga *bot* akan menerima informasi terkait posisi, orientasi, serta bila *bot* terdeteksi oleh pemindai musuh.

Implementasi algoritma *greedy* ke dalam *bot* bisa menggunakan pendekatan *heuristic* yang berbeda-beda. Hal tersebut dikarenakan *bot* bisa mengetahui informasi *bot* lain seperti sisa energi, jarak, posisi, dan lain-lain sesuai yang disediakan API. Kemudian *bot* bisa di-*program* berdasarkan informasi yang didapat untuk melakukan suatu hal.

Cara untuk menjalankan *bot* cukup mudah. Pertama-tama, *engine robocode* akan melakukan pemindaian terhadap folder yang di-*set* oleh pengguna. Kemudian, *engine robocode* akan *list* seluruh direktori *bot* yang ada. *Engine* robocode sendiri sudah bisa *compile bot* dengan sendirinya dan pengguna hanya tinggal memilih *bot* mana saja yang ingin dipilih lalu menekan tombol *boot*. Setelah beberapa waktu, akan muncul berbagai *bot* yang sudah di-compile dan berhasil *join* di kotak bawah kiri. Apabila *bot* tidak muncul, artinya *bot* tersebut gagal di-compile akibat error dalam kode atau alasan lainnya. Setelah *bot* sudah *join*, pengguna dapat memilih *bot* mana saja yang akan bertarung. Pengguna kemudian tinggal menekan tombol '*start battle*' untuk memulai pertandingan. Pengaturan jenis pertandingan, jumlah ronde, jumlah *bot* maksimal yang bertanding, *server*, dan berbagai macam peraturan lainnya dapat diatur sebelum pertandingan dimulai bahkan sebelum *server* robocode berjalan.

3. Penerapan Algoritma

3.1. Identifikasi

Pada tiap langkah, ada sejumlah kemungkinan tindakan yang dapat kita ambil, yakni:

- bergerak;
- menembak dengan energi tertentu;
- memindai musuh; dan
- bereaksi terhadap gangguan eksternal (tembakan peluru, tabrakan tembok atau *bot* lain).

Langkah-langkah ini menyusun suatu **himpunan kandidat** dari solusi untuk instansiasi persoalan kali ini, yakni pertandingan Robocode. Sebagian dari tindakan tadi dapat kita pilih untuk dieksekusikan pada suatu giliran. Dengan begini, kita pada dasarnya memilih elemen dari himpunan kandidat untuk dimasukkan ke dalam **himpunan solusi**, yang secara keseluruhan merepresentasikan solusi dari persoalan ini. Elemen ini dipilih oleh **fungsi seleksi**, yang pada kasus ini mempertimbangkan setidaknya dua hal utama:

- melakukan tindakan dengan konsumsi energi sesedikit mungkin untuk sebanyak mungkin mendapatkan bonus *survival* atau bahkan *last survival*, dan
- mendapatkan poin sebanyak-banyaknya dengan menyerang musuh secara efisien.

Akibatnya, barangkali kita perlu untuk menetapkan sejumlah batasan untuk menyaring elemen himpunan kandidat yang kita pilih untuk menjadi bagian dari himpunan solusi. Misalnya, kami memastikan bahwa tindakan yang diambil tidak mengkonsumsi energi yang terlalu banyak (disepakati 5 poin), tidak membuat meriamnya terlalu panas untuk menembak, dan menghindari menabrak tembok. Jika menabrak musuh diperlukan, kami menetapkan batas maksimum energi *bot* musuh supaya layak untuk ditabrak, yakni 15 energi, sebab menabrak *bot* musuh akan mengurangi energi kita, meskipun akan menghasilkan bonus yang mantap jika kita berhasil membunuh musuh yang bersangkutan. Batasan-batasan ini menjadi dasar dari **fungsi kelayakan** yang berfungsi menentukan apakah suatu kandidat *layak* untuk ditambahkan ke dalam himpunan solusi.

Menimbang tujuan akhir dari permainan Robocode adalah mendapatkan skor sebanyak mungkin, memantau skor yang telah diperoleh dan memaksimalkan skor dengan memilih tindakan dengan keuntungan terbesar pada tiap giliran adalah suatu keharusan. Hal ini diimplementasikan sebagai suatu **fungsi objektif** yang berperan memaksimalkan/meminimalkan "nilai" dari suatu solusi yang diambil pada suatu giliran.

3.2. Eksplorasi

3.2.1. Alternatif Solusi 1: Weakest Energy

Alternatif solusi ini dirancang oleh Sebastian Hung Yansen. *Bot* ini dirancang untuk menarget dan menyerang musuh yang memiliki energi paling rendah. Setelah penembakan, *bot* akan mendekati musuh tersebut kemudian *scan* area untuk mencari musuh berikutnya yang memiliki energi paling rendah dan *bot* akan lanjut melakukan hal tersebut hingga permainan berakhir. Apabila *bot* yang terdeteksi memiliki energi paling rendah dan sedang menyentuh *bot*, *bot* tersebut akan melawan dengan cara menabraknya.

Secara pergerakan, *bot* hanya bergerak secara *zig zag* sederhana terhadap musuh yang ia tembak. Apabila *bot* terkena peluru musuh, ia hanya akan mundur untuk menghindar secara sederhana.

3.2.2. Alternatif Solusi 2: Closest Enemy

Alternatif solusi ini dirancang oleh Aramazaya. *Bot* ini dirancang dengan tujuan *pure greedy*. Strategi *bot* ini adalah dengan melakukan *scanning* tanpa bergerak. Ketika telah melakukan *scan* 360 derajat, *bot* akan menembak musuh terdekat dari lokasi *Bot*. Jika tertembak, *Bot* akan melakukan *ramming* (menabrak) ke arah *bot* yang menembakan peluru. Jika *ramming* berhasil, *bot* akan memutar tembakan dan menembak ke arah *bot* lawan. Secara pergerakan, *bot* tidak melakukan gerakan apapun untuk memaksimalkan strategi *ramming* ketika terkena *hit.* Terdapat juga strategi ketika *bot* menabrak *wall,* yaitu *bot* akan mundur 50 langkah.

3.2.3. Alternatif Solusi 3: Lazy Survival

Alternatif solusi ini dirancang oleh Z. Nayaka Athadiansyah. Berdasarkan identifikasi (subbab 3.1), *bot* ini dirancang untuk bertahan hidup selama mungkin dan mendapatkan skor sebanyak mungkin dengan *effort* sesedikit mungkin. Jadi, tidak hanya *greedy*, alternatif solusi ini juga *lazy*.

Cara mengendalikan *bot* dapat dikotakkan menjadi dua aspek, yakni dari segi *movement* (gerakan) dan *melee* (serangan).

Dari segi *movement*, ada beberapa faktor yang dipertimbangkan. Yang pertama, jelas *bot* harus menghindari menabrak dinding. Kedua, *bot* harus menghindari *bot* lain, entah itu untuk mencegah ditabrak (*ramming*) maupun terkena tembakan. Berangkat dari dua faktor tersebut, alternatif solusi kali ini mengadopsi ide dari algoritma perencanaan jalur berbasis medan potensial buatan (*artificial potential field path planning*) yang lebih familiar dijumpai di robotika.

Dengan algoritma *artificial potential field path planning*, tiap *bot* musuh dan dinding dipandang sebagai objek yang memberikan gaya tolak terhadap *bot* kita. Jika tiap vektor gaya ini dijumlahkan, kita akan mendapatkan titik teraman untuk *bot* pada suatu giliran. Panjang dari vektor gaya yang ditimbulkan oleh tiap robot x ditentukan dengan persamaan

$$f(x) = \frac{w_1}{r(x)} + w_2 h(x)$$

dengan $r(x)$ adalah jarak robot ke titik terluar *bounding circle* terdekat ke arah robot lawan x yang dituju, $h(x)$ adalah rasio dari energi robot x terhadap robot kita, serta w_1 dan w_2 adalah konstanta yang didapatkan dengan *tuning* lewat pengujian. Dari sudut pandang *greedy*, strategi ini memungkinkan kita untuk memilih jalur dengan risiko paling minimum. Akibatnya, *survivability bot* dapat makin tinggi sehingga memungkinkannya mendapatkan bonus poin *survival* sebanyak mungkin.

Dari segi *melee*, kita berusaha untuk menyerang musuh yang dekat-dekat saja supaya kemungkinan peluru meleset dapat diperkecil. Kita menetapkan 60 sebagai ambang batas jarak maksimum supaya suatu *bot* dapat dipertimbangkan untuk diserang. Daya dari peluru yang ditembakkan di-*scaling* secara dinamis: makin dekat, makin kuat. Di sini kita menerapkan *linear targetting* untuk membidik musuh, dengan mengasumsikan posisi musuh selanjutnya berdasarkan arah gerak dan kecepatan musuh yang didapatkan dengan pemindaian radar.

3.2.4. Alternatif Solusi 4: Strongest Energy

Alternatif solusi ini kami rancang bersama-sama. Secara garis besar, *bot* ini menarget musuh-musuh dengan energi paling besar. Pergerakan yang ia lakukan adalah *zig-zag* seperti pada alternatif solusi 1 agar dapat menghindar dengan lebih baik dan penembakannya mengikuti algoritma yang diterapkan di alternatif solusi 3.

3.3. Analisis

3.3.1. Alternatif Solusi 1

Pemilihan target berdasarkan energi *bot* yang paling lemah terdengar efektif di atas kertas namun ada situasi-situasi di mana algoritma tersebut kurang efektif di implementasi. Hal tersebut dikarenakan *bot* bisa saja mendeteksi bahwa *bot* dengan energi paling rendah memiliki jarak paling jauh sehingga ia akan menghabiskan banyak waktu dan energi untuk mendekati serta menembak ke arahnya.

Kompleksitas waktu dari alternatif solusi ini adalah O(n) dikarenakan algoritma *bot* menggunakan looping sederhana *while loop* .

3.3.2. Alternatif Solusi 2

Ide ini merupakan implementasi paling dasar dari strategi *greedy* dimana *bot* akan menargetkan musuh terdekat. Implementasi yang sedikit unik dari strategi ini adalah penambahan *ramming* ke arah peluru. Dengan menggunakan *ramming* tanpa algoritma pengarah yang lebih detail, strategi ini menggambarkan strategi *greedy* dimana *bot* hanya melihat kondisi ketika Ia terkena tembakan dan tidak memikirkan kondisi setelah melakukan *ramming*. Setelah melakukan *testing*, didapat bahwa strategi ini sangat bagus untuk pertarungan jarak dekat tetapi tidak terlalu bagus untuk jarak jauh dikarenakan tingkat akurasi dari *ramming*.

Kompleksitas waktu dari solusi ini adalah O(n) dikarenakan tidak adanya nested loop dan hanya digunakannya satu perulangan biasa.

3.3.3. Alternatif Solusi 3

On paper, ide yang dipakai sebenarnya bagus karena banyak dipakai di bidang robotika, khususnya untuk UAV (*unmanned aerial vehicles*). Akan tetapi, masih perlu banyak penyesuaian supaya algoritmanya dapat diimplementasikan sebagaimana mestinya. *Aim* dari *bot* ternyata masih perlu diperbaiki lagi. *Bot* menembak dengan baik ketika musuh berada pada jarak yang dekat. Akan tetapi, pada musuh yang berjarak jauh atau bergerak secara osilasi, *bot* ini kurang mampu menembak dengan tepat.

Secara keseluruhan, alternatif solusi ini hanya melibatkan perulangan sederhana; tidak ada *nested loops*. Selain itu, penggunaan *dictionary* untuk menyimpan gaya tolak yang diberikan oleh seluruh *bot* hanya memakan kompleksitas waktu O(1). Dengan demikian, kompleksitas waktu dari alternatif solusi ini adalah O(n).

3.3.4. Alternatif Solusi 4

Penggabungan dua fungsi dari dua alternatif solusi yang berbeda terdengar seperti sesuatu yang baik. Tetapi nyatanya, algoritma kurang sesuai dan kurang cocok dengan satu sama lain. Pergerakan yang mendekati musuh hanya membahayakannya.

Seperti alternatif solusi lainnya, alternatif solusi ini juga hanya menggunakan perulangan sederhana dan menggunakan *dictionary* sehingga kompleksitas waktunya adalah O(n)

3.4. Hasil

Berdasarkan alternatif solusi-alternatif solusi yang telah dipaparkan, kelompok kami sepakat untuk memilih alternatif solusi 3 yakni *Lazy Survival*. Alasan serta pertimbangan kami memilih alternatif solusi tersebut dikarenakan fokusnya pada *Survivability* membuatnya tahan lebih lama dan memiliki kesempatan mendapatkan poin yang lebih banyak dibandingkan

dengan *bot* lain yang mendekati musuh untuk mendapatkan poin dengan bahaya *bot* tersebut mudah ditembak musuh.

4. Implementasi dan Pengujian

4.1. Implementasi

4.1.1. Alternatif Solusi 1: Weakest Energy

Pseudocode alternatif solusi 1 beserta penjelasannya sebagai berikut:

```
procedure OnScannedBot(evt : ScannedBotEvent)

{Menyimpan data musuh ketika bot scan musuh}

Deklarasi
       {Data dari musuh yang di-scan}
       evt.Energy, evt.X, evt.Y : double
       evt.ScannedBotId : integer
       {Data yang disimpan bot}
       weakestEnergy, weakestEnemyX, weakestEnemyY : double
       weakestEnemy : integer

Algoritma
       if (evt.Energy < weakestEnergy) then
              weakestEnergy = evt.Energy
              weakestEnemy = evt.ScannedBotId
              weakestEnemyX = evt.X
              weakestEnemyY = evt.Y
       {Data ini akan disimpan dan digunakan fungsi lain}
```

```
procedure OnHitByBullet(evt : HitByBulletEvent)

{Menghindar ketika bot tertembak oleh musuh}

Deklarasi
       {Data dari musuh yang menembak}
       evt.Bullet.Direction : double
       {Data yang disimpan bot}
       bulletBearing : double

Algoritma
       bulletBearing = CalcBearing(evt.Bullet.Direction)
       TurnRight(NormalizeRelativeAngle(90 - bulletBearing))
       Back(50)
```

```
procedure OnBotDeath(evt : BotDeathEvent)

{Reset data musuh dengan energi paling kecil yang tersimpan}

Deklarasi
       {Data dari musuh yang menembak}
       evt.Bullet.Direction : double
       evt.VictimId : integer
       {Data yang disimpan bot}
       weakestEnergy : double
       weakestEnemy : integer

Algoritma
       if evt.VictimId = weakestEnemy then
              weakestEnemy <- -1
              weakestEnergy < double.MaxValue
```

```
procedure OnHitBot(HitBotEvent evt)

{Menabrak musuh yang menyentuh bot daripada menembak}

Deklarasi
       evt.Energy, evt.X, evt.Y : double
       weakestEnergy, DIrection : double

Algoritma
       if (evt.Energy < weakestEnergy) then
              SetTurnRight(NormalizeRelativeAngle(DirectionTo(evt.X,
              evt.Y) - Direction))
              Forward(100)
```

```
procedure FireAtWill()

{Menembak musuh}

Deklarasi
       weakestEnergy, weakestEnemyX, weakestEnemyY : double
       weakestEnemy : integer
       bearing, gunBearing, GunDirection : double

Algoritma
       if weakestEnemy ≠ -1 then
              bearing = DirectionTo(weakestEnemyX,weakestEnemyY)
              gunBearing    =    NormalizeRelativeAngle(bearing    -
              GunDirection)
              TurnGunLeft(gunBearing)
              {Damage berbeda apabila jarak jauh dan dekat}
              if (DistanceTo(weakestEnemyX, weakestEnemyY) < 50 then
                     Fire(3)
              else
                     Fire(1)
              MoveToEnemy(weakestEnemyX, weakestEnemyY)
              {Reset target}
              weakestEnemy = -1
              weakestEnergy = double.MaxValue
```

```
procedure MoveToEnemy(targetX, target : double)

{Menembak musuh}

Deklarasi
       angleToEnemy, offset, Direction : double

Algoritma
       angleToEnemy = DirectionTo(targetX, targetY)
       TurnLeft(NormalizeRelativeAngle(angleToEnemy-Direction))
       offset = 45
       SetForward(150);
       SetTurnRight(offset);
       WaitFor(new TurnCompleteCondition(this));
       SetTurnLeft(2 * offset);
       WaitFor(new TurnCompleteCondition(this));
       SetTurnRight(offset);

{TurnCompleteCondition merupakan fungsi tambahan memeriksa sisa
giliran bot}
```

4.1.2. Alternatif 2: AramBot

```
procedure OnScannedBot(ScannedBotEvent evt)

{Menyimpan data musuh dengan jarak terkecil atau mencari musuh
dengan ID tertentu dan melakukan ramming}

Deklarasi
       p1, p2, closestPoint, perpDir : Point
       hit : boolean
       eventDistance, closest, closestDir, Direction, evt.X, evt.Y :
       double
       perpID, evt.ScannedBotId : integer

Algoritma
       if Not hit, then
          eventDistance ← FindDistance(p1, p2)
          if eventDistance < closest, then
              closest ← eventDistance
              closestPoint ← p1
       else, then
          if evt.ScannedBotId == perpID, then
              perpDir ← p1
              TurnLeft(NormalizedRelativeAngle(DirectionTo(perpDir.X,
              perpDir.Y) - Direction)
              Forward(500)
              hit ← false
              perpID ← 0
```

```
procedure OnHitByBullet(HitByBulletEvent evt)

{Menyimpan ID musuh yang menembak peluru}

Deklarasi
       hit : boolean
       perpID, evt.Bullet.OwnerId : integer

Algoritma
       perpID ← evt.Bullet.OwnerId
       hit ← true
```

```
procedure OnHitBot(HitBotEvent evt)

{Menembak musuh yang sudah tertabrak}

Deklarasi
       closestPoint : Point
       hit : boolean
       GunHeat, closest, closestDir, bearing, gunDirection : double

Algoritma
       TurnRadarRight(45)
       TurnRadarLeft(90)
       closestDir = DirectionTo(closestPoint.X, closestPoint.Y)
       double bearing ← NormalizeRelativeAngle(closestDir -
       GunDirection)
       TurnGunLeft(bearing)
       if GunHeat = 0, then
          Fire(3)
```

```
procedure OnHitWall(HitWallEvent evt)

{Mundur ketika menabrak dinding}

Deklarasi

Algoritma
       Back(50)
```

```
function FindDistance(Point p1, Point p2) → double

{Mengembalikan jarak antara 2 titik}

Deklarasi
       p1.X, p1.Y, p2.X, p2.Y : double

Algoritma
       → Math.sqrt(Math.Pow(p2.X-p1.X, 2) + Math.Pow(p2.Y-p1.Y, 2))
```

4.1.3. Alternatif 3: NayakaBot

Program utamanya pada dasarnya hanya menjalankan tiga fungsi, yakni Gerak(), Tembak(), dan fungsi untuk merotasikan radar supaya dapat memindai seluruh musuh.

```
procedure Gerak()
{Menggerakkan bot berdasarkan total gaya yang dihasilkan}
Deklarasi
    totalGaya : Vektor
    Titik : TitikGaya
    Direction : float

Algoritma
    totalGaya = Vektor(0, 0)
    for each Titik in TitikGaya do
        totalGaya = totalGaya + Titik.Value.Gaya

    Direction = NormalizeRelativeAngle(totalGaya.GetDirection())

    {Mengubah arah bot sesuai dengan vektor gaya}
    SetTurnLeft(Direction)
    WaitFor(NextTurnCondition(this))

    {Bergerak maju sesuai dengan panjang vektor gaya}
    SetForward(totalGaya.GetLength())
    WaitFor(NextTurnCondition(this))
```

```
procedure Tembak()
{Menembak musuh jika kondisi memungkinkan}
Deklarasi
     enemyVelocity, enemyHeading, bulletSpeed, distance, bulletPower
: float
    t : Titik

Algoritma
           if   (not   Mungsuh.IsEmpty()   and   GunHeat   =   0   and
abs(GunTurnRemaining) < 10) then
        enemyVelocity = Mungsuh.Speed
           enemyHeading = Mungsuh.Direction * π / 180   {Convert ke
radian}
        distance = Mungsuh.Distance
        bulletPower = ScaleBulletPower(distance)
        bulletSpeed = 20 - 3 * bulletPower

        {Prediksi posisi musuh}
        t = PredictEnemyPosition()

        {Menyesuaikan arah tembakan dan menembak}
        SetTurnGunLeft(GunBearingTo(t.X, t.Y))
        SetFire(bulletPower)
    return
```

```
function ScaleBulletPower(distance : float) : float
{Menghitung kekuatan peluru berdasarkan jarak musuh}
Deklarasi
    maxPower, maxDistance : float

Algoritma
    maxPower = 3.0
    maxDistance = 1000

    if (distance > maxDistance) then
        distance = maxDistance - 100

    return maxPower * (1 - distance / maxDistance)
```

```
function PredictEnemyPosition() : Titik
{Memperkirakan posisi musuh berdasarkan kecepatan dan arah}
Deklarasi
      EnemyDistance, BulletSpeed, t, enemyVelocity, enemyHeading :
float
    predictedX, predictedY : float

Algoritma
       EnemyDistance = Mtk.CalcDistance(this.X, this.Y, Mungsuh.X,
Mungsuh.Y)
    BulletSpeed = 20 - 3 * ScaleBulletPower(EnemyDistance)
    t = EnemyDistance / BulletSpeed + 5

    enemyVelocity = Mungsuh.Speed
    enemyHeading = Mungsuh.Direction

     predictedX = Mungsuh.X + enemyVelocity * cos(enemyHeading * π /
180) * t
     predictedY = Mungsuh.Y + enemyVelocity * sin(enemyHeading * π /
180) * t

    return (predictedX, predictedY)
```

Selain itu, ada beberapa kelas bentukan yang digunakan. `Mungsuh` (bahasa Jawa dari *musuh*) digunakan untuk menyimpan informasi dari *bot* musuh yang sedang diincar. `TitikGaya` digunakan untuk memodelkan partikel yang memberikan gaya tolak terhadap *bot*, yang pada program utama disimpan dalam bentuk *dictionary*.

```
class MungsuhBot

{untuk menyimpan musuh yang sedang diincar}

Deklarasi
    ID : int
    X : float
    Y : float
    Energy : float
    Direction : float
    Speed : float
    Distance : float

Algoritma
    Method Constructor():
         ID = -1
         X = -1
         Y = -1
         Energy = -1
         Direction = -1
         Speed = -1
         Distance = -1

    { Konstruktor dengan Parameter }
     Method Constructor(id : int, x : float, y : float, energy :
float, direction : float, speed : float, distance : float):
         ID = id
         X = x
         Y = y
         Energy = energy
         Direction = direction
         Speed = speed
         Distance = distance

  { Menghapus data musuh yang tersimpan }
    Method Reset():
         ID = -1
         X = -1
         Y = -1
         Energy = -1
         Direction = -1
         Speed = -1
         Distance = -1

    { Memperbarui data musuh berdasarkan hasil pemindaian }
    Method Update(e : ScannedBotEvent, BotX : float, BotY : float):
         ID = e.ScannedBotId
         X = e.X
         Y = e.Y
         Energy = e.Energy
         Direction = e.Direction
         Speed = e.Speed
         Distance = Mtk.CalcDistance(X, Y, BotX, BotY)

    { Mengecek apakah tidak ada musuh yang sedang diincar }
    Method IsEmpty() : boolean
         Return (ID == -1)
```

```
class Vektor
{merepresentasikan vektor dalam sistem koordinat Robocode}

Deklarasi
    X : float
    Y : float

Algoritma
    { Konstruktor dengan panjang dan arah }
    Method Constructor(length : float, Direction : float):
        Direction = Direction * (π / 180)  { konversi ke radian }
          Direction = Direction - (π / 2)     { sesuaikan ke sistem
koordinat Robocode }
        X = length * cos(Direction)
        Y = length * sin(Direction)

    { Konstruktor dengan informasi posisi dan energi }
      Method Constructor(botX : float, botY : float, botEnergy :
float,
                         musuhX : float, musuhY : float, musuhEnergy :
float):
        w1 = 20
        w2 = 30
        r = Mtk.CalcDistance(botX, botY, musuhX, musuhY)
        h = botEnergy / musuhEnergy
        f = (w1 / r) + (w2 * h)
        Direction = Mtk.CalcAngle(botX, botY, musuhX, musuhY, false)
        Direction = Direction * (π / 180)
        Direction = Direction - (π / 2)
        X = f * cos(Direction)
        Y = f * sin(Direction)

    { Operator overload untuk negasi vektor }
    Method Operator - (a : Vektor) : Vektor
        Return new Vektor(-a.X, -a.Y)

    { Operator overload untuk penjumlahan vektor }
    Method Operator + (a : Vektor, b : Vektor) : Vektor
        Return new Vektor(a.X + b.X, a.Y + b.Y)

    { Operator overload untuk pengurangan vektor }
    Method Operator - (a : Vektor, b : Vektor) : Vektor
        Return new Vektor(a.X - b.X, a.Y - b.Y)

    { Operator overload untuk dot product (perkalian skalar) }
    Method Operator * (a : Vektor, b : Vektor) : float
        Return (a.X * b.X) + (a.Y * b.Y)

    { Operator overload untuk perkalian skalar dengan konstanta }
    Method Operator * (a : Vektor, b : float) : Vektor
        Return new Vektor(a.X * b, a.Y * b)

    { Menghitung panjang (magnitude) vektor }
    Method GetLength() : float
        Return sqrt(X * X + Y * Y)
```

```
    { Menghitung arah vektor dalam radian (disesuaikan dengan sistem
koordinat) }
    Method GetDirection() : float
        Return atan2(Y, X) + (π / 2)
```

```
class TitikGaya
{merepresentasikan gaya dalam bentuk vektor}

Deklarasi
    Gaya : Vektor

Algoritma
    { Konstruktor dengan vektor gaya }
    Method Constructor(gaya : Vektor):
        Gaya = gaya

    { Konstruktor dengan informasi posisi dan energi }
      Method Constructor(botX : float, botY : float, botEnergy :
float,
                         musuhX : float, musuhY : float, musuhEnergy :
float):
        w1 = 20
        w2 = 30
        r = Mtk.CalcDistance(botX, botY, musuhX, musuhY)
        h = musuhEnergy / botEnergy
        f = (w1 / r) + (w2 * h)
            Gaya = new Vektor(f, Mtk.CalcAngle(botX, botY, musuhX,
musuhY, false))
```

```
class Mtk
{menyediakan fungsi matematika untuk perhitungan jarak dan sudut}

Deklarasi
    { Tidak ada atribut }

Algoritma
    { Konstruktor kosong }
    Method Constructor():
        { Tidak melakukan apa-apa }

    { Menghitung jarak antara dua titik }
     Method CalcDistance(x1 : float, y1 : float, x2 : float, y2 :
float) : float
        Return sqrt((x1 - x2)^2 + (y1 - y2)^2)

    { Menghitung sudut antara dua titik }
    Method CalcAngle(x1 : float, y1 : float, x2 : float, y2 : float,
radians : bool) : float
        angle = atan2(y2 - y1, x2 - x1)
        If radians = false then
            angle = angle * (180 / π)
        Return angle
```

4.1.4. Alternatif 4: AmbatuTank

```
procedure MoveToEnemy(targetX, target : double)

{Menembak musuh}

Deklarasi
       angleToEnemy, offset, Direction : double

Algoritma
       angleToEnemy = DirectionTo(targetX, targetY)
       TurnLeft(NormalizeRelativeAngle(angleToEnemy-Direction))
       offset = 45
       SetForward(150);
       SetTurnRight(offset);
       WaitFor(new TurnCompleteCondition(this));
       SetTurnLeft(2 * offset);
       WaitFor(new TurnCompleteCondition(this));
       SetTurnRight(offset);

{TurnCompleteCondition merupakan fungsi tambahan memeriksa sisa giliran bot}
```

```
procedure Tembak()
{Menembak musuh jika kondisi memungkinkan}
Deklarasi
     enemyVelocity, enemyHeading, bulletSpeed, distance, bulletPower : float
    t : Titik

Algoritma
       if (not Mungsuh.IsEmpty() and GunHeat = 0 and abs(GunTurnRemaining) < 10) then
        enemyVelocity = Mungsuh.Speed
           enemyHeading = Mungsuh.Direction * π / 180  {Convert ke radian}
        distance = Mungsuh.Distance
        bulletPower = ScaleBulletPower(distance)
        bulletSpeed = 20 - 3 * bulletPower

        {Prediksi posisi musuh}
        t = PredictEnemyPosition()

        {Menyesuaikan arah tembakan dan menembak}
        SetTurnGunLeft(GunBearingTo(t.X, t.Y))
        SetFire(bulletPower)
       MoveToEnemy(Mungsuh.X, Mungsuh.Y)
    return
```

```
function ScaleBulletPower(distance : float) : float
{Menghitung kekuatan peluru berdasarkan jarak musuh}
Deklarasi
    maxPower, maxDistance : float

Algoritma
    maxPower = 3.0
    maxDistance = 1000

    if (distance > maxDistance) then
        distance = maxDistance - 100

    return maxPower * (1 - distance / maxDistance)
```

```
function PredictEnemyPosition() : Titik
{Memperkirakan posisi musuh berdasarkan kecepatan dan arah}
Deklarasi
      EnemyDistance, BulletSpeed, t, enemyVelocity, enemyHeading :
float
    predictedX, predictedY : float

Algoritma
       EnemyDistance = Mtk.CalcDistance(this.X, this.Y, Mungsuh.X,
Mungsuh.Y)
    BulletSpeed = 20 - 3 * ScaleBulletPower(EnemyDistance)
    t = EnemyDistance / BulletSpeed + 5

    enemyVelocity = Mungsuh.Speed
    enemyHeading = Mungsuh.Direction

     predictedX = Mungsuh.X + enemyVelocity * cos(enemyHeading * π /
180) * t
     predictedY = Mungsuh.Y + enemyVelocity * sin(enemyHeading * π /
180) * t

    return (predictedX, predictedY)
```

Bot menggunakan beberapa kelas yang sama dengan alternatif solusi 3.

4.2. Pengujian

Pengujian dilakukan sebanyak 10 kali yang berupa pertandingan antara semua *bot* dengan keempat alternatif solusi dalam format *classic* atau *Free for All*. Selama pengujian, muncul beberapa kejadian-kejadian unik sebagai berikut:

- Berdasarkan tempat *spawn bot*, alternatif solusi 2 bisa menjadi *bot* paling kuat atau paling lemah dikarenakan kebanyakan *bot* memiliki pergerakan yang bisa menghindar
- *Bot* dengan pergerakan *zig-zag* mampu untuk memutari sebuah *bot* yang diam saja. Walau begitu, *bot* tersebut masih bisa rawan untuk ditembak akibat gerakannya yang mendekati musuh.

4.3. Hasil

Didapatkan bahwa *bot* dengan alternatif solusi *Lazy Survival* mampu mendapatkan skor tertinggi dikarenakan fokusnya untuk *survivability*. Oleh karena itu, alternatif solusi tersebut digunakan sebagai *main bot* kelompok.

5. Penutup

5.1. Kesimpulan

Berdasarkan hasil pembuatan berbagai *bot* dengan pendekatan algoritma *greedy* yang berbeda-beda, dapat disimpulkan bahwa terdapat berbagai macam pendekatan algoritma *greedy* yang bisa memberikan hasil yang berbeda-beda. *Bot* yang paling efektif, tapinya, tetap bergantung pada bagaimana pembuat *bot* mengimplementasikan fungsi-fungsi, algoritma, beserta kelakuan *bot* untuk menjadi pemenang di dalam pertandingan.

5.2. Saran

Sebastian Hung Yansen
Mengerjakan tubes ini jadi sempat nostalgia dikit karena dulu di masa SD pernah ada mata pelajaran robotik. Tubes ini sangat *enjoyable* dan sangat seru, apalagi sampai bisa menimbulkan persaingan dalam kelas sehingga orang-orang berusaha untuk menjadi yang terbaik. Lovyu para asisten <3 <3

Aramazaya
Jujur saya kurang setuju mengenai penggunaan C# pada tubes kali ini dikarenakan lebih banyaknya dokumentasi untuk robocode versi java. *Nevertheless*, tubes ini cukup menyenangkan dan jauh lebih baik dari membuat website. (Please jangan buat website). Selain itu, template dan engine yang sudah disiapkan kak asisten sungguh sangat membantu menghilangkan kebingungan-kebingungan yang akan muncul jika menggunakan source code robocode langsung.

Zulfaqqar Nayaka Athadiansyah
Jujur ini tubes paling *enjoyable* selama di IF setelah tubes Logkom yang *ez* itu. Tenggat waktu pengerjaannya kurang lebih 3 pekan, tapi bisa dikerjakan dengan *nyantai* sambil eksplorasi ke bidang robotika. Yang enak dari tubes kali ini adalah hal-hal yang barangkali ribet buat kebanyakan orang *mostly* sudah disiapkan *preset* atau *template* dari sananya sehingga kami bisa fokus memikirkan *logic* dari botnya saja. *Props to* kakak-kakak asisten mata kuliah IF2211 Strategi Algoritma yang gacor!

Lampiran

Tautan repositori GitHub: https://github.com/Aramazaya/Tubes1_Ambatutank
Tautan video bonus: https://youtu.be/fmNuko_KfQo

No	Poin	Ya	Tidak
1	*Bot* dapat dijalankan pada *engine* yang sudah dimodifikasi asisten.	☑	
2	Membuat 4 solusi *greedy* dengan heuristik yang berbeda.	☑	
3	Membuat laporan sesuai dengan spesifikasi.	☑	
4	Membuat video bonus dan diunggah pada YouTube.	☑	

Daftar Pustaka

Robowiki. *Enemy Dodging Movement*.

O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986, DOI: 10.1177/027836498600500106.

Y. Wang, "Review on Greedy Algorithm," *Theoretical and Natural Science*, vol. 14, pp. 233–239, 2023, DOI: 10.54254/2753-8818/14/20241041.