UTFPR
Engenharia de Computação
Disciplina de Sistemas Inteligentes

Relatório da Atividade 1

Alunos: Guido Margonar Moreira, João Pedro Padoan, Cristian Andre Sanches
Professor orientador: Rafael Mantovani

Março
2023

UTFPR
Engenharia de Computação
Disciplina de Sistemas Inteligentes

Relatório

Relatório da primeira atividade do curso de Sistemas Inteligentes.

Alunos: Guido Margonar Moreira, João Pedro Padoan, Cristian André Sanches

Professor orientador: Rafael Mantovani

Março
2023

Conteúdo

1 Introdução **1**

2 Metodologia **2**
2.1 Wave Surfing 2
2.2 GuessFactor Targeting 7
2.3 Por quê Neo? 10

3 Resultados **11**

4 Conclusão **13**

Bibliografia **14**

1 Introdução

O objetivo desta atividade foi implementar um tanque de guerra virtual controlado por funções e classes implementadas em Java, usando a plataforma Robocode. O Robocode é um jogo desenvolvido em Java cujo objetivo é que os jogadores desenvolvam tanques robôs para enfrentar os tanques desenvolvidos por outros jogadores. Os tanques serão confrontados contra tanques desenvolvidos por outras equipes compostas por alunos da disciplina, e o vencedor (tanto do um-versus-um quanto do melee) receberão pontos adicionais na atividade. Para atingirmos a vitória, criamos rotinas a partir das funções de controle do tanque pensando em tornar o nosso tanque o mais versátil possível, já que não sabemos as estratégias que serão empregadas por outra equipe, e minimizar o fator sorte o máximo que pudemos.

Neste relatório, mostraremos a teoria e implementação das estratégias de combate que escolhemos para o nosso tanque robô. Na seção de metodologia explicamos os algoritmos de Wave Surfing e GuessFactor Targetting, que são algoritmos de movimentação e targetting, respectivamente, já bem conhecidos e amplamente usados no cenário competitivo de Robocode. Na mesma seção, mostramos e comentamos as partes cruciais da implementação dos algoritmos. Na seção de resultados, mostramos os testes que realizamos contra robôs padrões e contra os robôs desenvolvidos pelas equipes do semestre passado de modo a provar a eficácia das estratégias escolhidas.

Figura 1: O Robocode é um jogo de programação onde os jogadores desenvolvem tanque robô para enfrentar contra outros tanques.

2 Metodologia

A principio enquanto aprendíamos a lógica básica sobre o funcionamento do Robocode tinhamos a ideia de fazer um robô capaz de desviar de balas inspirados pela famosa cena do Matrix, porém logo descobrimos que não é possível escanear a posição dos tiros, assim antes de começar a programar o nosso tanque fizemos um levantamento das estratégias de combate utilizadas pelos tanques robôs com maior taxa de vitória no cenário competitivo. No site *LiteRumble* encontramos um ranking dos 1194 melhores tanques no momento ranqueados por porcentagem de vitória. Pesquisando sobre os primeiros colocados observamos uma consistência entre os melhores robôs na escolha do uso do algoritmo de *Wave Surfing* para a movimentação do corpo do tanque. O *DrussGT*, desenvolvido pelo usuário *Skillgannon*, atualmente o #1 no ranking mundial 1v1, usa *Wave Surfing* para movimentação do tanque e *Dynamic Clustering* como sistema de mira. O *Diamond*, atual #2 do ranking mundial 1v1 e desenvolvido por *Voidious*, o administrador do site *Robowiki*, também usa o Wave Surfing para movimentação do tanque e o Dynamic Clustering como sistema de mira, alternando a movimentação para o *Minimum Risk Movement* em melee. O *Neuromancer*, campeão mundial de melee e também desenvolvido por Skillgannon, usa o Wave Surfing para movimentação, adaptado para desviar de balas de múltiplos tanques. Depois de todo este levantamento de dados e vendo a consistência no uso do *Wave Surfing* como sistema de movimentação, optamos por implementar o *Wave Surfing* no nosso robô.

2.1 Wave Surfing

No Robocode o radar dos tanques robôs não conseguem detectar as balas do adversário, de forma que os competidores precisam desenvolver formas de fazê-lo de forma indireta. A técnica de segmentação consiste em encontrar quais ângulos têm maior probabilidade de atingir o alvo pela divisão de um intervalo contínuo de ângulos possíveis em pontos discretos, e encontrando quais "pontos"são mais prováveis. Dessa forma, a técnica de segmentação é implementada pelo rastreamento das balas por meio de ondas (em inglês, waves).

O radar do robô é incapaz de detectar se o inimigo realizou um disparo. Porém, podemos fazer a leitura contínua da energia do oponente, e sabendo que um disparo gasta entre 0.1 e 3.0 pontos de energia, se detectarmos uma baixa de uma magnitude dentro deste intervalo é provável que o inimigo re-

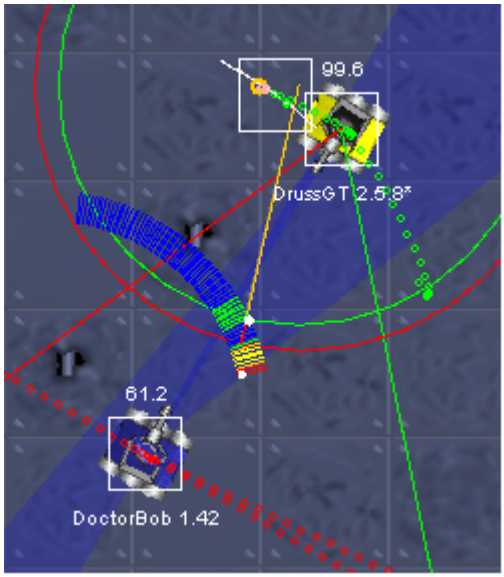

Figura 2: O DrussGT, atual campeão do 1v1 com uma taxa de vitória de 99.92%, usa o algoritmo de *Wave Surfing* como sistema de movimentação do corpo do robô.

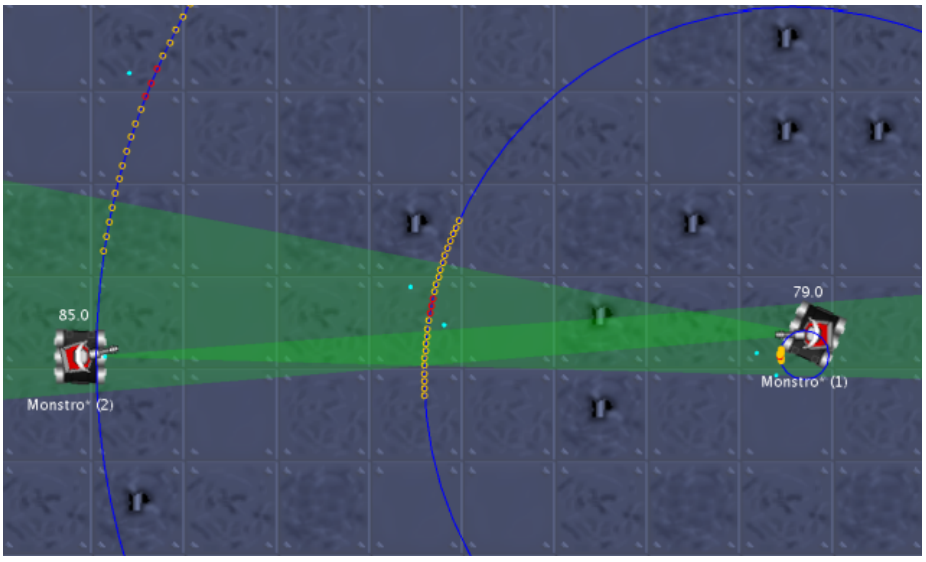

Figura 3: De acordo com o nosso levantamento de dados baseados no ranking competitivo do *LiteRumble*, o *Wave Surfing* é o sistema de movimentação mais vitorioso do Robocode

alizou um disparo. Dessa forma, mantemos um escaneamento constante da energia do inimigo para detectar os seus disparos para que possamos esquivar

deles.

Quando detectamos um disparo, não sabemos a direção do tiro mas sabemos o momento e a velocidade da bala. Dessa forma, podemos inferir que a bala pode estar em qualquer ponto com centro na posição do robô que realizou o disparo e raio do tamanho do produto entre a velocidade e o tempo decorrido desde o início do disparo. Esta circunferência expansiva é uma onda, e nela podemos classificar regiões mais perigosas que outras. Na figura 3 vemos três ondas referentes aos disparos do tanque Monstro(1), com regiões perigosas mostradas em relação ao tanque Monstro(2) que usa o sistema de movimentação *Wave Surfing*. Nesta estratégia movimentamos o tanque de região segura em região segura, evitando as regiões perigosas. Essa forma de movimentação pode ser entendida como surfar uma onda (isto é, a circunferência em expansão), e por isso a estratégia é denominada Wave Surfing (surfando onda, em tradução livre).

```
if (energia_bala < 3.01 && energia_bala > 0.09 && direcoes_surf.size() > 2) {
    OndaPerigosa onda_inimigo = new OndaPerigosa();
    onda_inimigo.tempo_tiro = getTime() - 1;
    onda_inimigo.velocidade_bala = velocidadeBala(energia_bala);
    onda_inimigo.distancia_viajada = velocidadeBala(energia_bala);
    onda_inimigo.sentido_surf = ((Integer)direcoes_surf.get(2)).intValue();
    onda_inimigo.angulo_surf = ((Double)anguloabs_surf.get(2)).doubleValue();
    onda_inimigo.posicao_tiro = (Point2D.Double) posicao_inimigo.clone();
    ondas_perigosas.add(onda_inimigo);
}
```

Figura 4: Detectamos um disparo por meio do escaneamento contínuo da energia do inimigo, verificando se existe uma baixa de energia no intervalo entre 0.1 e 3.0. Quando a condição é verificada, criamos uma nova onda expansível a partir do tempo do tiro e da velocidade da bala (obtida a partir da energia gasta pelo oponente).

Para construir as regiões de perigo na circunferência dividimos o máximo ângulo de escape em setores e armazenamos eles em uma lista. Sempre que o robô é atingido, incrementamos o perigo na posição da lista onde o robô foi atingido, a partir da busca na lista de uma onda com distância e velocidade compatível com o tiro. O valor de perigo adicionado é calculado em relação a posição do inimigo.

Depois de predizermos o tiro e adicionarmos uma nova onda perigosa, atualizamos a nossa lista de ondas perigosas, percorrendo a lista e verificando se a onda já ultrapassou a nossa posição. Caso a condição se verifique,

```
public void onHitByBullet(HitByBulletEvent e){
    if (!ondas_perigosas.isEmpty()){
        Point2D.Double local_acertado_por_bala = new Point2D.Double(e.getBullet().getX(), e.getBullet().getY());
        OndaPerigosa onda_hit = null;
        for (int x = 0; x < ondas_perigosas.size(); x++){
            OndaPerigosa onda_inimigo = (OndaPerigosa) ondas_perigosas.get(x);
            if (Math.abs(onda_inimigo.distancia_viajada -
                posicao_robo.distance(onda_inimigo.posicao_tiro)) < 50
                && Math.abs(velocidadeBala(e.getBullet().getPower()) -
                onda_inimigo.velocidade_bala) < 0.001){
                onda_hit = onda_inimigo;
                break;
            }
        }
        if (onda_hit != null){
            gravarHit(onda_hit, local_acertado_por_bala);
            ondas_perigosas.remove(ondas_perigosas.lastIndexOf(onda_hit));
        }
    }
}
```

Figura 5: Se a onda for igual a uma onda perigosa já armazenada, incrementamos o perigo da onda por meio da função *gravarHit*. Para isso precisamos percorrer a lista de ondas perigosas com complexidade O(n).

removemos a onda da lista. A função relevante, chamada de *atualizarOndas*, pode ser vista na figura 6.

```
public void atualizarOndas() {
    for (int x = 0; x < ondas_perigosas.size(); x++) {
        OndaPerigosa onda_inimigo = (OndaPerigosa) ondas_perigosas.get(x);
        onda_inimigo.distancia_viajada = (getTime() - onda_inimigo.tempo_tiro) *
                                         onda_inimigo.velocidade_bala;
        if (onda_inimigo.distancia_viajada > posicao_robo.distance(onda_inimigo.posicao_tiro) + 50) {
            ondas_perigosas.remove(x);
            x--;
        }
    }
}
```

Figura 6: Se uma onda da lista ultrapassar a posição do robô ela se torna irrelevante, e portanto realizamos a sua remoção da lista de ondas perigosas com complexidade O(n).

Depois disso, surfamos a onda mais próxima. Para isso precisamos descobrí-la na lista de ondas perigosas, e é para isso que serve a função *getOndaMaisProxima*, como pode ser visto na figura 7. Esta função de minimização percorre a lista de ondas, checando a distância do robô a cada uma das ondas perigosas, e achando o valor mínimo. Por fim, a função retorna a onda mais próxima.

A função *surfarOndaMaisProxima*, como mostrado na figura 8, usa a função *getOndaMaisProxima* para encontrar a onda mais próxima e, caso

```
public OndaPerigosa getOndaMaisProxima(){
    double distancia_mais_proximo = 50000;
    OndaPerigosa surfar_onda = null;
    for (int x = 0; x < ondas_perigosas.size(); x++){
        OndaPerigosa onda_inimigo = (OndaPerigosa) ondas_perigosas.get(x);
        double distancia = posicao_robo.distance(onda_inimigo.posicao_tiro) -
                                                    onda_inimigo.distancia_viajada;
        if (distancia > onda_inimigo.velocidade_bala && distancia < distancia_mais_proximo){
            surfar_onda = onda_inimigo;
            distancia_mais_proximo = distancia;
        }
    }
    return surfar_onda;
}
```

Figura 7: Obtemos a onda mais próxima do robô percorrendo a lista de ondas com complexidade O(n). Usaremos ela para fazer com que o robô surfe a onda mais próxima.

ela exista, inicia a rotina de surf. Para surfar nós checamos o perigos para a esquerda e direita e calculamos a distância do inimigo para decidirmos se iremos nos aproximar ou nos afastar dele. Dessa forma, nos movemos para o lado com o menor índice de perigo e nos afastamos ou nos aproximamos do robô do oponente com a função *virarParaTras*. Caso a onda perigosa mais próxima não exista (ou seja, caso o retorno de *getOndaMaisProxima* seja null), simplesmente decidimos se vamos nos aproximar ou nos afastar do inimigo com base em sua distância em relação ao nosso robô.

```
public void surfarOndaMaisProxima(){
    OndaPerigosa surfar_onda = getOndaMaisProxima();

    if (surfar_onda == null){
        double aproximar_ou_afastar = 0;

        if (distancia_inimigo > distancia_maxima){
            aproximar_ou_afastar = virar_aproximar_D;
            double angulo_para_ir = 0;

            if ((posicao_inimigo.x > posicao_robo.x &&
                 posicao_inimigo.y < posicao_robo.y) ||
                (posicao_inimigo.x < posicao_robo.x &&
                 posicao_inimigo.y > posicao_robo.y)){
                angulo_para_ir = suavisarParede(posicao_robo,
                                    anguloAbsolutoDoTarget(posicao_inimigo, posicao_robo) +
                                    aproximar_ou_afastar, 1);
            }else{
                angulo_para_ir = suavisarParede(posicao_robo,
                                    anguloAbsolutoDoTarget(posicao_inimigo, posicao_robo) +
                                    aproximar_ou_afastar, -1);
            }
            virarParaTras(this, angulo_para_ir, 10);-

        }else if (distancia_inimigo < distancia_minima){
            aproximar_ou_afastar = (virar_afastar_D);
            double angulo_para_ir = 0;

            if ((posicao_inimigo.x > posicao_robo.x && posicao_inimigo.y < posicao_robo.y) ||
                (posicao_inimigo.x < posicao_robo.x && posicao_inimigo.y > posicao_robo.y)){
                angulo_para_ir = suavisarParede(posicao_robo, anguloAbsolutoDoTarget(posicao_inimigo, posicao_robo) + aproximar_ou_afastar, 1);
            }else{
                angulo_para_ir = suavisarParede(posicao_robo, anguloAbsolutoDoTarget(posicao_inimigo, posicao_robo) + aproximar_ou_afastar, -1);
            }
            virarParaTras(this, angulo_para_ir, 10);
        }
        return;
    }

    double perigo_esquerda = checarPerigo(surfar_onda, -1);
    double perigo_direta = checarPerigo(surfar_onda, 1);
    double angulo_para_ir = anguloAbsolutoDoTarget(surfar_onda.posicao_tiro, posicao_robo);

    double aproximar_ou_afastar = 0;
    if (distancia_inimigo > distancia_maxima) {
        aproximar_ou_afastar = (virar_aproximar);
    } else if (distancia_inimigo < distancia_minima) {
        aproximar_ou_afastar = (virar_afastar);
    } else {
        aproximar_ou_afastar = (Math.PI/2);
    }
    if (perigo_esquerda < perigo_direta) {
        angulo_para_ir = suavisarParede(posicao_robo, angulo_para_ir - aproximar_ou_afastar, -1);
    } else {
        angulo_para_ir = suavisarParede(posicao_robo, angulo_para_ir + aproximar_ou_afastar, 1);
    }
    virarParaTras(this, angulo_para_ir, 100);
}
```

Figura 8: Surfamos a onda perigosa mais próxima caso ela exista. Se ela não existir, simplesmente nos afastamos ou nos aproximamos do inimigo a partir de um ângulo calculado.

2.2 GuessFactor Targeting

O método *GuessFactor Targeting* é uma heurítica de tiro que funciona de maneira semelhante ao *Wave Surfing*. Porém, ao invés de tentar prever a direção para onde o inimigo irá atirar, o *GuessFactor* consiste em tentar prever para onde o inimigo irá se mover. Para isso, a rotina do *GuessFactor* consiste em criar ondas expansivas a partir dos tiros do nosso robô e monitorá-las para encontrar qual posição é ideal para o mirar a arma quando a onda passa pelo inimigo. Usamos exatamente a mesma estratégia de segmentação da onda do *Wave Surfing*, que explicamos em detalhes na subseção anterior, no *GuessFactor*, subdividindo a onda em áreas de acordo com a probabilidade de acerto do tiro.

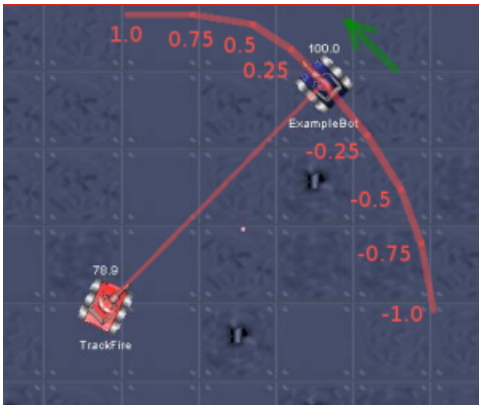

Figura 9: Ângulo máximo de escape dividido em guess factors.

Para implementar o *GuessFactor* criamos uma classe OndaDeTiro com informações sobre um tiro: a posição do robô, a velocidade do tiro, a posição do robô inimigo e o momento tiro. Checamos se o inimigo está se movendo em sentido horário ou anti-horário dentro dos ângulos máximos de escape para onde ele pode se mover. Sempre que um inimigo é detectado a função checarHit é usada para ver se o tiro já teria passado ou atingido o inimigo, e comparamos o valor em que o ângulo em que do tiro com o ângulo ideal que acertaria o inimigo com precisão. Aumentamos o valor do *GUESS_FACTORS* no índice equivalente ao guess factor ideal.

```
public boolean checarHit(double x_inimigo, double y_inimigo, long tempo_atual){

    // Se a distância da origem da onda ao nosso inimigo passou a distância que a bala teria viajado
    if (Point2D.distance(x_inicial, y_inicial, x_inimigo, y_inimigo) <=
       (tempo_atual - tempo_atirar) * getVelocidadeBala()){
        double direcao_desejada = Math.atan2(x_inimigo - x_inicial, y_inimigo - y_inicial);
        double angulo_offset = Utils.normalRelativeAngle(direcao_desejada - startBearing);
        double guessFactor = Math.max(-1, Math.min(1, angulo_offset / anguloMaximoDeEscape()))
                             * direcao;
        int indice = (int) Math.round((segmento_retorno.length - 1) / 2 * (guessFactor + 1));
        segmento_retorno[indice]++;

        return true;
    }
    return false;
}
```

Figura 10: Método *checarHit* da classe *OndaDeTiro*.

Em seguida, deletamos a onda que já passou pelo inimigo dado que ela já foi usada para o ajuste da arma. Fixamos a força do tiro como o poder

de tiro máximo permitido pelo Robocode dividido por 1.5, que foi o valor que rendeu os melhores resultados nos testes contra outros robôs, que serão detalhados na seção de resultados. Checamos se o inimigo está se movendo no sentido horário ou anti-horário e criamos uma onda com as informações atuais do jogador e do inimigo para salvar os dados do tiro atual. Em seguida, consideramos qual índice dos guess factors atuais tem maior chance de atingir o inimigo e convertemos o melhor indice para o formato de -1 a 1, denotando a direção do movimento do inimigo. Considerando o anguloMaximoDeEscape, o guess factor e o sentido do jogador, podemos calcular o ângulo ideal para o realizar o disparo. O ângulo ao qual o canhão precisa se alinhar é calculado comparando o ângulo atual e o ângulo absoluto em relação ao inimigo somado ao ângulo ideal que calculamos anteriormente. Por fim, o canhão se alinha à direção, a onda do tiro é salva em um vetor e, caso a distância esteja dentro do limite escolhido, o nosso robô *Neo* faz o disparo.

```
// Aqui é onde definimos a rotina de tiro do robô
if (energia_inimigo >= 3){
    distancia_inimigo = e.getDistance();

    // Encontrar posição do inimigo
    double x_inimigo = getX() + Math.sin(angulo_absoluto) * distancia_inimigo;
    double y_inimigo = getY() + Math.cos(angulo_absoluto) * distancia_inimigo;

    // Processar as ondas
    for (int i = 0; i < ondas.size(); i++) {
        OndaDeTiro onda_atual = (OndaDeTiro) ondas.get(i);
        if (onda_atual.checarHit(x_inimigo, y_inimigo, getTime())) {
            ondas.remove(onda_atual);
            i--;
        }
    }
    double poder = Math.min(3, Math.max(.1, Rules.MAX_BULLET_POWER/1.5));

    /* Se ele não está movendo não tentamos adivinhar a direção.
       Apenas usamos a direção que obtemos anteriormente */
    velocidade_inimigo = e.getVelocity();
    if (velocidade_inimigo != 0) {
        if (Math.sin(e.getHeadingRadians()-angulo_absoluto)*velocidade_inimigo < 0){
            direcao = -1;
        }else{
            direcao = 1;
        }
    }

    int[] guess_factors_atuais = GUESS_FACTORS;

    OndaDeTiro nova_onda = new OndaDeTiro(getX(), getY(), angulo_absoluto, poder,
            direcao, getTime(), guess_factors_atuais);

    // Inicializamos no meio (30/2)
    int melhor_indice = 15;
    for (int i = 0; i < 31; i++)
        if (guess_factors_atuais[melhor_indice] < guess_factors_atuais[i])
            melhor_indice = i;

    double guessfactor = (double) (melhor_indice-((GUESS_FACTORS.length-1)/2) /
                                ((GUESS_FACTORS.length-1)/2);
    double angulo_offset = direcao * guessfactor * nova_onda.anguloMaximoDeEscape();
    double ajustar_arma = Utils.normalRelativeAngle(angulo_absoluto - getGunHeadingRadians() +
                                                    angulo_offset);
    setTurnGunRightRadians(ajustar_arma);

    ondas.add(nova_onda);
    if (distancia_inimigo < distancia_maxima_tiro){
        setFire(Rules.MAX_BULLET_POWER/1.5);
    }

}else{
    setTurnGunRightRadians(robocode.util.Utils.normalRelativeAngle(angulo_absoluto - getGunHeadingRadians()));
    setFire(Rules.MAX_BULLET_POWER/1.5);
}
```

Figura 11: Rotina de tiro usando o algoritmo de guess factors.

2.3 Por quê Neo?

Neo é o nome que decidimos dar ao nosso robô. Pelo fato de nosso tanque se esquivar das balas maravilhosamente bem usando o Wave Surfing, chegamos a conclusão que ele se assemelha a um personagem do Matrix adepto em artes marciais metafísicas, do qual o Neo é o mais belo exemplo.

Figura 12: Neo, protagonista do filme Matrix, se esquivando de uma saraivada de balas.

3 Resultados

Como mostramos em detalhes na seção de metodologia, tivemos sucesso em implementar os sistemas de movimentação e targeting que nos propomos a estudar. Dado o critério da prática, testamos o nosso robô com a implementação completa do Wave Surfing e GuessFactor Targeting em confrontos contra os robôs desenvolvidos pelas equipes do semestre passado, usando uma grande quantidade de rounds (10000 rodadas) para minimizar ao máximo o fator sorte. O resultados dos confrontos podem ser vistos abaixo nas figuras 11 até 16.

Também testamos o *Neo*, nosso robô, contra os robôs do semestre passado na modalidade melee. No melee o *Neo* tem um desempenho relativamente ruim por que o *Wave Surfing* implementado só consegue levar em conta os tiros de um inimigo por vez. Dessa forma, enquanto o *Neo* escolhe um alvo e desvia de seus tiros, ele acaba recebendo muito dano dos tiros dos outros robôs. Para melhorar o nosso robô no melee poderíamos implementar um algoritmo de Wave Surfing para combate contra mais de um oponente. Para melhorar o sistema de targeting poderíamos implementar o *Dynamic Clustering*, um algoritmo bastante complicado de mira usado por alguns dos melhores tanques robôs do cenário competitivo, mas faltou tembo hábil e por isso optamos por implementar o GuessFactor. Almeida (2017), o TCC de um brasileiro que desenvolveu um robô para o cenário competitivo, também escolheu implementar o *GuessFactor Targeting* ao invés do *Dynamic Clustering*.

Results for 10000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	Matrix.Neov3*	1322772 (78%)	476800	95360	628653	121484	425	50	9541	459	0
2nd	mantova.CONTRA4*	366749 (22%)	22950	4590	331330	7439	386	53	464	9536	0

Save OK

Figura 13: Resultado do teste na modalidade 1x1 contra o robô CONTRA4.

Results for 10000 rounds

Rank	Robot Name	Total Score...	Survival	Surv Bonus	Bullet D...	Bullet Bonus	Ram Dmg ...	Ram Bo...	1...	2n...	3r...
1st	Matrix.Neov3*	1331978 (79%)	478850	95770	633819	122700	782	57	9579	421	0
2nd	mantova.SAM...	350086 (21%)	21050	4210	317743	7039	43	0	423	9577	0

Save OK

Figura 14: Resultado do teste na modalidade 1x1 contra o robô SAMU.

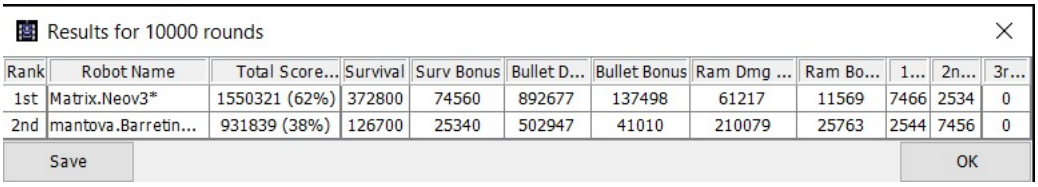

Figura 15: Resultado do teste na modalidade 1x1 contra o robô Barretinho.

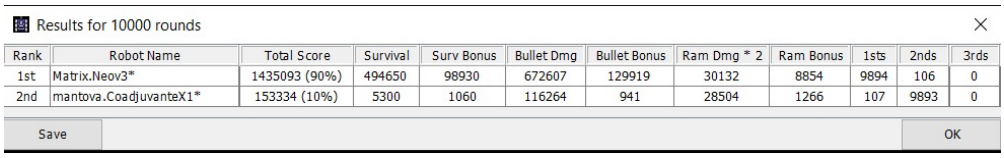

Figura 16: Resultado do teste na modalidade 1x1 contra o robô CoadjuvanteX1.

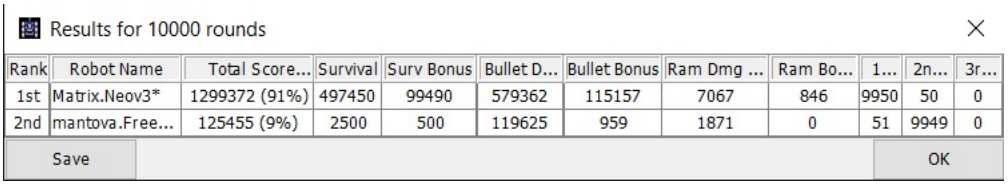

Figura 17: Resultado do teste na modalidade 1x1 contra o robô Freeza.

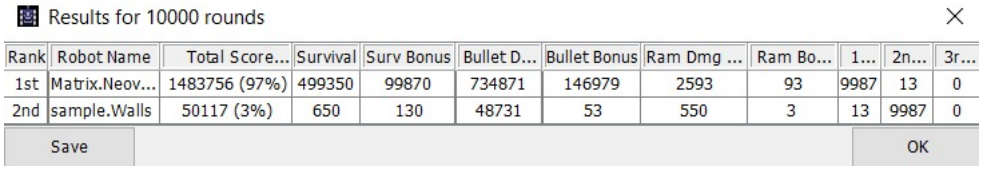

Figura 18: Resultado do teste na modalidade 1x1 contra o robô Walls.

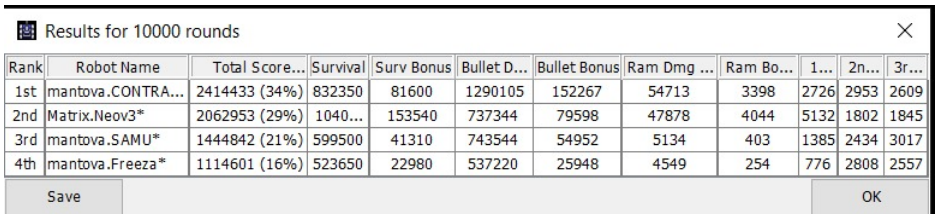

Figura 19: Resultado do teste na modalidade melee contra os robôs.

4 Conclusão

Em conclusão, dado o resultado das simulações referentes ao robô *Neo*, podemos notar padrões de comportamento durante batalha, sendo estes a maneira como ele calcula uma possível posição do inimigo e atira tendo um ângulo referente há previsão baseado no angulo ideal para os tiros anteriores que são salvos em *GUESS_FACTORS*, se afastar do inimigo para manter um padrão de desvio referente ao tiro, a característica de girar em torno do inimigo para manter distancia segura e permitir um tiro mais certeiro, a busca de inimigos mais próximo e a tendencia de, caso esteja próximo de mais da parede, desvie para o caminho contrario.

Em simulações de duelo o robô *Neo* se mantém estável, com porcentagens de vitoria mais altas, mas em simulações de combate com mais de um inimigo o robô *Neo* não produz muita eficiência, pois ele se mantém na base de desvio do inimigo analisado. Com base na coleta de dados simulados em combate com mais de um inimigo, mesmo com um padrão de desvio de tiro no final do round, o *Neo* não mantém a forma de desvio de mais de um inimigo disparando em sentidos opostos. Sendo assim, quando encurralado o Neo não consegue prever escapatória para desviar.

Como objetivo foi criar um robô com a capacidade de desviar de balas aleatórias como podemos ver no filme Matrix, o projeto tentou se aproximar dessas características, mas por meio da análise dos resultados em um teste melee contra vários robôs oponentes, o *Neo* consegue lidar bem contra apenas um inimigo. Para melhorar o nosso projeto poderíamos implementar funções de localização de dano do robô e o uso do radar como função de dupla resposta, para manter a coerência com o dano obtido e o a realização do desvio para um local onde o número de inimigos fosse menor.

Bibliografia

ALMEIDA, E. V. G. Desenvolvendo um Robô Competitivo para Robocode. Salvador, Brasil, UFBA. 2017.

ROBOWIKI. Wave Surfing. 2012. Disponível em: <http://robowiki.net/wiki/Wave_surfing>. Acesso em: 25 de Março de 2018.

ROBOWIKI. GuessFactor Targeting (traditional). 2017. Disponível em: <http://robowiki.net/wiki/Guess_Factor_Targeting>. Acesso em: 25 de Março de 2018.