

Controlador de IA — Gladiator Arena

Trabalho 1 — INF01048

Prof. Bruno Castro da Silva
bsilva@inf.ufrgs.br

Gabriel Marangoni Moita
gmmoita@inf.ufrgs.br

Marcos Praisler de Souza
mpsouza@inf.ufrgs.br

Roberta Robert
rrobert@inf.ufrgs.br

1 Objetivo

O objetivo deste trabalho é utilizar técnicas de Inteligência Artificial para desenvolver um controlador capaz de controlar um robô gladiador dentro de uma arena, na qual compete até a morte contra outro robô. O programa de IA a ser desenvolvido deverá ser capaz de decidir qual melhor ação tomar nas diversas situações possíveis, levando em consideração algumas informações sobre o inimigo.

As técnicas de IA a serem utilizadas para resolução deste trabalho ficam a critério de cada trio, mas deverão envolver os métodos de Busca Local estudados na disciplina, como Hill Climbing, Simulated Annealing, Busca em Feixe Local e algoritmos genéticos. **É obrigatória a implementação de pelo menos 3 (três) algoritmos.** Os grupos poderão, alternativamente, utilizar variantes destes métodos (ou outros métodos de busca local), caso achem apropriado.

O controlador a ser desenvolvido **não** pode ser manualmente codificado; isto é, não é permitido que se programe manualmente um software que verifica a condição atual do Gladiador e determina a melhor ação a ser executada. O objetivo é que tal controlador seja *aprendido* de forma automática utilizando-se alguns dos métodos de aprendizado estudados na disciplina.

2 Enredo Fictício — A Arena de Gladiadores

Em outubro de 2020 foi firmado pela ONU, em uma reunião de emergência, um acordo internacional que proibia todo e qualquer esporte que envolvesse agressões físicas entre humanos. Sendo assim, estavam banidos esportes como MMA, boxe, luta greco-romana, taekwondo, esgrima, *paintball*, *airsoft*, entre outros, vistos os acontecimentos lamentáveis ocorridos durante as Olimpíadas de Verão daquele ano, no Japão.

Essa decisão causou grande revolta entre os entusiastas dessas modalidades. Uma grande discussão começou, buscando definir algum tipo de atividade que pudesse trazer a mesma euforia aos espectadores que os esportes vetados traziam.

Um grupo de empresas japonesas, baseado em filmes de ficção científica como *Robot Jox* (1990), lançou, em 2021, a primeira linha de robôs gladiadores. Era permitido que robôs controlados por Inteligências Artificiais lutassem entre si. Os investidores apoiaram a ideia, criando o Circuito Internacional de Arena de Gladiadores, contando com diversas arenas ao redor do mundo, todas preparadas para receber as mais violentas batalhas de robôs. Atualmente, os diversos campeonatos ao redor do mundo movimentam mais de 20 bilhões de dólares por ano.

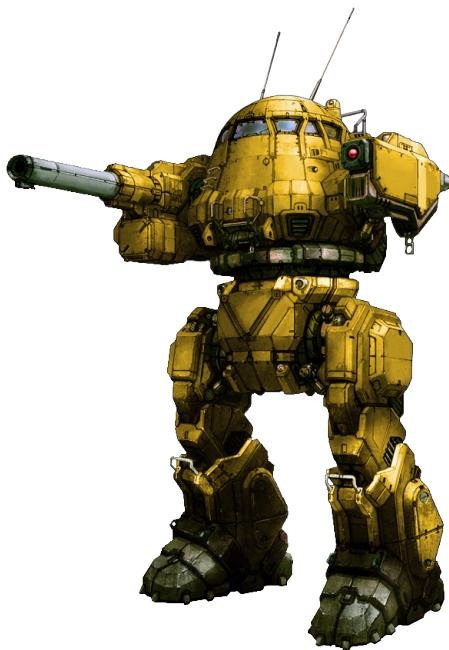


Figure 1: Modelo de robô gladiador equipado com um rifle de precisão e com detectores de distância de gladiadores inimigos e de projéteis. O robô possui campo de visão limitado.

3 O Simulador — Gladiator Arena

Neste trabalho iremos utilizar uma simulador virtual da Arena de Gladiadores, o qual é utilizado pelas equipes para treinar suas Inteligências Artificiais antes de transferi-las para os robôs reais. O modelo simplificado que utilizaremos (Figura 1) consiste em uma simulação bidimensional na qual um personagem circular controlado pela Inteligência Artificial deve batalhar até a morte contra outros *bots*. É possível enviar quatro tipos de comandos ao robô: para que o robô rotacione no sentido horário (R), rotacione no sentido anti-horário (L), não se mova (\emptyset) ou atire (S). Por default, o robô sempre se move para frente, com velocidade constante.

O software controlador do Gladiador envia 30 comandos por segundo ao robô. Dado que o objetivo do robô é derrotar o gladiador inimigo, um controlador eficaz deve levar em conta o estado atual do robô e do inimigo a fim de determinar quais ações/comandos são mais apropriados. Por exemplo, se o Gladiador estiver com o inimigo na mira e próximo a ele, deve ser aplicado um comando de atirar, a fim de tentar atingi-lo. Neste trabalho, nós assumimos que o estado s_t do Gladiador no tempo t consiste em uma tupla de números $s_t(t) = [d_{\text{inimigo}}, v_{\text{inimigo}}, d_{\text{bala}}, v_{\text{bala}}]$, os quais correspondem aos valores lidos pelos sensores do robô no tempo t . Aqui, d_{inimigo} corresponde à distância do inimigo, v_{inimigo} corresponde à presença (1) ou não (0) do inimigo no campo de visão, d_{bala} corresponde à distância da bala mais próxima e v_{bala} corresponde à presença (1) ou não (0) de uma bala no campo de visão. A distância do inimigo e da bala mais próxima são fornecidas independentemente delas estarem dentro do campo de visão do robô. Caso não haja nenhuma bala disparada, o sensor d_{bala} retorna a maior distância possível em que uma bala poderia estar.

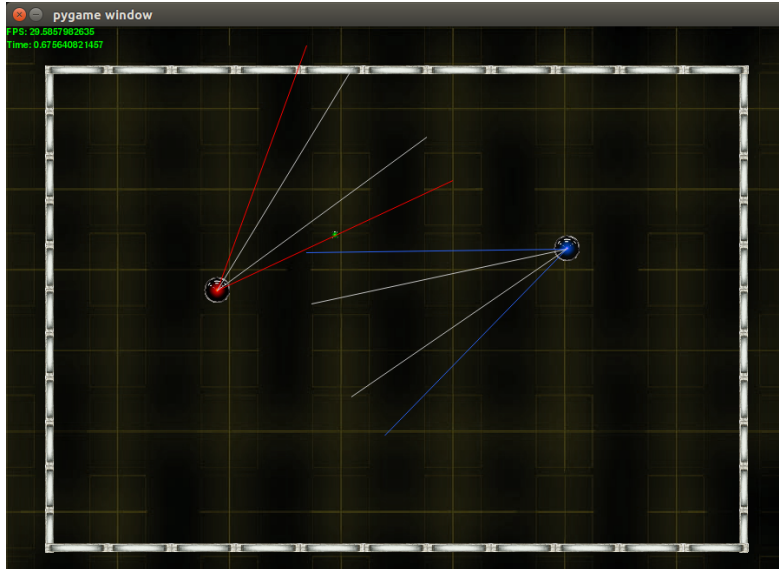


Figure 2: O simulador *Gladiator Arena*.

A interface gráfica do simulador (Figura 2) apresenta vários elementos: os dois robôs gladiadores, e também, associado a cada um dois cones: o cone de visão (*fov*) e o cone de tiro (*sight*). O cone de visão possui um ângulo de $\frac{\pi}{4}$ radianos de abertura (45 graus) e delimita a área na qual o gladiador pode ver inimigos ou balas. O cone de tiro possui um ângulo de $\frac{\pi}{8}$ radianos de abertura (22.5 graus) e delimita os ângulos nos quais o tiro poderá ser atirado; a direção exata de um tiro disparado, dentro do cone, é escolhida aleatoriamente no momento de um comando de disparo.

4 Treinamento de um Controlador

O objetivo deste trabalho é utilizar técnicas de Busca Local estudadas em aula para aprender um controlador para o Simulator Arena. O controlador a ser desenvolvido será uma função $\pi_\theta(s_t)$, a qual recebe como entrada o estado do robô no tempo t e devolve uma ação dentre $\{R, L, \emptyset, S\}$, onde:

- R corresponde ao comando para Girar no sentido horário (comando 1);
- L corresponde ao comando Girar no sentido anti-horário (comando 2);
- \emptyset corresponde ao comando de não executar nenhuma ação; o agente não se move (comando 3);
- S corresponde ao comando Atirar (comando 4).

A função π_θ , portanto, deverá ser aprendida para que consiga, com base em informações a respeito do inimigo e da bala mais próxima, determinar uma ação apropriada, buscando vencer a batalha o mais rápido possível. Considere a situação abaixo, na qual os gladiadores estão virados em direções opostas:

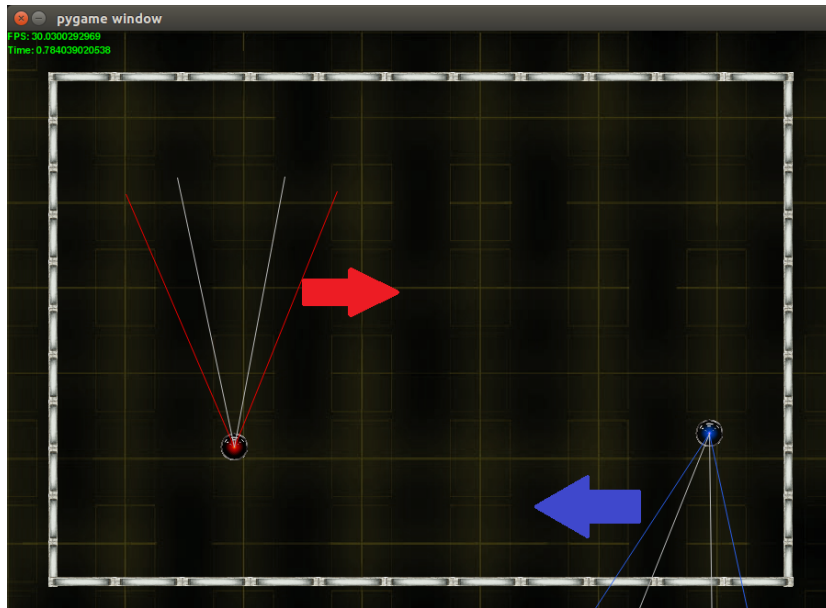


Figure 3: Possível estado durante uma batalha.

Nessa situação, a melhor ação a ser retornada pelo controlador $\pi_\theta(s_t)$ do agente azul seria girá-lo no sentido horário, a fim de colocar o inimigo em seu campo de visão e permitir um disparo.

Perceba que o controlador π_θ , descrito acima, depende de parâmetros θ . Neste trabalho, definimos θ como um vetor de N números reais: $\theta = [\theta_1, \dots, \theta_N]$. Dependendo dos valores destes parâmetros, o controlador π_θ irá retornar diferentes ações para cada estado. O objetivo do trabalho é aprender os valores dos parâmetros que, quando utilizados pelo controlador π_θ , permitem com que o gladiador vença a batalha. Note que isso corresponde a um problema de otimização: desejamos maximizar uma função $\text{Valor}(\theta)$, a qual avalia a qualidade de um controlador π parametrizado por θ , de forma que, quanto mais rápido o controlador π_θ conseguir derrotar o inimigo, maior será $\text{Valor}(\theta)$. Portanto, qualquer processo de busca local que identifique os parâmetros θ que maximizam $\text{Valor}(\theta)$ pode ser utilizado.

A fim de permitir o aprendizado deste controlador, precisamos, primeiramente, decidir como a função π_θ será implementada computacionalmente. Uma maneira simples de implementá-la é a seguinte. Seja $Q_\theta(s_t, A_i)$ uma função que retorna um valor numérico denotando a *preferência* do Gladiador por executar uma ação A_i , quando este se encontra no estado s_t . Quanto maior $Q_\theta(s_t, A_i)$, maior será a preferência do Gladiador pela execução desta ação, ou seja, mais ele julga a ação A_i como sendo a mais apropriada naquele estado da batalha. Se o estado do robô for, por exemplo, $s_t = [10, 1, 100, 0]$, isso indica que o Gladiador encontra-se a 10 unidades de distância do inimigo, que o inimigo está em seu campo de visão, que existe uma bala a 100 unidades de distância dele, e que ela não está em seu campo de visão. A melhor ação, neste caso, seria atirar, visto que o robô tem o inimigo em sua mira. Isso significa que $Q_\theta(s_t, S)$ deverá ser um valor positivo grande. Podemos definir, de maneira semelhante, funções de preferência para as ações R , L e \emptyset : $Q_\theta(s_t, R)$, $Q_\theta(s_t, L)$ e $Q_\theta(s_t, \emptyset)$, respectivamente. Note que se tivermos acesso aos valores de preferência corretos para cada uma das cinco possíveis ações, quando em um estado s_t , o controlador $\pi_\theta(s_t)$ poderá simplesmente escolher a ação com maior valor de preferência:

$$\pi_\theta(s_t) = \arg \max_{A_i \in \{R, L, S, \emptyset\}} Q_\theta(s_t, A_i).$$

uma vez que isso, por definição, significa que ele estaria escolhendo as ações mais apropriadas para cada estado s_t .

A pergunta natural, agora, é como iremos representar as funções de preferência Q_θ . Uma possibilidade é defini-las de forma que dependa de uma variável (que chamaremos de *prox*) que indica a proximidade do inimigo, caso ele esteja no campo de visão do robô, e que é zero caso contrário: $prox = (1/d_{\text{inimigo}}) \times v_{\text{inimigo}}$. Note que esta variável é calculada com base nos sensores do agente, os quais são armazenados em seu estado s_t . A este tipo de variável, chamamos de uma *feature*. *Features* são, portanto, valores que calculamos com base nos sensores do agente (cujos valores são armazenados em seu estado s_t), e com base nos quais o robô irá calcular sua preferência por cada uma das ações. O uso deste tipo de *feature* para o cálculo da preferência de cada uma das ações resultaria nas seguintes funções de preferência:

$$\begin{aligned}
Q_\theta(s_t, R) &\equiv Q_\theta(d_{\text{inimigo}}, v_{\text{inimigo}}, d_{\text{bala}}, v_{\text{bala}}, D) &= \theta_1 + (\theta_2 \times \text{prox}) \\
Q_\theta(s_t, L) &\equiv Q_\theta(d_{\text{inimigo}}, v_{\text{inimigo}}, d_{\text{bala}}, v_{\text{bala}}, E) &= \theta_3 + (\theta_4 \times \text{prox}) \\
Q_\theta(s_t, S) &\equiv Q_\theta(d_{\text{inimigo}}, v_{\text{inimigo}}, d_{\text{bala}}, v_{\text{bala}}, S) &= \theta_5 + (\theta_6 \times \text{prox}) \\
Q_\theta(s_t, \emptyset) &\equiv Q_\theta(d_{\text{inimigo}}, v_{\text{inimigo}}, d_{\text{bala}}, v_{\text{bala}}, \emptyset) &= \theta_7 + (\theta_8 \times \text{prox})
\end{aligned}$$

Aqui, onde os coeficientes θ correspondem aos parâmetros que deverão ser aprendidos pelo algoritmo de busca local. Ao se ajustar os $N = 8$ parâmetros $\theta_1, \dots, \theta_8$, define-se diferentes funções de preferência pelas ações R , L , S e \emptyset , o que implica que o controlador irá controlar o Gladiador de maneiras diferentes dependendo do valor de seu estado s_t (e, conseqüentemente, da *feature prox*). Algumas destas maneiras de reagir ao estado do agente podem ser mais eficazes para que se vença a batalha, e outras nem tanto. Parâmetros que resultem em preferências corretas implicam que o Gladiador irá escolher ações apropriadas para que, dado o seu estado s_t atual (e, portanto, as *features* calculadas com base nele), o robô consiga se aproximar de uma vitória. Qualquer processo de busca local que maximize $\text{Valor}(\theta)$, portanto, resultará no aprendizado das preferências Q_θ de forma que as ações mais apropriadas em cada estado tenham maior preferência.

Vamos pensar, agora, de que forma a *feature prox* influenciaria na preferência do agente por atirar. Se o inimigo não estiver visível, $\text{prox} = 0$, e portanto a preferência $Q_\theta(s_t, S) = 0$. Se o inimigo estiver visível, por outro lado, então prox retornará a proximidade deste inimigo. Suponha que θ_8 é um número positivo; nesse caso, a preferência por atirar aumentará (será um número positivo maior) conforme proximidade do inimigo aumenta. Se θ_8 for negativo, por outro lado, a preferência por atirar diminuirá (se tornará um número mais *negativo*), conforme a proximidade do inimigo aumenta. Essas duas possibilidades são ilustradas na Figura 4 e discutidas a seguir.

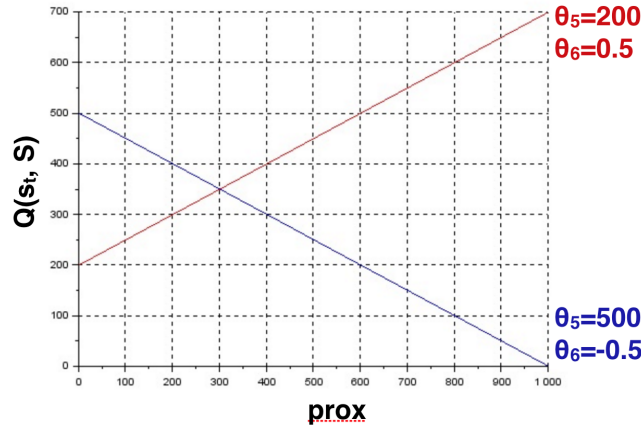


Figure 4: Exemplo do efeito de diferentes pesos θ_i na função $Q_\theta(s_t, S)$.

A Figura 4 mostra como a preferência do agente por atirar varia em função do valor da *feature prox*. A curva vermelha representa a preferência por atirar, caso o controlador esteja utilizando parâmetros $\theta_5 = 200$ e $\theta_6 = 0.5$. Note que, neste caso, a preferência por atirar aumenta com a proximidade do inimigo. A curva azul, por outro lado, representa a preferência por atirar caso o controlador utilize parâmetros $\theta_5 = 500$ e $\theta_6 = 0.5$. Nesse caso, a preferência por atirar *diminui* com a proximidade do inimigo. Um controlador que utilizasse os parâmetros da curva vermelha, portanto, provavelmente teria maior Valor (melhor performance), visto que corresponderia a um robô que atira quando o inimigo está mais próximo.

Note que a representação acima para a preferência por atirar possui uma limitação importante: o controlador apenas leva em consideração a proximidade do inimigo! O grupo de alunos poderia pensar em outras *features*: que informações do estado (e possivelmente do estado anterior) o agente precisaria combinar para obter *features* que representassem outras informações úteis para escolha da ação? Por exemplo, como seria calculada uma *feature* que representasse um valor indicando que há uma bala se aproximando do agente? Ou uma *feature* que representasse um valor indicando que o tiro do inimigo errou o gladiador? Ou uma *feature* que indicasse que o inimigo está próximo mas não está visível?

Além do desenvolvimento do algoritmo de busca local, cada grupo também deverá definir um conjunto de *features* relevantes que sejam capazes de comunicar ao agente as todas informações que podem ser úteis para determinar a melhor ação a cada momento, dependendo do que está ocorrendo. Inimigo se aproximando? Inimigo errou tiro? Bala em rota de colisão com o agente? Inimigo na mira? Note que essas *features* são combinações das informações percebidas pelo agente através de seus sensores. Digamos que o grupo de alunos tenha definido 4 *features*, f_1, \dots, f_4 , calculadas utilizando as informações dos sensores no estado s_t do robô. Nesse caso,

$$\begin{aligned} Q_\theta(s_t, R) &= \theta_1 + (\theta_2 \times f_1) + (\theta_3 \times f_2) + (\theta_4 \times f_3) + (\theta_5 \times f_4) \\ Q_\theta(s_t, L) &= \theta_6 + (\theta_7 \times f_1) + (\theta_8 \times f_2) + (\theta_9 \times f_3) + (\theta_{10} \times f_4) \\ Q_\theta(s_t, S) &= \theta_{11} + (\theta_{12} \times f_1) + (\theta_{13} \times f_2) + (\theta_{14} \times f_3) + (\theta_{15} \times f_4) \\ Q_\theta(s_t, \emptyset) &= \theta_{16} + (\theta_{17} \times f_1) + (\theta_{18} \times f_2) + (\theta_{19} \times f_3) + (\theta_{20} \times f_4) \end{aligned}$$

Note que como estamos, neste exemplo, utilizando 5 pesos por função de preferência, e como há 4 funções de preferência Q (uma para cada ação), teremos um total de $5 \times 4 = 20$ parâmetros θ a serem aprendidos, ao invés de apenas $N = 8$, como era o exemplo anterior onde havia apenas uma única *feature* (*prox*). Isso indica que, se quisermos utilizar funções de preferência mais complexas, que determinam a preferência por cada ação com base em mais informações (mais *features*), pagaremos o preço através de um maior número de parâmetros que precisam ser aprendidos—o que torna o aprendizado mais lento.

Durante o processo de desenvolvimento do seu gladiador, você poderá perceber que diferentes tipos de *features* podem auxiliar ou prejudicar o aprendizado. Um dos objetivos deste trabalho é experimentar com tipos diferentes de *features* e determinar quais permitem o aprendizado rápido de um controlador eficaz. Note que, por um lado, precisamos utilizar um número suficiente de *features*, de forma a permitir com que as funções de preferência tenham a seu dispor todo tipo de informação relevante para o cálculo da preferência por uma ação. Por outro lado, não desejamos que o número de *features* seja alto a ponto de tornar o processo de aprendizado/busca local excessivamente lento. O trio deverá experimentar diversos tipos de *features* f_i , calculadas com base nos sensores do robô e armazenadas em seu estado s_t , e descobrir quais resultam em aprendizado mais rápido de um controlador de alta performance. O papel do algoritmo de aprendizado será, então, descobrir o peso (θ_i de cada uma destas *features*, de forma que, quando combinadas como no exemplo acima, façam com que o agente tenha preferência pelas ações que maximizam sua performance; em outras palavras, que o ajudem a vencer as batalhas o mais rápido possível.

A função de Valor(θ) que iremos utilizar neste trabalho, e que mede a performance do robô, é calculada da seguinte forma: um robô que utiliza o controlador com parâmetros θ é colocado em batalha contra outro *bot*, e o simulador observa o tempo que o robô conseguiu permanecer vivo (*survival_time*) e o tempo até ele acertar um tiro (*hit_time*). *max_time* é o tempo máximo de uma batalha. Define-se que é $hit_time = max_time$ caso o robô não tenha conseguido atingir o inimigo com um tiro, e que $survival_time = max_time$ caso ele tenha conseguido atingir um tiro (e, portanto, vencer a batalha). A performance é calculada pela seguinte fórmula:

$$\text{Valor}(\theta) = 0.5 \times \left(\frac{\text{survival_time}}{\text{max_time}} \right) + 0.5 \left(\frac{\text{max_time} - \text{hit_time}}{\text{max_time}} \right) - 0.5$$

Note que a performance de um controlador estará sempre no intervalo $[-0.5, +0.5]$, onde $+0.5$ seria uma vitória perfeita (o robô atinge o outro logo no início da batalha: $hit_time = 0$ e $survival_time = max_time$), 0 corresponde a um empate ($hit_time = max_time$ e $survival_time = max_time$) e -0.5 a uma derrota perfeita (o robô é atingido no início da batalha: $hit_time = max_time$ e $survival_time = 0$). Em outras palavras, a vitória perfeita seria o gladiador derrotar seu oponente sem passar nenhum segundo, a derrota perfeita seria o gladiador ser derrotado sem passar nenhum segundo, e o empate é o tempo limite ter sido atingido (30 segundos, o que equivale a 900 ações). Se o gladiador vence a partida, terá performance acima de zero: quanto mais próxima de 0.5 , mais rápido o robô venceu a batalha. Se o gladiador perde a partida, terá performance abaixo de zero: quanto mais próxima de -0.5 , mais rápido ele foi derrotado. Neste trabalho, definimos que caso ocorra uma colisão entre os gladiadores, ambos perdem com penalidade máxima: performance -0.5 .

Uma pergunta ainda não foi respondida: ao treinar o seu robô em partidas simuladas, contra que tipo de *bot* ele irá competir? Caso você faça ele competir com um robô que sempre executa ações aleatórias, provavelmente não será difícil de aprender um controlador que vença todas as partidas. Caso você faça ele competir com um *bot* que sempre consegue desviar de todos os tiros, e sempre mantém a distância, será extremamente difícil vencer. É importante, portanto, perceber a influência do *bot* inimigo escolhido para ser utilizado durante o processo de aprendizado.

Nós disponibilizamos vários *bots* que implementam comportamentos simples, e que você pode utilizar durante o processo de aprendizado. O *bot* chamado de *lazy_bot*, por exemplo, sempre executa a ação *R*, e é, portanto, facilmente derrotável. O *bot* *ninja_bot*, por outro lado, possui um comportamento bem mais complexo, e é mais difícil de ser vencido. Note que em ambos os casos, o seu algoritmo de aprendizado estaria tentando achar parâmetros θ que maximizam o desempenho do robô (permitem uma vitória rápida). Entretanto, um agente treinado contra o *lazy_bot* provavelmente não teria chance contra um agente treinado contra o *ninja_bot*, pois ele nunca teria tido a chance de enfrentar um inimigo com comportamento mais complexo e eficiente, e portanto aprenderia apenas técnicas de luta mais simples.

Considerando isso, para fins de competitividade, é **permitido** aos trios implementar outros *bots* mais complexos dos que os disponibilizados, ou que possuam algum padrão de comportamento que desejem que o seu robô aprenda a derrotar. Detalhes de como fazer isso podem ser encontrados na *documentação técnica*, ao final deste documento. Os *bots* fornecidos são apenas *bots* de exemplo. Como haverá uma etapa de competição contra outros agentes desenvolvidos pelos outros trios, é sugerida a implementação de melhores *bots* para que, através do aprendizado contra esses *bots* mais eficientes, o grupo aumente a chance de seu robô ser competitivo na fase de enfrentamentos. A ideia é que se forçar seu robô a treinar contra *bots* mais competentes, ele estará preparado para lutar contra oponentes também mais eficientes.

A fim de melhorar a performance do seu robô, portanto, você deve investir em três frentes: 1) desenvolvimento de um algoritmo de busca local eficiente, que encontre soluções com alta performance; 2) desenvolvimento de boas *features*, que sejam relevantes para que o robô consiga determinar qual a melhor ação a cada momento; e 3) desenvolvimento de *bots* de treinamento eficientes, a fim de permitir com que seu robô pratique contra robôs eficazes e portanto aprenda a lidar com uma grande variedade de possíveis comportamentos competitivos, por parte de outros agentes.

5 Normalização de Features

Uma prática comum na implementação de Inteligências Artificiais é a **normalização de features**. O problema que tal normalização resolve é que *features* diferentes podem ter faixas de valores muito distintas. Por exemplo, imagine que estamos utilizando duas *features*: d_{inimigo} e $d_{\text{inimigo}} \times d_{\text{bala}}$. Considerando que tanto d_{inimigo} e d_{bala} variam no intervalo $[0, 1000]$, a *feature* $d_{\text{inimigo}} \times d_{\text{bala}}$ variará no intervalo $[0, 1000000]$. Qual o problema? Vejamos os efeitos disso na seguinte função de preferência:

$$Q_{\theta}(s_t, F) = (\theta_1 \times d_{\text{inimigo}}) + (\theta_2 \times (d_{\text{inimigo}} \times d_{\text{bala}}))$$

O primeiro termo dessa soma está multiplicando um peso por um número na faixa das centenas, e o segundo termo da soma está multiplicando um peso por um número que pode ser até 1 milhão. Isso significa que, ao se fazer a soma para cálculo da preferência, o valor da preferência dependerá *muito* mais da *feature* $d_{\text{inimigo}} \times d_{\text{bala}}$ do que da *feature* d_{inimigo} . Suponha, por exemplo, que $\theta_1 = \theta_2 = 1$, e que $d_{\text{inimigo}} = 50$ e $d_{\text{bala}} = 1000$; neste caso, o valor de $Q_{\theta}(s_t, F)$ será:

$$Q_{\theta}(s_t, F) = (1 \times 50) + (1 \times (50 \times 1000)) = 50,050$$

Desse total, a imensa maioria (50,000) se deve ao imenso valor resultante do uso da *feature* $d_{\text{inimigo}} \times d_{\text{bala}}$, e uma minoria (50) se deve à *feature* d_{inimigo} . Isso significa que, para que a *feature* d_{inimigo} tenha *qualquer influência razoável* no valor de preferência da ação, os pesos que a multiplicam teriam que ser *extremamente* mais altos. Para que ambas as *features* contribuíssem igualmente para o cálculo de $Q_{\theta}(s_t, F)$, de fato, teríamos que ter algo como $\theta_1 = 1000$ e $\theta_2 = 1$.

O fato de que θ_1 precisa assumir valores *muito* maiores do que θ_2 é um problema para os algoritmos de busca local, pois significa que, ao se gerar os vizinhos de uma determinada solução $[\theta_1, \theta_2]$, teríamos que somar/subtrair números muito maiores no primeiro parâmetro, e muito menores no segundo. A perturbação em θ_1 teria que consistir, por exemplo, na soma ou subtração de um número aleatório no intervalo de $[-100, +100]$, enquanto que a perturbação em θ_2 teria que consistir, por exemplo, na soma ou subtração de um número aleatório no intervalo $[-0.1, +0.1]$.

Para resolver esse problema, fazemos a **normalização de features**. Este processo força com que todas as *features* assumam valores em um mesmo intervalo, conhecido. De forma geral, se temos uma *feature* com valores na faixa $[\min, \max]$, podemos normalizá-la para a faixa entre 0 e 1 da seguinte maneira:

$$featureNormalizada = \frac{featureOriginal + \min}{\max - \min}$$

Podemos também normalizá-la para faixa entre -1 e +1 assim:

$$featureNormalizada = 2 \times \frac{featureOriginal + min}{max - min} - 1$$

Ao se fazer isso, a geração dos vizinhos fica mais fácil: como sabemos que todas *features* (agora normalizadas) estão nessa faixa relativamente pequena de números, podemos gerar números aleatórios, por exemplo, sempre no intervalo $[-0.1, +0.1]$, e teremos certeza que todas as *features* vão ter alguma influência real no valor final da função de preferência. Caso contrário, *features* com valores em faixas muito maiores acabariam influenciando o valor de preferência muito mais; e isso, por sua vez, significaria que (na prática) *features* com faixa de valores muito pequena poderiam acabar sendo ignoradas ao se determinar a qualidade de uma ação.

6 Especificação Técnica do Simulador – Instruções

Essa seção apresenta instruções de como instalar o ambiente de desenvolvimento e o simulador necessários para desenvolver o trabalho. Você precisará: 1) instalar a linguagem Python; 2) obter uma cópia do código fonte básico do Gladiator Arena, o qual você irá estender com suas funções de aprendizado; e 3) desenvolver as funções que implementam o seu método de aprendizado para controle do Gladiator.

6.1 Como instalar Python no seu sistema

Nota: O simulador Gladiator Arena **não** funciona na versão 3.5 do Python, nem no Windows nem no Linux. As instruções a seguir descrevem como instalar a versão correta do Python.

6.1.1 Instalando Python no Linux

Para rodar código Python em um sistema Linux, existem dois tipos de instalação: a virtual e a global. Cada uma delas requer passos diferentes, e cada uma destas alternativas oferece graus de liberdade e segurança diferentes, conforme discutido abaixo:

Instalação global: com passos mais simples, a instalação global é aquela que fica disponível em todo o sistema; ou seja, diferentes projetos que estejam em diferentes partes do seu sistema de arquivos têm acesso à mesma versão do Python, e aos mesmos pacotes instalados. Os passos de instalação, nesse caso, são mais simples, e podem ser uma opção para aqueles que não trabalham extensivamente com a linguagem. Em contrapartida, esse tipo de instalação não é recomendável para aqueles que trabalham com múltiplos projetos ao mesmo tempo, tendo em vista que os projetos podem requerer versões diferente do Python, e os pacotes necessários a um projeto podem gerar conflitos com os pacotes necessários a outros projetos.

Passos para instalação: versões mais recentes de Debian e Ubuntu já vêm com o Python instalado nativamente. Para verificar se a linguagem consta no sistema, rode o seguinte comando no terminal:

```
$ python --version
```

Se a linguagem Python já estiver instalada, o comando retornará sua versão. Caso contrário, é possível instalá-la com os seguintes comandos:

```
$ sudo apt-get install Python2.7 (distribuicoes Debian/Ubuntu)
$ sudo yum install python (distribuicoes Red Hat/RHEL/CentOS)
```

Se a instalação tiver sucesso, o comando que verifica a versão da linguagem retornará algo como:

```
$ python --version
$ python 2.7.6
```

Instalação virtual: na instalação virtual se utiliza um programa chamado virtualenv para criar, na pasta do projeto, uma pasta contendo os binários

de uma versão específica do Python (a ser especificada pelo usuário). Essa instalação só ficará disponível para uso no projeto da disciplina. A instalação do `virtualenv` também é possível via `apt-get`/`yum`:

```
$ sudo apt-get install python-virtualenv
$ sudo yum install python-virtualenv
```

Após instalar o *virtualenv*, a criação de um ambiente virtual Python é feita através do seguinte comando:

```
path/to/project: $ virtualenv <nome do ambiente>
```

onde "path/to/project" descreve a pasta na qual o seu projeto está sendo desenvolvido. Por exemplo, o comando:

```
$ virtualenv env
```

cria uma pasta chamada "env" na pasta local onde o projeto está localizado. Para ativar o ambiente (e, assim, dizer para o sistema onde pacotes e bibliotecas Python devem ser instalados), basta executar o comando:

```
$ source env/bin/activate
```

Após isso, todos pacotes e bibliotecas instalados serão colocados na pasta "env" e estarão disponíveis para uso cada vez que o ambiente criado anteriormente for ativado através do comando `activate`. Note que o comando de ativação deve ser repetido cada vez que um terminal for aberto.

6.1.2 Instalando Python no Windows

Instalação global: Baixe a versão **2.7** do Python no *site*

<https://www.python.org/downloads/>

e adicione o local de instalação (por exemplo, `C:\Python27`) à variável `PATH`. Para fazer isso, vá em **[[Painel de Controle > Sistema > Alterar as variáveis de ambiente > Variáveis de ambiente]]** e troque o valor da variável `PATH` para:

```
valor_antigo_de_PATH;C:\Python27;C:\Python27\Scripts
```

O exemplo acima assume que os arquivos referentes à linguagem foram instalados em `C:\Python27`.

6.2 Instalando os pacotes necessários para o simulador

Após instalar o Python, precisamos instalar alguns pacotes e bibliotecas necessárias para a execução do simulador do Gladiator Arena. A ferramenta de instalação de pacotes Python é chamada *pip*. O *pip* já vem incluso na instalação do Python desde a versão 2.7. Caso ele não esteja instalado, entretanto, rode os seguintes comandos (no Linux):

```
$ sudo apt-get install python-pip
$ sudo yum install python-pip
```

O simulador do Gladiator necessita de quatro pacotes/bibliotecas Python:

- **pyglet** (responsável pela interface do jogo);

- **pymunk** (responsável pela simulação da física);
- **pygame** (responsável pelo jogo);
- **numpy** (para operações matemáticas).

Todos estes pacotes podem ser instalados através do comando:

```
pip install <nome_do_pacote>.
```

6.3 Copiando o repositório do Gladiator Arena

Antes de mais nada, crie uma conta no site do **Bitbucket** (bitbucket.org), caso ainda não possua uma (por que o bitbucket ao invés do github? porque os repositórios privados são de graça :-)).

O código base que todos os alunos utilizarão para desenvolver o trabalho está disponível em um repositório nos servidores do Bitbucket. Um repositório nada mais é do que um local no qual um código fonte encontra-se salvo, e que armazena o histórico completo de todas modificações feitas ao código. Como o histórico de todas as modificações feitas ao código é salvo pelo sistema, pode-se ter um controle maior do estado atual do projeto (comparado com versões anteriores), assim como controle de quais mudanças e atualizações ao código foram feitas por qual desenvolvedor. O fato de que o repositório guarda o histórico completo de desenvolvimento também permite que se analise versões antigas do programa, e que se desfça eventuais modificações que introduziram *bugs*. Este tipo de sistema de gerência de código fonte se chama controle de versões. O Bitbucket é um sistema online que armazena repositórios com estas funcionalidades. O repositório contendo o código básico do simulador, e que será estendido pelas trios através do desenvolvimento de métodos de aprendizado, se encontra no Bitbucket.

Para que cada grupo possa ter sua própria **cópia privada** do código do simulador Gladiator Arena disponibilizado no Bitbucket (e que irá ser modificado durante a implementação do método de aprendizado do trio) é preciso que cada grupo copie, no Bitbucket, o repositório principal do simulador através da opção *fork*. A opção de fork está visível no canto superior da tela da página do repositório no site Bitbucket:

<https://bitbucket.org/marcosps96/gladiators-ia>

No momento desta cópia, cada trio poderá dar um nome a sua própria cópia do projeto, e poderá também modificar as suas permissões de acesso. Ao executarem o *fork*, **marquem a checkbox que torna o repositório privado—isso é importante para evitar que outros grupos copiem a sua solução!**

Após feito o *fork*, cada trio terá, em sua conta no Bitbucket, uma cópia particular do repositório contendo o código base do simulador. Depois de criada esta cópia, o grupo deve adicionar os usuários *gmmoita*, *marcosps96* e *Achren* (os monitores) e *bsilvapo* (o professor) para que tenham acesso ao seu repositório (através da opção "invite users to this repo"). Isso permitirá com que os monitores e o professor da disciplina analisem o progresso sendo feito pelo trio durante

o desenvolvimento do trabalho.

Após a criação da cópia do projeto no site do Bitbucket, é necessário baixar do Bitbucket o código-fonte para a máquina local na qual o trabalho será desenvolvido. Isso vai permitir com que um aluno altere o código-fonte localmente, no seu computador, como de costume durante o desenvolvimento de um programa. Permitirá também com que o código desenvolvido localmente por um dos alunos possa ser enviado de volta para o servidor Bitbucket, a fim de que possa ser acessado pelos outros alunos do trio — os quais podem estar desenvolvendo *outras* partes do software em seu próprio computador. Esse tipo de sincronização das contribuições feitas por cada aluno através do site Bitbucket evita com que (p.ex.) o aluno 1 tenha que enviar um email para o aluno 2 contendo o novo código fonte, forçando o aluno 2 a manualmente identificar quais partes do código foram alteradas pelo aluno 1, e copiá-las para sua própria versão do código.

Para gerenciar o seu repositório do Bitbucket, utilizaremos o programa **GIT**. O GIT é usado 1) para baixar o código fonte do site Bitbucket para uma máquina local, na qual o aluno irá desenvolver seu código; 2) para salvar alterações feitas ao código no histórico de desenvolvimento do projeto; e 3) para enviar a versão do código sendo desenvolvida por um aluno, em seu computador, de volta ao servidor Bitbucket, a fim de que possa ser acessada pelo outro aluno, professor e monitores.

Para instalar o GIT, siga as instruções em:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Os principais comandos GIT que vocês precisarão utilizar são os seguintes:

`$ git clone <endereço do repositório>` - faz uma cópia do código no repositório Bitbucket para a máquina de desenvolvimento. Esse passo é executado apenas uma vez, ao se baixar o código do Bitbucket pela primeira vez para um computador. Como o repositório de cada trio (criado e nomeado através do processo de fork, descrito anteriormente) é privado, o usuário precisará se identificar para ter acesso através do clone. O endereço do repositório criado pelo trio durante o fork possui a seguinte forma:

`https://<user_bitbucket>@bitbucket.org/user_dono/nome_repo.git`

Alternativamente, vocês podem descobrir o endereço de seu repositório entrando no site do Bitbucket, acessando a sua cópia do repositório, e copiando o seu endereço no canto superior direito da dela; por exemplo:



Figure 5: Dentro do campo estará o endereço SSH.

Depois de clonar o repositório, vocês terão uma cópia completa do código fonte em sua máquina local, a qual poderão modificar normalmente. Sempre que forem adicionados novos arquivos Python ao seu código, você precisará adicioná-lo ao projeto. Isso é feito através do comando *git add*:

```
$ git add <arquivos_a_serem_adicionados_ao_projeto>
```

Depois de se adicionar um ou mais arquivos ao projeto, será possível usar comandos GIT para salvar versões específicas daquele arquivo (ou do projeto completo) no histórico de desenvolvimento do repositório. Isso permitirá, conforme mencionado anteriormente, que seja feito o controle das diferentes versões do programa, conforme ele vai sendo desenvolvido. Para salvar a versão atual do código fonte no repositório local (criado através do comando *git clone*) utilize o comando *commit*:

```
$ git commit -m "mensagem"
```

Esse comando salva as mudanças feitas ao código fonte no repositório local do projeto. A mensagem serve para explicar quais modificações foram feitas ao código; por exemplo, quais novas funcionalidades foram incluídas no projeto, ou quais bugs foram corrigidos.

```
$ git status
```

Esse comando compara o código fonte atual na pasta do projeto com a última versão gravada no repositório do projeto. Isto é, caso você altere arquivos depois de tê-los gravado no repositório (através do comando *commit*), o comando status irá informar exatamente quais arquivos foram modificados.

Todos os comandos acima atualizam apenas o repositório local do projeto, em um computador específico no qual o programa está sendo desenvolvido — por exemplo, no computador de um dos alunos do trio. A fim de enviar essas modificações de volta para o servidor Bitbucket, de modo que os outros membros do trio possa, baixá-las nos seus próprios computadores (através do comando *pull*, descrito mais adiante), e também para que os monitores e professor possam acompanhar o progresso do desenvolvimento, é necessário utilizar o comando *git push*:

```
$ git push origin master
```

Esse comando envia a versão atual do código (gravada no repositório local do projeto através de comandos *commit*) ao servidor do Bitbucket. Depois de feito um *push*, o histórico de todos os *commits* feitos poderá ser visualizado na página do repositório, na área de *commits*.

```
$ git pull
```

Suponha que o aluno 1 do grupo esteja desenvolvendo parte do programa em seu computador, e que tenha enviado sua nova versão do código para o servidor Bitbucket através do comando *push*. O aluno 2 poderá então baixar estas alterações e automaticamente integrá-las a sua versão do código, gravada em sua máquina local. O comando *pull* baixa do Bitbucket o código enviado pelo aluno 1, compara-o com o código local do aluno 2, verifica quais são as novidades, e integra todas modificações feitas pelo aluno 1 no código do aluno 2, de forma que ambos os códigos fiquem sincronizados e que aluno 2 possa ter acesso as novas contribuições do aluno 1.

6.3.1 Exemplo de uso

Um exemplo de uso do GIT seria o seguinte: imaginem que cada aluno da trio baixou o código fonte do Bitbucket utilizando o comando *git clone*. Cada aluno, portanto, tem agora uma cópia do projeto em sua própria máquina local. Imagine que o aluno 1 fez mudanças na função **compute_features()** (no arquivo *AI/Controller.py*), e que tais mudanças são necessárias para que o aluno 2 possa prosseguir com sua parte do trabalho. O primeiro aluno executaria os seguintes comandos:

```
$ git add AI/Controller.py
```

```
$ git commit -m "mudancas na funcao compute_feature"
```

```
$ git push origin master
```

Após a execução destes comandos, todas as modificações feitas ao arquivo *AI/Controller.py* pelo aluno 1 seriam gravadas no repositório local (comando *commit*) e depois enviadas ao site do Bitbucket (comando *push*). O aluno 2, para ter acesso a estas modificações em sua própria máquina, executaria então (em seu próprio computador) o comando:

```
$ git pull
```

Este comando iria baixar todas as alterações feitas pelo aluno 1 (e enviadas ao Bitbucket) e integrá-las ao código local do aluno 2, sincronizando o código de ambos alunos.

O método de desenvolvimento descrito acima pode parecer complicado a primeira vista, mas sistemas de controle de versão deste tipo (tal como o GIT) são amplamente utilizados, tanto em empresas quanto academicamente, a fim de permitir o controle do estado do *software* em diferentes momentos de seu desenvolvimento, e também para facilitar o compartilhamento de funções desenvolvidas por um programador com todo o resto do time de desenvolvimento. Para mais informações, um bom ponto de partida para estudar o sistema GIT é o guia já citado anteriormente, e também o seguinte manual:

<https://www.atlassian.com/git/tutorials/what-is-version-control/benefits-of-version-control>

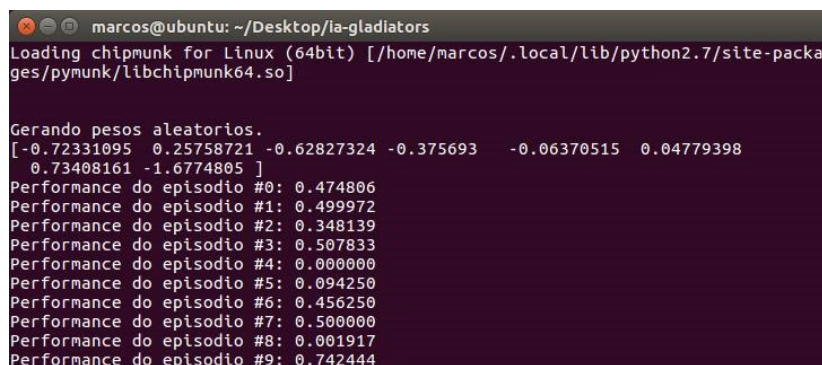
7 Como Utilizar o Gladiator Arena

Há 2 modos diferentes de utilizar o Gladiator Arena: o modo **learn** e o modo **evaluate**.

O modo **learn** é usado para executar o método de aprendizado implementado pelo trio, a fim de que se aprenda um controlador eficaz para o Gladiador. Esse modo não aciona a visualização gráfica do Gladiador sendo controlado, a fim de tornar a execução do aprendizado mais rápida. No modo **learn**, o Gladiador é inicializado em uma configuração aleatória (uma posição e orientação aleatória dentro da arena), e então utiliza-se o controlador desenvolvido pelo trio para batalhar contra um bot escolhido—mais detalhes sobre isso a seguir. A performance do controlador é então medida: mede-se por quanto tempo ele conseguiu sobreviver, caso tenha perdido, ou em quanto tempo conseguiu derrotar o oponente, etc. Tal informação é passada ao método de aprendizado (algoritmo de busca local) implementado pelo trio na função **Controller.update**. Tal função poderá, então, alterar os parâmetros $\theta_1 \dots \theta_N$ do controlador a fim de atualizá-lo e melhorar a sua performance. A função **Controller.compute_features**, que também deve ser desenvolvida pelo grupo, recebe o estado s_t do agente (o qual inclui o valor atual de todos os seus sensores) e deverá retornar um conjunto de *features* a ser determinado pelo grupo. Por fim, a função **Controller.take_action**, também sob responsabilidade do grupo, deverá por calcular os valores de preferência para cada uma das ações, utilizando para isso os parâmetros atuais θ do controlador e as *features* calculadas com base no estado s_t ; com base nos valores de preferência, deve retornar a ação a ser executada. Os parâmetros iniciais a serem utilizados pelo controlador, no início do processo de aprendizado, podem ser informados através de um arquivo texto informado durante a execução do simulador. Caso nenhum arquivo seja informado, pesos iniciais serão gerados automaticamente de forma aleatória. Exemplo de uso:

```
$ python __init__.py learn ninja_bot pesosiniciais.txt
```

O código acima produzirá uma saída parecida com essa:



```
marcos@ubuntu: ~/Desktop/la-gladiators
Loading chipmunk for Linux (64bit) [/home/marcos/.local/lib/python2.7/site-packages/pymunk/libchipmunk64.so]

Gerando pesos aleatorios.
[-0.72331095  0.25758721 -0.62827324 -0.375693   -0.06370515  0.04779398
  0.73408161 -1.6774805 ]
Performance do episodio #0: 0.474806
Performance do episodio #1: 0.499972
Performance do episodio #2: 0.348139
Performance do episodio #3: 0.507833
Performance do episodio #4: 0.000000
Performance do episodio #5: 0.094250
Performance do episodio #6: 0.456250
Performance do episodio #7: 0.500000
Performance do episodio #8: 0.001917
Performance do episodio #9: 0.742444
```

Figure 6: Saída produzida pelo modo *learn*.

O processo de aprendizado utiliza o conceito de um episódio de treinamento: em um episódio, o Gladiador é inicializado em uma configuração aleatória, e

o controlador atual (com parâmetros $\theta_1 \dots \theta_N$) é utilizado para controlá-lo até que ele derrote o inimigo, seja derrotado, ou que um tempo-limite interrompa o episódio. Quando isso ocorrer, sua performance (tempo até a vitória/derrota ou empate) é calculada e informada para a função de aprendizado, a qual poderá atualizar os parâmetros do controlador. Neste momento, diz-se que o episódio de treinamento acabou. A seguir, um novo episódio de treinamento é iniciado, e o processo se repete. O modo *learn* executa episódios de treinamento indefinidamente, até que o usuário interrompa o processo de aprendizado com Ctrl-C. A cada dez episódios, os parâmetros atuais aprendizados do controlador são gravados na pasta “params/”.

O modo **evaluate**, por sua vez, permite que o aluno teste e visualize o que acontece quando o Gladiador é controlado através de um determinado conjunto de parâmetros, descobertos durante o processo de aprendizado. Nesse modo, o programa roda a interface gráfica do simulador, mostrando o Gladiador batalhando contra um *bot* escolhido. Os *bots* disponíveis estão listados no arquivo **Simulator/bots.py**. Como esse modo é utilizado para verificar o comportamento do Gladiador resultante de um conjunto de parâmetros, um arquivo texto com os parâmetros que se deseja testar deve ser informado. Exemplo de uso:

```
$ python __init__.py evaluate ninja_bot pesos.txt
```

O programa, nesse caso, vai alimentar o controlador do Gladiador com os parâmetros informados e rodar/exibir o simulador, conforme o exemplo da Figura 7:

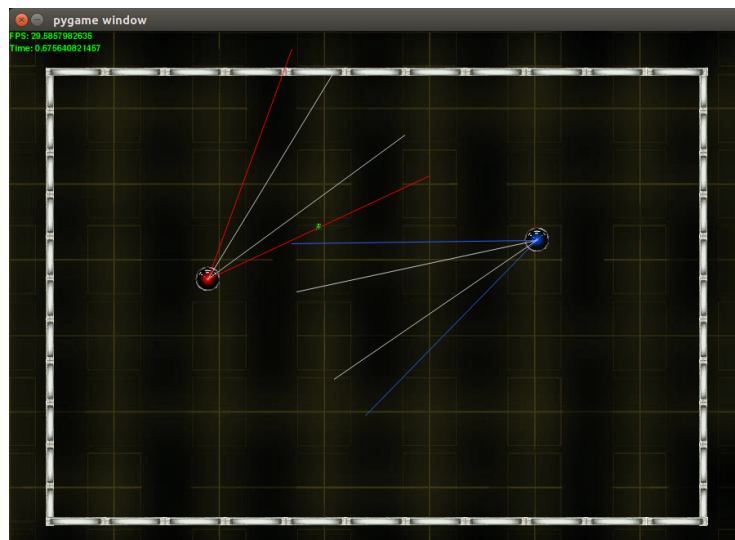


Figure 7: O simulador *Gladiator Arena* em modo *evaluate*.

Note que no modo **evaluate**, os parâmetros do controlador não são atualizados ao final de cada episódio. Este modo apenas avalia um controlador aprendido pelo modo **learn**.

8 Entrega e Apresentação de Resultados

Cado trio deverá fazer *commit* de seu código, *de forma regular*, para o repositório Bitbucket compartilhado com o professor e com os monitores (vide o item 6 da especificação técnica para mais detalhes). Isso permitirá com que o progresso sendo feito pelo trio seja avaliado durante o desenvolvimento do trabalho. Ao final, o trio deverá entregar a última versão do código fonte utilizado.

Cado trio deverá preparar um relatório técnico descrevendo o projeto desenvolvido. Descreva quais algoritmos de busca foram implementados e avaliados. Cado trio **deverá** implementar¹ e avaliar **ao menos três algoritmos**. Discuta as razões que os levaram a tentar tais algoritmos, se eles funcionaram, o quão eficientes foram na descoberta de controladores eficazes para o robô, quais as dificuldades foram encontradas em sua implementação, e desvantagens do uso daquele tipo de método de busca—p.ex., muito lento, não encontrou soluções boas, etc. Discuta também quais parâmetros do algoritmo de busca você precisou ajustar; tamanho da população, se utilizando um Algoritmo Genético; parâmetro de temperatura, caso utilizando Simulated Annealing; etc. Descreva *como* você descobriu valores razoáveis para tais parâmetros. Fale sobre quais *features* de estado você utilizou, e os porquês. **Não** discuta com outros grupos, durante o desenvolvimento do trabalho quais *features* seu trio está utilizando ou avaliando (novamente, vide Seção 9 para mais informações sobre o que constitui plágio).

Por fim, você deverá apresentar e discutir a performance do seu controlador para cada um dos algoritmos avaliados. Isso deverá ser feito através de gráficos de *curva de aprendizado* para cada algoritmo. Uma curva deste tipo consiste em um gráfico onde o eixo horizontal representa o número do episódio de treinamento, e o eixo vertical mostra qual a performance do controlador naquele episódio². Caso o método de aprendizado do trio estiver conseguindo melhorar de performance do controlador de forma consistente, isso será refletido de forma clara no gráfico, que será uma curva crescente. Prepare este tipo de gráfico para os **diferentes algoritmos** avaliados e para os **diferentes conjuntos de *features*** os quais você experimentou, a fim de demonstrar o impacto do algoritmo e da representação das funções de preferência na performance do processo de busca local. Apresente, por fim, o *tempo* de aprendizado (em iterações), para cada algoritmo e tipo de *features* utilizadas. Discuta a razão de possíveis acelerações no tempo de aprendizado em função destas escolhas.

No dia indicado no cronograma da disciplina, haverá uma competição entre os controladores implementados por cada trio. Neste dia, cada grupo deverá levar, em um arquivo texto, o conjunto de parâmetros ótimos aprendidos por seu algoritmo de busca. Deverá, também, entregar o seu relatório escrito. Ao final do curso, cado trio deverá apresentar oralmente os seus trabalhos e conclusões.

¹i.e., desenvolver totalmente do zero! Vide política de plágio na seção 9.

²Você pode gravar estas informações em um arquivo ou imprimi-las na tela, para posterior uso no Excel ou outro software que produza gráficos.

9 Política de Plágio

Trios poderão **apenas** discutir questões de *alto nível* relativas a resolução do problema em questão. Poderão discutir, por exemplo, quais técnicas de busca local estão considerando utilizar, quais suas vantagens e desvantagens, etc. **Não** é permitido que os trios utilizem quaisquer códigos fonte provido por outros trios, ou encontrados na internet. Os alunos **poderão** fazer consultas na internet ou em livros **apenas** para estudar o modo de funcionamento das técnicas de IA, e para analisar o **pseudo-código** que as implementa. **Não** é permitida a análise ou cópia de implementações concretas (em quaisquer linguagens de programação) da técnica escolhida. **Não** discuta com outros grupos, durante o desenvolvimento do trabalho, quais *features* seu trio está utilizando ou avaliando. O objetivo deste trabalho é justamente implementar a técnica do zero e descobrir as dificuldades envolvidas na sua utilização para resolução de um problema de aprendizado. Toda e qualquer fonte consultada pelo trio (tanto para estudar os métodos a serem utilizadas, quanto para verificar a estruturação da técnica em termos de *pseudo-código*) **precisa obrigatoriamente** ser citada no relatório final. O professor e monitores utilizam rotineiramente um sistema anti-plágio que compara o código-fonte desenvolvido pelos trios com soluções enviadas em edições passadas da disciplina, e também com implementações conhecidas e disponíveis online.

Qualquer nível de plágio (ou seja, utilização de implementações que não tenham sido 100% desenvolvidas pelo trio) poderá resultar em nota zero no trabalho. Caso a cópia tenha sido feita de outro trio da disciplina, *todos* os alunos envolvidos (não apenas os que copiaram) serão penalizados. Esta política de avaliação **não** é aberta a debate posterior. Se você tiver quaisquer dúvidas sobre se uma determinada prática pode ou não, ser considerada plágio, não **assuma** nada: pergunte ao professor e aos monitores.

Note que, considerando-se os pesos das avaliações desta disciplina (especificados e descritos no plano de ensino) percebe-se que nota zero em qualquer um dos trabalhos de implementação **obrigatoriamente** resulta em média inferior a 6.0 nos projetos práticos, o que impede com que o aluno faça prova de recuperação. Ou seja: caso seja detectado plágio, há o risco *direto* de reprovação. Os trios deverão desenvolver o trabalho **sozinhos**.

10 Considerações Finais

- Lembre-se que o simulador do Gladiador **não** funciona com a versão 3.5 do Python. Certifiquem-se de instalar a versão 2.7, conforme descrito nas instruções acima.
- Não hesitem em consultar o professor e/ou os monitores em caso de dúvida em relação ao trabalho.
- **Plágio é terminantemente proibido, conforme descrito no item 9!**
- Divirtam-se :)