Desenvolvimento de um Agente Inteligente utilizando Q-Learning para um Ambiente de Aprendizado

Bruno Cesar Meneguzzu Zamarchi[1], **Árton Pereira Dorneles** [1]

[1]Instituto Federal Farroupilha – Campus Frederico Westphalen (IFFar/FW)
Caixa Postal 169 – 98.400-000 – Frederico Westphalen – RS – Brasil

brunozamarchi@yahoo.com.br, arton.dorneles@iffarroupilha.edu.br

Abstract. *With the recent advance of digital entertainment, there is a great demand in the market for developers with the ability to work with artificial intelligence in the games industry. To facilitate mastery of this area, several learning environments focused on artificial intelligence have been proposed in the literature with different levels of complexity and learning curves. In this paper, we propose the development of an artificial intelligence learning environment in the Python language and with a focus on beginners. The proposed environment offers a simulation of a cops and robbers game with a simple set of rules and an easy-to-learn programming interface. In addition, we also propose an implementation of an intelligent agent to validate the environment, trained with the Q-Learning method. The experimental results demonstrate that this training methodod is effective for creating and testing of intelligent behavior in the proposed simulation environment.*

Resumo. *Com o avanço recente do entretenimento digital, existe uma grande demanda no mercado por desenvolvedores com capacidade para atuar com inteligência artificial na indústria de jogos. Para facilitar o domínio desta área, vários ambientes de aprendizado com foco em inteligência artificial tem sido propostos na literatura com diferentes níveis de complexidade e curvas de aprendizado. Neste trabalho, foi desenvolvido um ambiente de aprendizado de inteligência artificial na linguagem Python e com foco em iniciantes. O ambiente oferece uma simulação de um jogo de polícia e ladrão com um conjunto de regras simples e uma interface de programação fácil de aprender. Além disso, este trabalho também propõe uma implementação de um agente inteligente para validar o ambiente, treinado com o método Q-Learning. Os resultados experimentais demonstram que o uso desta metodologia é eficaz para criar e testar comportamento inteligente no ambiente de simulação proposto.*

1. Introdução

Com o avanço crescente de tecnologias digitais, a inteligência artificial tem sido inserida cada vez mais em sistemas de uso pessoal e industrial, proporcionando soluções distribuídas para uma gama abrangente de aplicações nas áreas de entretenimento, finanças, educação, saúde e apoio na tomada de decisões e gestão de negócios. Dentro da área de entretenimento, uma das áreas que tem atraído bastante interesse de pesquisadores na literatura científica é a aplicação de inteligência artificial para jogos digitais.

Conforme Millington e Funge (2018), um dos maiores desafios enfrentados pela equipe de desenvolvimento de um jogo é a criação de uma inteligência artificial robusta,

ou seja, é necessário projetar comportamentos inteligentes para os agentes do jogo de forma que o jogador tenha a sensação de estar enfrentando ou interagindo com um oponente humano ao invés de uma máquina. Considerando que a produção de jogos digitais é atualmente a maior indústria de entretenimento do mundo, existe uma grande demanda no mercado por programadores com capacidade de atuar nessa indústria, em especial, com experiência em inteligência artificial.

Visto que a implementação de uma inteligência artificial é um processo relativamente complexo quando comparado a outras áreas da computação, a curva de aprendizado desta habilidade também é mais longa, o que pode desestimular o interesse de novos programadores para área. Uma das formas de minimizar esse problema reduzindo a barreira de entrada para o aprendizado de temas complexos é a utilização de ambientes de aprendizagem de simulação. Conforme Stange et al. (2017), um ambiente de aprendizagem de simulação pode ser descrito como um local artificial criado para realizar experimentos em condições controladas, como por exemplo, no ambiente entitulado "Mundo de Wumpus", que simula um jogo de labirinto em que o agente inteligente pode se mover em quatro direções, atirar ou cair em buracos enquanto tenta encontrar um tesouro escondido em uma sala. Em ambientes como este é possível projetar e testar um agente inteligente de forma mais lúdica e precisa, atraindo a atenção, especialmente, de desenvolvedores iniciantes.

Embora existam várias propostas de ambientes de aprendizado voltadas para o desenvolvimento de inteligência artificial em jogos, estas ainda podem apresentar barreiras para um iniciante em função da quantidade de regras do jogo do ambiente, estilo de jogo, necessidade de uso de linguagem de programação específica ou dependência de bibliotecas externas. Nesse contexto, foi desenvolvido um ambiente de aprendizado de inteligência artificial utilizando a linguagem Python para simular um jogo de polícia e ladrão com um conjunto de regras simples. Além de permitir que os desenvolvedores implementem a inteligência dos agentes do jogo por meio de uma interface simplificada, permite que sejam realizados testes para comparação de desempenho entre os agentes. Como uma forma de facilitar a comparação da eficiência dos agentes e também como exemplo de utilização do ambiente, neste trabalho também é proposta uma implementação de inteligência artificial para um agente no papel de policial com uso da técnica de aprendizado não-supervisionado conhecida como Q-Learning.

O restante deste trabalho está organizado como segue. A Seção 2 define os principais conceitos utilizados no trabalho, bem como trabalhos relacionados ao desenvolvimento de ambientes de aprendizagem para inteligência artificial. A Seção 3 define formalmente o ambiente de aprendizado detalhando as regras do jogo, além dos parâmetros e entidades envolvidas. A Seção 4 detalha a implementação do ambiente proposto e o desenvolvimento do agente inteligente do policial utilizando *Q-Learning*. A Seção 5 apresenta os experimentos computacionais e comparações de parâmetros realizados no ambiente proposto. Por fim, a Seção 6 apresenta as considerações finais do trabalho e sugestões de trabalhos futuros.

2. Referencial Teórico

Nesta seção são apresentados os conceitos teóricos e tecnológicos necessários para a compreensão deste trabalho, bem como os principais trabalhos relacionados.

2.1. Fundamentos de Inteligência Artificial e Aprendizado de Máquina

A Inteligência Artificial (IA) pode ser definida como um sistema que age por conta própria para realização de uma tarefa ou comportamento inteligente. Sistemas desse tipo estão inseridos cada vez mais mais no nosso dia a dia, como por exemplo, no corretor ortográfico de smartphones ou em implementações de anúncios digitais que são treinados para exibir recomendações de compra com base em interesses. Estas aplicações são exemplo de sistemas de aprendizado de máquina, uma área promissora da inteligência artificial.

Conforme Russel e Norvig (1996), aprendizado de máquina é a arte de criação de máquinas que sejam capazes de realizar ações que humanos teriam de pensar para realizar. É um processo que permite a máquina tomar decisões de forma semelhante a de um agente humano, ou seja, agindo de maneira autônoma em um determinado ambiente como um agente inteligente.

Agentes inteligentes são entidades implementadas via sistema e que tentam determinar de forma autônoma a melhor ação a ser realizada para resolver um problema específico. Suponha que um agente humano precise realizar a tarefa de buscar um site na internet. Se fizesse isso por conta própria, poderia levar um longo tempo testando endereços no navegador. Já com a ajuda de um agente inteligente não seria necessário nem mesmo conhecer o endereço do site, bastaria informar o agente o que precisa e ele se encarrega de fazer uma busca inteligente capaz de encontrar vários endereços que resolvem o problema de busca do usuário. Um agente inteligente passa por um processo de treinamento ou aprendizado conduzido por um desenvolvedor, de forma que aprendam a lidar com diversas situações diferentes por conta própria (Rudowsky, 2004). Estes agentes são categorizados de acordo com seu funcionamento, agentes reativos desenvolvem seu conhecimento a partir de interações com seu ambiente, agentes cognitivos são mais complexos, com mecanismos de tomada de decisões avançados, interações sofisticadas e com um objetivo fortemente estabelecido, agentes baseados em objetivos procuram atingir os objetivos presente no ambiente, e agentes baseados na utilidade tentam maximizar suas expectativas procurando realizar os objetivos de maior valor. Foram implementados agentes reativos que agem conforme o estado do ambiente em que se encontram. Entre os modelos de aprendizagem, podemos destacar os modelos supervisionados e não-supervisionados.

De acordo com Cord e Cunningham (2008) a aprendizagem supervisionada parte do princípio de treinar a inteligência artificial a partir de informações de um conjunto de dados de entrada e saída. Neste modelo de aprendizagem a IA é treinada em um ambiente em que cada entrada já possui o resultado esperado, assim a IA aprenderá a reconhecer padrões nos dados de entrada. Por exemplo, se desejamos ensinar uma IA a reconhecer carros é necessário prover um conjunto de dados dizendo que na imagem existe um carro. Por outro lado, se quisermos que a IA seja capaz de reconhecer a marca do carro temos que prover imagens já rotuladas com as marcas. Após a IA estudar várias imagens, ela será capaz de reproduzir com determinada precisão os rótulos em imagens novas.

Em contrapartida, no processo de aprendizagem não-supervisionada o treinamento da IA acontece partindo apenas de informações de entrada com o objetivo de aprender uma resposta adequada a partir da interação com os dados. Por exemplo, supondo que uma IA receba um conjunto de entrada com imagens de várias frutas, a IA não saberá dizer se na imagem há uma maçã ou uma banana, mas ela será capaz de agrupar as frutas

por suas características como cor, tamanho e formato. Uma das principais técnicas para aprendizagem não-supervisionada é a Aprendizagem por reforço.

Conforme Russel e Norvig (1996), na Aprendizagem por Reforço, existe um conjunto pré-determinado de ações que a IA pode executar de acordo com as informações do ambiente em que está inserida. Em um jogo, por exemplo, um agente pode estar ciente de informações ao seu redor, como a proximidade de obstáculos ou inimigos que podem impactar na realização de um objetivo. Dependendo da ação que o agente tomar em relação a esses estímulos, os resultados podem ser negativos ou positivos. Para proceder com o aprendizado, é necessário avaliar diferentes parametrizações para a execução de ações do agente, quando algo positivo acontece o agente é recompensado, mas quando o resultado é negativo, há uma punição associada. Desta forma, ao final de cada geração de avaliação é escolhido um agente com parâmetros que obtiveram mais recompensas. O processo é repetido de forma iterativa até que uma condição de aprendizado seja atingida.

2.2. *Q-Learning*

Conforme Russel e Norvig (1996), *Q-Learning* é um modelo de aprendizagem por reforço de uso geral que também pode ser classificado como um método de programação dinâmica assíncrona. Com esta técnica é possível treinar agentes inteligentes para agirem de forma efetiva com base nas consequências das ações realizadas.

Para aplicar o *Q-Learning* pressupõe-se a existência de um conjunto de estados E no ambiente de aplicação, um conjunto de ações A que podem ser realizadas por um agente inteligente e valores de recompensa R_e para $e \in E$. Sempre que o agente encontra um estado $e \in E$ ele precisa escolher uma ação $a \in A$ que o leva para outro estado sujeito a uma recompensa possivelmente diferente. Nesse contexto, o método *Q-Learning* tem como objetivo criar um mapeamento para cada par (a, e) com um valor numérico que represente a qualidade de realizar a ação a no estado e. Esse mapeamento fica armazenado como uma memória em uma tabela Q_{ea} chamada *Q-table*. Cada valor da tabela é chamado *Q-value*. Desta forma, após serem testadas múltiplas ações em diferentes estados por um dado número de gerações G, o agente se torna capaz de agir de forma inteligente no ambiente em que foi treinado apenas inferindo a melhor ação a partir dos *Q-values*.

Conforme Dearden et al. (1998), um agente deve começar em um estado aleatório do ambiente no início de cada geração e executar ações para percorrer uma quantidade significativa de estados. Para que isso ocorra de forma efetiva, além de ser necessário definir um número adequado de gerações G, as ações devem ser realizadas conforme uma política de exploração que evolui conforme o estágio do treinamento. Nas gerações iniciais, quando o agente possui pouco conhecimento, este deve optar por realizar mais ações aleatórias para explorar o ambiente. Em contrapartida, nas gerações finais, o agente deve adotar uma abordagem mais gulosa que priorize a realização de boas ações.

Outro ponto importante no processo de treino utilizando *Q-Learning* e a atualização da *Q-table*. Sempre que o agente realiza uma ação $a \in A$ seguindo a política de exploração, o ambiente mudará de um estado $e \in E$ para um novo estado $e' \in E$. Com isso, o valor correspondente a Q_{ae} deve ser calculado conforme a Equação (1). Nesta equação, Q_{ea} é atualizado com o valor associado ao *Q-value* do estado anterior($Q_{e'a}$) acrescido de um percentual α de Δ. O parâmetro α é a taxa de aprendizagem que varia entre 0 e 1, e Δ é definido pelo método *Temporal Difference* criado por Sutton e Barto (1987) conforme

a Equação (2). O valor de Δ é composto da recompensa imediata associada ao novo estado ($R_{e'}$) acrescida de um percentual λ da diferença entre o maior *Q-value* entre todas as ações possíveis para o novo estado e o *Q-value* do estado anterior. O parâmetro λ que varia de 0 a 1 é conhecido como fator de desconto.

$$Q_{ea} = Q_{e'a} + \alpha\Delta \tag{1}$$

$$\Delta = R_{e'} + \lambda \max_{i \in A} Q_{e'i} - Q_{ea} \tag{2}$$

Na Figura 1 é apresentado um pseudocódigo para detalhar o processo de treinamento utilizando *Q-Learning*. A função `QLearningTraining` recebe quatro parâmetros: a taxa de aleatoriedade final da movimentação (ε_f), a taxa de aprendizagem (α), o fator de desconto (λ) e o número total de gerações utilizado no treinamento (G). O algoritmo inicia inicializando a *Q-Table*, as recompensas conforme as configurações da aplicação específica e o ε que gerencia a escolha das ações. A seguir, no laço externo das linhas 4-19 são executadas várias gerações do treinamento. cada uma se inicia na linha 4 com a escolha de um estado aleatório para o ambiente da simulação.

Algoritmo `QLearningTraining` ($\varepsilon_f, \alpha, \lambda, G$)

1: Inicializa Q com zeros.
2: Inicializa R com recompensas/penalidades conforme a aplicação.
3: $\varepsilon = 0$
4: **for** $g \leftarrow 1$ **to** G **do**
5: Seleciona um estado inicial $e \in E$ aleatoriamente.
6: $\varepsilon = \varepsilon_f(g/G)$
7: **repeat**
8: Escolhe um x aleatório entre 0 e 1.
9: **if** $x > \varepsilon$ **then**
10: Escolhe uma ação $a \in A$ aleatoriamente.
11: **else**
12: Escolhe uma ação $a \in A$ onde $a = \arg\max_{a \in A} Q_{ea}$.
13: **end if**
14: Executa a ação a originando um novo estado e'.
15: $\Delta \leftarrow R_{e'} + \lambda \max_{i \in A} Q_{e'i} - Q_{ea}$.
16: $Q_{ea} \leftarrow Q_{e'a} + \alpha\Delta$.
17: $e \leftarrow e'$.
18: **until** e seja um estado terminal
19: **end for**
20: **return** Q

Figura 1. Pseudocódigo do algoritmo de Aprendizado utilizando Q-Learning

No laço mais interno, nas linhas 7-18, são realizadas várias ações até que um estado terminal seja encontrado. Um estado terminal é aquele encontrado em uma situação que a simulação se encerra, como por exemplo, quando o agente atinge um determinado objetivo ou quando é eliminado. A escolha da ação realizada pelo agente durante o treino é realizada nas linhas 6-13, podendo ser uma ação completamente aleatória para explorar

o ambiente (linha 10) ou se utilizar do conhecimento adquirido durante o treino, escolhendo a melhor ação com base nas informações armazenadas até o momento na *Q-table* (linha 12). Essa decisão é controlada pelo parâmetro ε que começa em zero (linha 3) e é atualizado conforme a política definida na linha 6. Essa política garante que o ε aumente gradualmente com as gerações, podendo chegar no seu valor máximo até ε_f. Uma vez que a ação é escolhida usando esse procedimento, o movimento é executado na linha 14, seguido da atualização da *Q-table* nas linhas 15-16, bem como a atualização do estado atual do ambiente na linha 17. Ao final do treinamento, na linha 20, a *Q-table* é retornada com o conhecimento adquirido e pode ser utilizada para realizar inferências.

2.3. Python

Python é uma linguagem de programação orientada a objetos, multiplataforma e interpretada, bastante utilizada em análise de dados, estatística, programação web e inteligência artificial. Além de ser uma linguagem de fácil aprendizado, ela oferece um grande conjunto de bibliotecas com funções pré-definidas, as quais permitem uma construção ágil de projetos. São exemplos de programas modernos implementados em Python: Blender 3D, Pinterest, Instagram e Spotify (Borges, 2014).

2.4. Trabalhos relacionados

Nesta seção são apresentados três ambientes de simulação propostos na literatura que tem objetivo semelhante ao deste trabalho.

2.4.1. RoboCode

RoboCode é um jogo de programação, que possui como meta criar uma inteligência artificial para controlar um tanque de forma independente em um ambiente de simulação. Cada partida do jogo gerencia um confronto entre diferentes técnicas de inteligência artificial sem interferência direta dos jogadores, permitindo avaliar o desempenho de cada uma. O RoboCode fornece um ambiente de aprendizagem onde as pessoas podem pôr em prática diferentes técnicas de criação de IAs, permitindo que estudantes avaliem as vantagens e desvantagens em seus códigos (Hartness, 2004).

2.4.2. RoboCup

Conforme Chen et al. (2001), a RoboCup é uma competição mundial que ocorre anualmente e tem como principal objetivo a criação de uma equipe de robôs humanoides com capacidade para agir de forma autônoma, que seja capaz de derrotar a equipe campeã mundial de futebol. Com isso vem a necessidade de um ambiente de simulação, onde possam testar por conta própria os sistemas criados, e para isto existe o Soccerserver, um ambiente multi-agente de tempo real, onde para conseguir a vitória um time deve fazer o máximo de gols enquanto evita levar gols. Assim sendo necessário que os membros de cada equipe precisem agir de forma rápida e flexível enquanto cooperam tendo em consideração os acontecimentos locais e globais. Devido a alta complexidade necessária para criação de IAs neste, ambiente seu uso é mais direcionado para usuários avançados.

2.4.3. Bomberman X

O projeto Bomberman X, proposto por de Souza (2016), tem como objetivo ajudar no ensino da criação de IAs entregando um ambiente que incentive os estudantes a ter maior interesse em aula e possam aprender de maneira mais efetiva o conteúdo estudado. Os autores implementaram um ambiente de simulação baseado na franquia de jogos Super Bomberman de 1993 a 1997, usando a linguagem de programação Java. Além de simular quase todas as funcionalidades do jogo original, o ambiente de simulação permite que programadores implementem a IA de seu personagem ao em vez de jogar diretamente.

3. Definição do Ambiente de Aprendizagem

Para viabilizar o projeto de agentes inteligentes é necessária a utilização de um ambiente de simulação com objetivos e ações pré-definidas. Nesta seção são apresentados os detalhes do ambiente proposto neste trabalho que simula um jogo de polícia e ladrão.

O ambiente é composto por uma arena quadrada e algumas paredes e dois agentes: um no papel de policial e outro no papel de ladrão. Além disso, existe um tanque de gasolina posicionado na arena que pode ser coletado pelos agentes. O policial se movimenta na arena utilizando um carro com um limite de combustível que pode ser renovado sempre que ele coleta um tanque de gasolina. Apesar do ladrão não ter limite de combustível, pelo fato de se movimentar a pé, a cada quatro movimentos realizados ele deve permanecer parado descansando por um turno. Esta limitação simula o efeito do cansaço.

Os agentes possuem quatro ações disponíveis para realizar na arena: andar para cima, direita, baixo ou esquerda. Em cada iteração da simulação, os agentes realizam de forma paralela uma desta 4 ações. Se um agente tentar se mover para uma posição fora da arena ou para uma posição com parede, este permanecerá imóvel. Se um dos agentes se mover para a localização do tanque de combustível ele o consumirá e um novo tanque de combustível aparecerá em uma nova localização aleatória na arena.

Neste jogo, os agentes possuem objetivos distintos. O objetivo do policial é capturar o ladrão antes que seu combustível acabe, já o objetivo do ladrão é fugir do policial. Sempre que uma ação é realizada pelo policial, este consome um ponto de sua reserva de combustível. Desta forma, se o policial ficar sem combustível o policial perde e o ladrão ganha. Caso o policial consiga se mover para a mesma localização do ladrão, então o policial ganha e o ladrão perde. Apesar do ladrão não precisar de combustível para correr, a cada 4 movimentos ele se cansa e fica uma interação descansando.

O ambiente de simulação pode ser iniciado com diferentes tamanhos de arena. Sendo T um parâmetro que define o tamanho do lado da arena, o valor da reserva de combustível R do policial é definido a partir do tamanho da arena como $R = T \cdot 2 + T/2$. Para fim de exemplificar comportamentos inteligentes no ambiente de jogo, as Figuras 2A e 2B apresentam dois cenários diferentes do ambiente de simulação.

Os símbolos “P”, “L”, “G”, “l” e “.”, representam, respectivamente, o policial, o ladrão, o combustível, as paredes e os espaços em que é permitida a movimentação. As setas representam a próxima ação de movimentação de cada agente. No cenário da Figura 2A o policial está próximo ao ladrão e possui uma grande reserva de gasolina. Nesta situação um comportamento inteligente esperado seria a polícia ir em direção ao

ladrão, enquanto o ladrão tentaria fugir e coletar o combustível para evitar que a polícia o colete. Já na Figura 2B, o ladrão está mais longe do policial, então seria esperado que o policial se movesse em direção ao combustível para reabastecer e depois perseguir o ladrão que permanece em fuga.

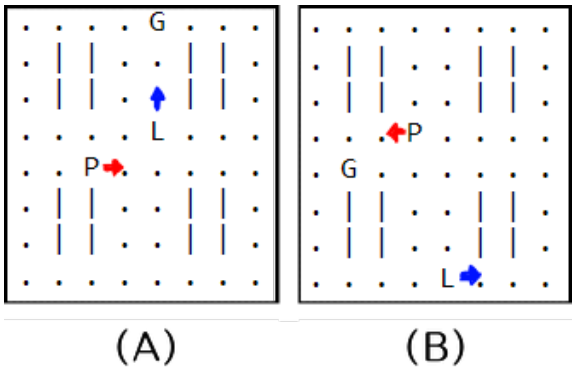

Figura 2. Cenários de exemplo do jogo com arenas de tamanho $T = 8$.

4. Implementação e desenvolvimento

Nesta seção é apresentado o desenvolvimento do ambiente de simulação definido na Seção 3. O ambiente, implementado em Python, é organizado com o uso de classes para auxiliar um iniciante na utilização do ambiente. A classe `Player` serve de base para implementação de agentes. A classe `Arena` armazena o mapa do local onde os agentes podem interagir. Finalmente, a classe `Game` controla a simulação do ambiente garantindo a execução das regras do jogo. O ambiente de simulação apresentado aqui está disponível no seguinte repositório público no GitHub: `https://github.com/super312/Ambiente-de-simula-o-policia-ladrao`

A classe `Player`, apresentada na Figura 3, serve como base para a criação dos agentes inteligentes no ambiente proposto. O construtor desta classe recebe como parâmetro as coordenadas (x,y) em que o agente será posicionado na arena do ambiente.

```
class Player:

    def __init__(self,x,y):

    def move(self,arena,action):

    def get_position(self):

    def next_action(self,gas,gas_x,gas_y,enemy_x,enemy_y,arena):
```

Figura 3. Trecho do código da classe Player.

A função `move` permite mover o agente. Ela recebe como parâmetros o objeto `arena` e uma ação do ambiente (`action`) para ser executada. Um ação pode ser 0, 1, 2 e 3, significando, respectivamente, um movimento para cima, direita, baixo e esquerda. Caso a nova posição seja em uma parede ou saia fora da arena o movimento será cancelado e o agente permanece no mesmo local. Com o `get_position` é possível obter à posição atual do agente na arena. A função `next_action` é a uma função abstrata que

pode ser implementada via herança para definir o comportamento do agente. Esta função é invocada pelo laço principal da simulação fornecendo como parâmetro informações do estado do jogo. Para exemplificar uma especialização desta classe, na Figura 4 é definida uma classe `LadraoAleatorio` implementando a função `next_action`. Nesta implementação, o agente escolhe sempre uma ação de movimento aleatória, ignorando outras informações do estado do jogo.

```
class LadraoAleatorio(Player):

    def next_action(self,gas,gas_x,gas_y,enemy_x,enemy_y,arena):

        return np.random.randint(4)
```

Figura 4. Trecho do código da classe LadraoAleatorio

Na Figura 5, a class `LadraoInteligente` implementa um mecanismo mais sofisticado para fugir da polícia. Basicamente, este agente realiza a ação de movimento escolhendo a direção que mais o afasta do policial.

```
class LadraoInteligente(Player):

    def next_action(self,gas,gas_x,gas_y,enemy_x,enemy_y,arena):

        distancia = [0,0,0,0]

        if not arena.is_wall(self.x,self.y-1):
            distancia[0] = abs(self.x-enemy_x) + abs(self.y-1 - enemy_y)
        if not arena.is_wall(self.x+1,self.y):
            distancia[1] = abs(self.x+1 - enemy_x) + abs(self.y - enemy_y)
        if not arena.is_wall(self.x,self.y+1):
            distancia[2] = abs(self.x-enemy_x) + abs(self.y+1 - enemy_y)
        if not arena.is_wall(self.x-1,self.y):
            distancia[3] = abs(self.x-1 - enemy_x) + abs(self.y - enemy_y)

        return np.argmax(distancia)
```

Figura 5. Trecho do código da classe LadraoInteligente

A classe Arena apresentada na Figura 6 é utilizada pela classe `Game` para armazenar informações do mapa no qual os agentes estarão se movimentando. Esta classe deve ser inicializada com um valor inteiro que representa o tamanho T da arena. T representa o lado da arena, que é quadrada. A função `set_walls` serve para adicionar paredes dentro da arena que servem de obstáculo para os agentes. A função `is_wall` está disponível para verificar se uma posição da arena possui uma parede. Por fim, `get_size`, retorna o tamanho da arena.

```
class Arena:

    def __init__(self,SIZE):

    def set_walls(self):

    def is_wall(self,x,y):

    def get_size(self):
```

Figura 6. Trecho do código da classe Arena

O controle do ambiente é realizado pela classe Game apresentada na Figura 7. Esta classe inicializa todo o ambiente e controla as regras do jogo. No seu construtor, ela recebe o tamanho que será utilizado na arena e os dois agentes que estarão se movendo nela, `p1` e `p2`, representando, respectivamente, o policial e o ladrão. Podemos ver sua estrutura na Figura 7. Já a função `play` permite executar uma partida do jogo. A função `print_state` é utilizada para imprimir no terminal a arena e mostrar em qual posição cada agente e o combustível estão naquele momento. Útil também para depuração. Com a função `update_gas_location` é verificado se um dos agentes coletou o combustível, em caso positivo, adicionado um novo combustível em um local aleatório da arena e restabelece a reserva de combustível caso o policial o pegue. Por fim, a função `end_of_game` serve para verificar se algum dos agentes venceu.

```
class Game:

    def __init__(self,SIZE,p1,p2):

    def play(self, slowdown = True):

    def print_state(self,store = False,a = (-1,-1)):

    def update_gas_location(self):

    def end_of_game(self):
```

Figura 7. Trecho do código da classe Game.

Para exemplificar e validar o ambiente, foi criada uma classe para implementar o algoritmo de treino do *Q-learning*. A Figura 8 apresenta um trecho da função `treino` que implementa em Python o pseudocódigo definido na Figura 1.

```
def treino(self,eps,generations,timelimit):
  epsilon = eps
  epsilon_inicial = eps
  epsilon_final = 0.9
  _lambda = 0.75
  lalpha = 0.9
  for gens in range(generations):
    p = gens / generations
    epsilon = epsilon_inicial + p * (epsilon_final - epsilon_inicial)
    self.p2.x,self.p2.y = np.random.randint(self.SIZE),np.random.randint(self.SIZE)
    while self.arena.is_wall(self.p2.x,self.p2.y):
      self.p2.x,self.p2.y = np.random.randint(self.SIZE),np.random.randint(self.SIZE)
    self.p1_current_gas = np.random.randint(self.p1_gas_maximum) + 1
    self.gas_x,self.gas_y = np.random.randint(self.SIZE),np.random.randint(self.SIZE)
    while self.arena.is_wall(self.gas_x,self.gas_y):
      self.gas_x,self.gas_y = np.random.randint(self.SIZE),np.random.randint(self.SIZE)
    gx,gy = self.gas_x,self.gas_y
    self.starting_location(self.p1_current_gas,gx,gy,self.p2.x,self.p2.y)
    while(not self.terminal_state(self.p1_current_gas,gx,gy,self.p2.x,self.p2.y)):
      action = self.next_action(self.p1_current_gas,gx,gy,self.p2.x,self.p2.y,epsilon)
      old_y,old_x,old_gas,old_gasx,old_gasy,old_ey,old_ex = self.p1.y,self.p1.x,self.p1_current_gas,gx,gy,self.p2.y,self.p2.x
      self.p1.move(self.arena, action)
      self.update_gas_location()
      gx,gy = self.gas_x,self.gas_y
      reward = self.rewards[self.p1_current_gas,gx,gy,self.p2.x,self.p2.y,self.p1.x,self.p1.y]
      old_q_value = self.q_values[old_gas,old_gasx,old_gasy,old_ex,old_ey,old_x,old_y,action]
      temporal_diference=reward+(_lambda* \
                    np.max(self.q_values[self.p1_current_gas,gx,gy,self.p2.x,self.p2.y,self.p1.x,self.p1.y]))-old_q_value
      new_q_value = old_q_value + (lalpha * temporal_diference)
      self.q_values[old_gas,old_gasx,old_gasy,old_ex,old_ey,old_x,old_y,action] = new_q_value
  txtname = "size_" + str(self.SIZE) + "_epsilon_" + str(epsilon_inicial) + "_geracoes_" + str(generations) + "_qvalues.csv"
  pickle.dump(self.q_values, open(txtname,"wb"))
```

Figura 8. Trecho do código da função de treino.

O método `treino`, faz parte da classe `Treino`, que especializa a classe `Game`, tendo acesso às principais informações necessárias do ambiente. Este método recebe o parâmetro `eps` que representa o ε_f, `generations` que representa o número de gerações G de treinamento e `timelimit` que define um tempo limite em segundos para o treino. Ao final do treinamento a *Q-table* é salva em um arquivo csv cujo nome é composto do tamanho da arena, seguido do ε_f, e o número de gerações que foram inseridos. Este arquivo é recarregado posteriormente para realizar as inferências na classe `IAPolicia` definida na Figura 9.

```
class IAPolicia(Player):

    def __init__(self,SIZE,epsilon,generations,x,y):
        super().__init__(x, y)

        try:
            txtname = "size_" + str(SIZE) + "_epsilon_" + str(epsilon) \
                + "_geracoes_" + str(generations) + "_qvalues.csv"
            self.q_values = pickle.load(open(txtname,"rb"))
        except:
            print("Treino não encontrado")

    def next_action(self,gas,gasx,gasy,enemy_x,enemy_y,arena):

        a0 = self.q_values[gas,gasx,gasy,enemy_x,enemy_y,self.x, self.y,0]
        a1 = self.q_values[gas,gasx,gasy,enemy_x,enemy_y,self.x, self.y,1]
        a2 = self.q_values[gas,gasx,gasy,enemy_x,enemy_y,self.x, self.y,2]
        a3 = self.q_values[gas,gasx,gasy,enemy_x,enemy_y,self.x, self.y,3]

        if a0 == 0 and a1 == 0 and a2 == 0 and a3 == 0:
            return np.random.randint(4)

        return np.argmax(self.q_values[gas,gasx,gasy,enemy_x,enemy_y,self.x, self.y])
```

Figura 9. Trecho do código da classe IAPolicia.

Para que se possa atualizar de forma otimizada a *Q-table* é necessário criar uma tabela de recompensas. Para o ambiente proposto neste trabalho é utilizada uma tabela com todos os valores em -1. Nas posições do combustível e do ladrão, são usadas recompensas de $+25$ e $+50$, respectivamente. Na Figura 10 é apresentado um estado da arena a esquerda, e a direita, é apresentada a matriz de recompensas associada. Com esta parametrização o agente policial irá buscar o menor caminho possível para ter o mínimo de punições possíveis antes de chegar a recompensa final.

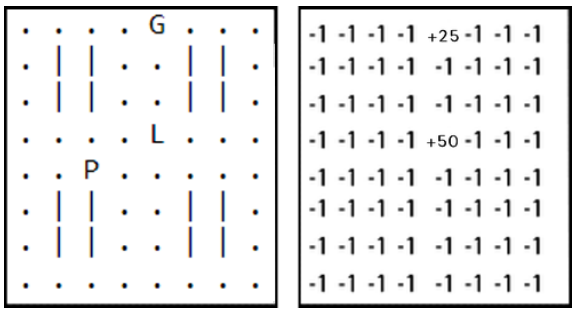

Figura 10. Exemplo de matriz de recompensas de um estado.

5. Experimentos Computacionais

Nesta seção são apresentados experimentos computacionais realizados no treinamento do agente inteligente no papel de policial e de sua avaliação de desempenho no ambiente de simulação proposto. Mais especificamente, os experimentos tem o objetivo de responder as seguintes questões:

i) Qual configuração de parâmetros é apropriada para um treinamento apropriado?
i) Quanto tempo computacional deve ser investido no processo de treinamento?
ii) Qual é a taxa de vitórias do agente inteligente implementado para o policial em comparação com diferentes implementações para o ladrão?

Os resultados dos experimentos foram obtidos a partir de um computador desktop equipado com um com processador AMD Ryzen 5 PRO 4650G (3.70 GHz) com 16GB de memória RAM sobre o sistema operacional Windows 11 Pro. Os algoritmos foram implementados utilizando a linguagem de Programação Python na versão 3.9 e executados via terminal de comando. Para a realização do treinamento dos testes foram utilizadas arenas com tamanho $T \in \{4,6,8,10\}$. Além disso, foram avaliados treinamentos com o método *Q-learning* usando $10^5, 10^6, 10^7, 10^8$ e 10^9 gerações conforme implementação apresentada na Figura 8. Os demais parâmetros de treino foram determinados por testes *ad-hoc* como $\varepsilon_f = 0.9, \alpha = 0.9$ e $\lambda = 0.75$. As recompensas foram definidas conforme configuração apresentada na Figura 10. Em cada treinamento foi estabelecido um tempo limite de 3 horas (10800 segundos). Além disso, para cada treinamento foi avaliada a taxa de vitória do agente policial (Figura 9) em 100 partidas com estado inicial aleatório em comparação com a implementação do Ladrão Aleatório apresentada na Figura 4 e também com o Ladrão Inteligente implementado conforme a Figura 5.

Na Tabela 1 são apresentados os resultados dos treinamentos e das taxas de vitórias obtidas nos testes em partidas enfrentando o Ladrão Aleatório e o Ladrão Inteligente. Na coluna *Tamanho da Arena* são apresentados os resultados para diferentes dimensões da arena. As colunas *Gerações* e *Tempo* apresentam, respectivamente, o número de gerações e o tempo utilizado no processo de treinamento. As últimas colunas apresentam a taxa de vitórias do agente policial em relação as implementações para o agente do ladrão. As execuções que finalizaram por tempo limite são marcadas com o símbolo "$*$". As melhores taxas de vitória nos testes com os dois tipos de ladrão para cada tamanho de arena são identificadas em negrito.

Analisando os resultado da Tabela 1, é possível perceber que o tempo de treinamento utilizado é proporcional ao número de gerações e aumenta ligeiramente conforme o tamanho da arena aumenta. Todos os processo de treinamento terminaram antes do tempo limite, com exceção das execuções com 10^9 gerações para $T \in \{6,8,10\}$. Além disso, é possível observar que a taxa de vitórias do policial aumenta com o número de gerações independente do tipo de implementação para o Ladrão. Este comportamento é esperado, visto que o processo de treinamento tem mais tempo para explorar diferentes estados do ambiente e aprimorar o processo de inferência.

Tabela 1. Resultados do Treinamento e Testes

Tamanho da Arena	Gerações	Tempo (seg)	Ladrão Aleatório	Ladrão Inteligente
$T = 4$	10^5	1	71	10
	10^6	11	**100**	63
	10^7	105	**100**	73
	10^8	1040	99	76
	10^9	10249	**100**	**81**
$T = 6$	10^5	2	36	4
	10^6	15	33	7
	10^7	134	99	31
	10^8	1218	**100**	**78**
	10^9	*10800	**100**	**78**
$T = 8$	10^5	2	27	3
	10^6	19	26	1
	10^7	184	50	3
	10^8	1591	**100**	49
	10^9	*10800	**100**	**74**
$T = 10$	10^5	2	26	1
	10^6	23	20	1
	10^7	225	20	3
	10^8	2101	58	58
	10^9	*10800	**100**	**79**

Em relação a taxa de vitórias, é notável que o agente policial implementado tem a capacidade de vencer sempre a implementação do Ladrão Aleatório dado um treinamento com mais gerações. A medida que o tamanho da arena, em especial no tamanho $T = 10$, a taxa de vitórias com 100% só é alcançada com 10^9 gerações de treino, sugerindo que para tamanhos maiores de arena seria necessário maior investimento de tempo no treinamento para manter a taxa de sucesso alta. Por outro lado, em comparação com o Ladrão Inteligente, a maior taxa de vitórias alcançada variou entre 74 a 81, garantindo uma quantidade significativa de vitórias para o Ladrão a medida que o tamanho do mapa aumenta. Isso acontece pois a medida que o mapa aumenta a única gasolina disponível fica cada vez mais distante do policial e o ladrão ainda possui mais espaço para evadir.

No geral, é possível concluir que o agente inteligente treinado com 10^9 gerações teve um desempenho satisfatório pois venceu a maioria das partidas em todos os tamanhos avaliados quando comparado a ambas as implementações do agente no papel de ladrão. Desta forma, é possível inferir que o *Q-learning* é eficaz para produzir comportamentos inteligentes no ambiente proposto e poderia ser um método promissor para uma implementação mais eficiente para o ladrão.

6. Considerações Finais

Neste trabalho foi realizada a criação de um ambiente de simulação para um jogo de polícia e ladrão com foco em desenvolvedores iniciantes em inteligência artificial. O ambiente proposto além de ser simples, visa auxiliar no ensino da criação de inteligências artificiais, permitindo que se tenha resultados visuais imediatos e com uma menor curva de aprendizado. Além disso, foi proposta uma implementação de um agente inteligente para o policial e realizada uma comparação com duas implementações distintas do agente no papel de ladrão. Os resultado experimentais demonstraram que a metodologia de treino adotada é eficaz para a criação de comportamentos inteligente e também serve de exemplo para utilização do ambiente proposto.

São possibilidades de trabalhos futuros: (i) Avaliar a utilização de uma taxa de aprendizado (α) dinâmica que decresce a medida que são executas as gerações conforme proposto por Even-Dar et al. (2003); (ii) Realizar uma implementação mais sofisticada para o ladrão explorando uma estratégia de coleta de gasolina; (iii) Implementação de uma interface gráfica que permita visualizar o jogo em modo gráfico em tempo real e acompanhar o andamento de cada partida do ambiente.

Referências

Borges, L. E. (2014). *Python para desenvolvedores: aborda Python 3.3*. Novatec Editora.

Chen, M., Dorer, K., Foroughi, E., Heintz, F., Huang, Z., Kapetanakis, S., Kostiadis, K., Kummeneje, J., Murray, J., Noda, I., et al. (2001). Robocup soccer server. *Manual for Soccer Server Version*, 7.

Cord, M. e Cunningham, P. (2008). *Machine learning techniques for multimedia: case studies on organization and retrieval*. Springer Science & Business Media.

de Souza, I. V. (2016). Ambiente de simulaÇÃo baseado no jogo super bomberman para ensino e pesquisa em inteligÊncia artificial. *Universidade Estadual do Sudoeste da Bahia – UESB*.

Dearden, R., Friedman, N., e Russell, S. (1998). Bayesian q-learning. *Aaai/iaai*, 1998:761–768.

Even-Dar, E., Mansour, Y., e Bartlett, P. (2003). Learning rates for q-learning. *Journal of machine learning Research*, 5(1).

Hartness, K. (2004). Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291.

Millington, I. e Funge, J. (2018). *Artificial intelligence for games*. CRC Press.

Rudowsky, I. (2004). Intelligent agents. *Communications of the Association for Information Systems*, 14(1):14.

Russel, S. e Norvig, P. (1996). Artificial intelligence—a modern approach by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, nj. *The Knowledge Engineering Review*, 11(1):78–79.

Stange, R., Cereda, P., e Neto, J. J. (2017). Agentes adaptativos reativos: formalização e estudo de caso. In *Memórias do XI Workshop de Tecnologia Adaptativa–WTA*, pages 63–71.

Sutton, R. S. e Barto, A. G. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the ninth annual conference of the cognitive science society*, pages 355–378. Seattle, WA.