GoToGol

CARLOS VICTOR D. ARAÚJO *
*Ciência da Computação - Mestrado
E-mail: carlos.araujo@students.ic.unicamp.br

DAIANE MENDES DE OLIVEIRA *
*Ciência da Computação - Mestrado
E-mail: daiane.oliveira@students.ic.unicamp.br

JOÃO PAULO K. CASTILHO *
*Ciência da Computação - Mestrado
E-mail: joao.castilho@students.ic.unicamp.br

PEDRO OLÍMPIO PINHEIRO *
*Ciência da Computação - Mestrado
E-mail: pedro.pinheiro@students.ic.unicamp.br

***Resumo* – Na área de robótica, padrões de mobilidade, como movimentação e percepção do ambiente, são importantes tópicos, principalmente em aplicações com robôs para usos exploratórios, militares e/ou comerciais. No presente trabalho aplicamos métodos para movimentação e percepção do ambiente com a proposta de realizar uma comparação de velocidade com que dois robôs atingem o objetivo. Com o uso do simulador *V-REP*, escolhemos usar o robô *Pioneer 3dx* e definimos uma cena final com o ambiente similar a um campo de futebol, com dois robôs, cada um posicionado em um gol e com um objetivo definido no gol oposto e vários obstáculos adicionados de maneira pseudo-aleatória entre um gol e outro. Para fazer com que os robôs atingissem o objetivo, utilizando padrões de movimentação e percepção do ambiente, selecionamos implementar um método com lógica *Fuzzy* e dois com aprendizado com reforço: *Q-Learning* e *Deep Q-Network* (DQN). Ao final, treinamos o robô com os métodos de aprendizado por reforço e comparamos cada um com o método de lógica *Fuzzy*. O robô com o método de lógica *Fuzzy* conseguiu alcançar o objetivo desviando corretamente de obstáculos, porém em temos de velocidade o robô usando o método de aprendizado por reforço, *Q-Learning*, conseguiu um resultado melhor.**

Palavras-chave* – Lógica *Fuzzy*, Aprendizagem por Reforço, *Pioneer 3dx

I. INTRODUÇÃO

O desenvolvimento da robótica, em alguns aspectos, foi devido à necessidade de criar sistemas capazes de lidar com tarefas que, de certa forma, são repetitivas ou perigosas para um ser humano. Para estes casos, vários robôs com especialidades diferentes foram desenvolvidos, de modo que muitos diferenciam-se principalmente pelos seus padrões de mobilidade, que é um importante tópico em torno da robótica [1], especialmente se tratando de robôs para usos exploratórios, militares e/ou comerciais.

Dentre as diferentes aplicações, vale ressaltar as aplicações desenvolvidas com o intuito de explorar diferenças de eficiência entre métodos aplicados, seja para movimentação, percepção do ambiente ou qualidade do resultado final. Um exemplo disso se aplica à comparação entre velocidade com que dois robôs atingem o objetivo utilizando padrões de movimentação baseados em redes neurais treinadas e lógica *Fuzzy* ou comparar a qualidade de um mapa do ambiente gerado a partir da extração de características com odometria e *ground truth*.

Este trabalho visa analisar a diferença de desempenho de acordo com a utilização de lógica *Fuzzy* e dois diferentes tipos de aprendizagem por reforço, *Q-Learning* e *Deep Q-Network* (DQN) para fazer com que o robô percorram o ambiente para chegar em um ponto objetivo, de modo que seja necessário desviar de obstáculos que estão dispostos no cenário. Os experimentos foram simulados usando o robô *Pioneer 3dx* no V-REP, focando na diferença de velocidade com que a rota é realizada pelo robô de acordo com a abordagem para o padrão de movimento e desvio de obstáculos.

Este trabalho encontra-se organizado da seguinte forma: a seção II apresenta trabalhos encontrados na literatura que mencionam as técnicas utilizadas. A seção III descreve os algoritmos utilizados. A seção IV apresenta como foi feita a divisão das tarefas do trabalho. Os resultados e análise dos experimentos realizados estão apresentados na seção V, e as conclusões são apresentadas na seção VI.

II. TRABALHOS RELACIONADOS

É comum o uso de lógica *Fuzzy* em robótica, veja por exemplo [2], [3]. Em [4] é apresentado um controlador utilizando lógica *Fuzzy* para que o robô imite comportamentos biológicos tais como evitar obstáculos e seguir paredes. O experimentos foram aplicados em um cenário real e permitiu ao robô demonstrar comportamentos inteligentes em ambientes complexos, gerando assim resultados bastante satisfatórios para o controlador baseado em lógica *Fuzzy*.

É extremamente extensa a literatura acerca da utilização de aprendizagem por reforço em robótica, veja por exemplo [5]. Mais especificamente, o método chamado de Q-Learning [6]. Em [7] é apresentado um *benchmark* de algoritmos de aprendizagem por reforço para robôs em um cenário real, a análise dos autores ressaltou que os algoritmos são altamente sensíveis à mudanças de parâmetros, mas que são rápidos no processo de treinamento e que obtiveram bons resultados quando aplicados à robôs reais. Foram executados mais de 450 experimentos independentes que levaram cerca de 950 horas de uso de robôs.

III. METODOLOGIA

Neste trabalho, primeiro escolhemos usar o robô *Pioneer 3dx* e, com o uso do simulador *V-REP*, definimos uma cena final com o ambiente similar a um campo de futebol, com dois robôs. Cada um deles está, inicialmente, posicionado em um gol, com um objetivo definido no gol oposto e vários obstáculos adicionados de maneira pseudo-aleatória entre um gol e outro. A Figura 1 é uma ilustração do cenário utilizado no simulador *V-REP*.

Na sequência, buscamos utilizar duas técnicas de aprendizado por reforço para treinar o robô *Pioneer 3dx* e chegar até um ponto objetivo no cenário com vários obstáculos e comparar com uma abordagem utilizando lógica *Fuzzy*. As técnicas escolhidas para trabalhar com aprendizado por reforço foram *Q-Learning* e *Deep Q-Network* (DQN). Por fim, realizamos as fases de implementação e testes com o uso da linguagem de programação *Python*, trabalhando com o repositório do GitHub[1] e de bibliotecas e softwares como: V-REP[2], Jupyter Notebook[3], NumPy[4], scikit-learn[5], Keras[6] e Matplotlib[7]. A Figura 2 apresenta a sequência de fases que compõem a metodologia deste trabalho.

Aprendizado por reforço é um algoritmo que busca treinar um agente a tomar determinadas ações que sigam uma política previamente estabelecida, mesmo quando o ambiente que o agente está sendo executado é desconhecido. Este algoritmo parte da premissa de que a cada iteração o agente encontra-se em um estado s e vai para um estado s', tomando uma ação. Cada ação tomada nos estados geram uma recompensa para o agente, que pode ser boa ou ruim. O objetivo final é tomar um conjunto de ações que maximizem essa recompensa.

Em particular, para este trabalho usamos as técnicas *Q-Learning* e *Deep Q-Network* (DQN). A seguir, descrevemos mais detalhes de cada uma e de como aplicamos para alcançar nosso objetivo no trabalho.

A. Q-Learning

O *Q-Learning* é um algoritmo que busca aprender uma política que ensinará ao agente qual ação tomar em determinadas circunstâncias. Esse aprendizado é armazenado em uma tabela, chamada de *matriz Q*. Nesta tabela, as linhas representam o conjunto de estados possíveis para o agente e as linhas representam as ações que podem ser tomadas. Cada ação leva de um estado s para um estado s'. Denotamos esta transição por $Q(s, a)$, em que s é o estado inicial e a é a ação tomada. Essa transição gerará uma recompensa r para o agente. Finalmente, o valor de $Q(s, a)$ é atualizado na matriz Q, de acordo com a seguinte equação:

$$Q(s, a) = Q(s, a) + \alpha((r + \gamma \max_{a'}(Q(s', a')) - Q(s, a))),$$

em que α indica a taxa de aprendizado, ou seja, o quanto a nova informação calculada é relevante, e γ representa o fator de desconto, ou seja, a relevância das recompensas que o agente virá obter no futuro tomando a ação a.

1) Go To Goal com Q-Learning: Nesta etapa do projeto, objetivamos construir um algoritmo de aprendizado por reforço utilizando *Q-learning* para fazer o robô *pioneer 3dx* chegar até um ponto objetivo, supondo que não há obstáculos à sua frente.

Neste sentido, nosso objetivo é criar uma política para ensinar ao nosso agente, o robô, qual ação deve ser tomada em uma determinada circunstância. Para isso, este método consiste em atribuir uma recompensa para cada ação que o robô teve. Esta recompensa é estabelecida por uma política que analisa a consequência de cada ação.

Deste modo, precisamos de alguma maneira de identificar se, dado o objetivo (chegar ao ponto no mapa), a ação tomada pelo robô foi boa ou ruim. Para isso, definimos um conjunto de estados. Nosso agente encontra-se em um determinado estado se atente às suas restrições. Para o nosso problema, definimos o estado do robô de acordo com o ângulo que ele se encontra em relação ao objetivo. As etapas da determinação do estado são descritas a seguir:

1) Nesta etapa, queremos determinar o ângulo que o robô está do objetivo. O robô sempre está entre 0 e 180 graus de um objeto, e à direita ou à esquerda deste. Portanto, mapeamos de 0 a 180 graus se o robô está virado para a esquerda do objetivo e de -180 a 0 graus se o robô está virado para a direita do objetivo. A Figura 3 ilustra uma situação em que o robô está virado 130 graus à direita do objetivo e, por esse motivo, dizemos que o robô está a -120 graus do seu alvo. Na figura, o círculo preto representa o ponto em que o robô quer chegar e o círculo branco, o robô.
2) Definir o conjunto de estados de acordo com o ângulo. Nesta etapa, criamos um conjunto de N *clusters*, que serão os nossos estados. Cada cluster é composto por um intervalo uniforme de ângulos, de modo que os N clusters compõe os 360 graus possíveis para o robô. O robô encontra-se em um determinado cluster – ou estado – se e somente se o seu ângulo está neste cluster. A Figura 4 ilustra uma subdivisão dos ângulos em 12 clusters. Note que se o robô estiver a -13 graus do objetivo, ficará no cluster entre -15^o e 15^o. Por outro lado, se ele estiver a 120 graus, ficará no cluster entre 105^o e 135^o. Para nossa implementação, definimos um total de 120 clusters.

Posteriormente, com os estados já formalizados, definimos as ações que o o robô pode tomar. São elas: andar para frente; virar à esquerda; virar à direita.

Com esse conjunto de estados e com essas ações, temos quase tudo o que precisamos para levar nosso robô até um ponto objetivo. Neste momento, o que queremos é uma boa política para determinar se uma ação tomada pelo robô, encontrando-se em um estado qualquer, foi boa ou ruim.

[1] https://github.com/
[2] http://www.coppeliarobotics.com/
[3] https://jupyter.org/
[4] https://numpy.org/
[5] https://scikit-learn.org
[6] https://keras.io/
[7] https://matplotlib.org/

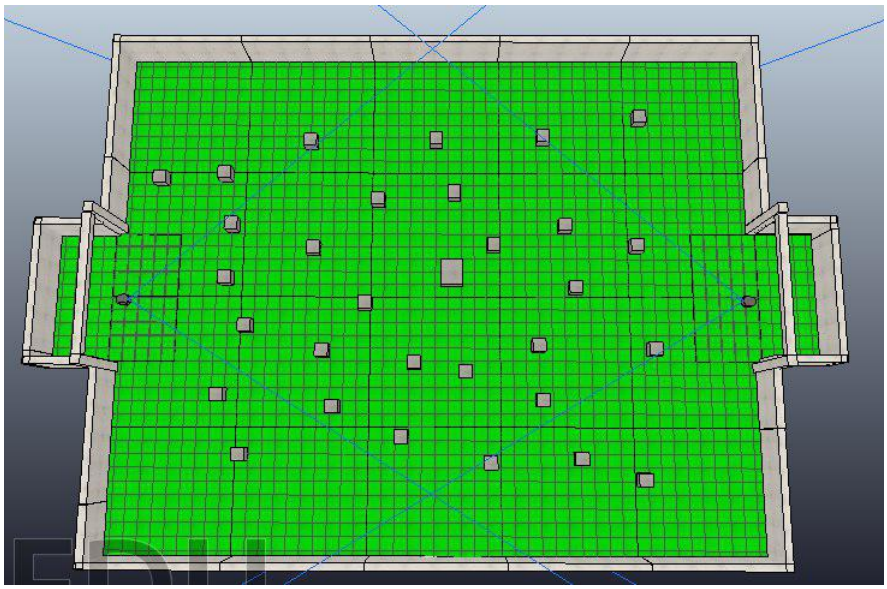

Figura 1: Cenário utilizado no simulador *V-REP*.

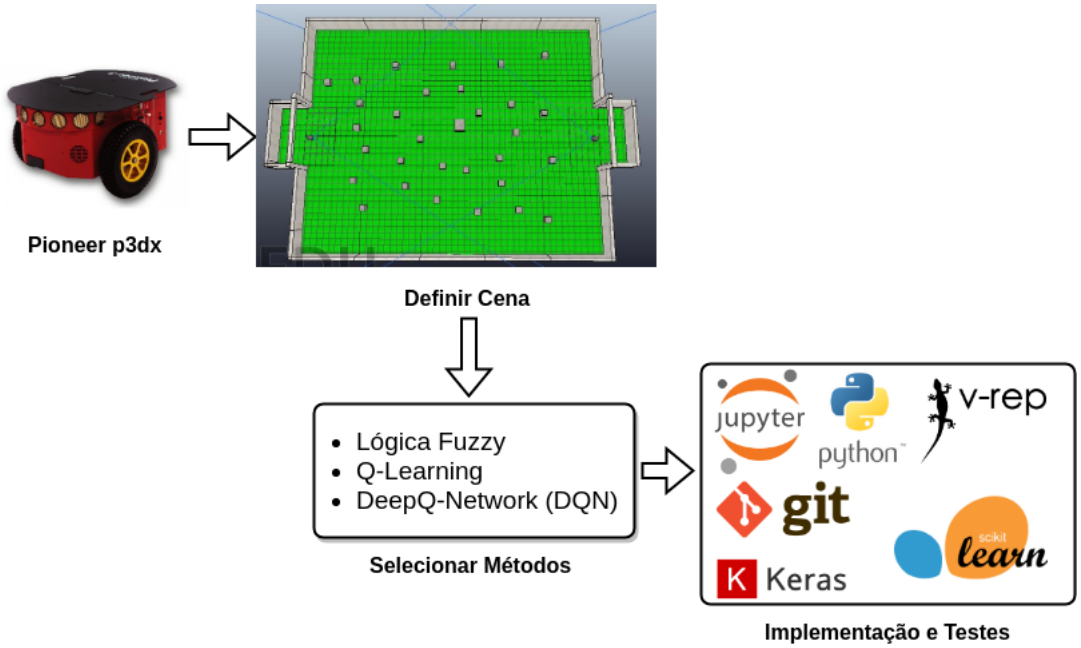

Figura 2: Metodologia aplicada.

Para isso, definimos uma função de recompensa, que atribui um valor para cada ação realizada. Esta função calcula um $\Delta_a = |old\_angle| - |new\_angle|$, em que old_angle representa o ângulo que o robô estava do objetivo antes de realizar a ação e o valor new_angle representa o ângulo que o robô ficou depois de tomar a ação. Note que quanto mais o valor de new_angle se aproxima de zero, maior será o valor de Δ_a. Além disso, a função de recompensa também calcula o quanto o robô se aproximou do objetivo: $\Delta_d = old\_d - new\_d$. Neste cálculo, old_d é a distância do robô ao objetivo antes de tomar a ação, e new_d é a distância depois de tomar a ação. Similarmente ao Δ_a, quanto mais próximo do objetivo, maior será o valor de Δ_d. Esta função de recompensa retorna, por fim, $\Delta_a + 50\Delta_d$.

Depois de definidos os estados possíveis, quais ações são possíveis e qual a política de recompensa, chegou a hora de treinar nosso robô no algoritmo. Para isso, inicialmente, definimos como 90% a probabilidade do robô tomar uma ação aleatória, ou seja, ignorar a política. Além disso, iniciamos o algoritmo com uma taxa de aprendizado de 60%. Esses valores foram utilizados porque mostraram os melhores resultados entre diversos testes realizados.

Posteriormente, para cada ação tomada, calculamos a con-

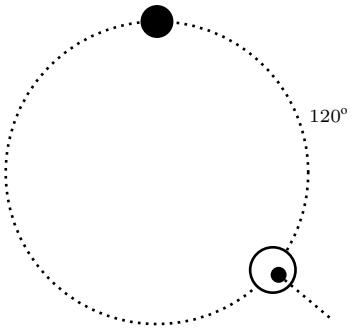

Figura 3: Um exemplo do robô a 120 graus do objetivo. O círculo preto representa o alvo, o círculo branco representa o robô, com sua frente indicada na figura.

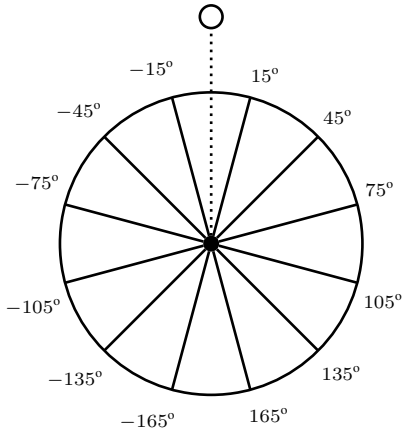

Figura 4: Divisão dos ângulos em clusters.

sequência ($Q(s, a)$) desta ação quando realizada no estado s:

$$Q(s, a) = learning\_rate \times (reward + 0.99 \times best_q - Q(s, a)).$$

Neste cálculo, $learning\_rate$ é a taxa de aprendizado, $reward$ é a recompensa obtida pela ação realizada e $best_q$ é o valor da melhor ação já calculada para o estado que a ação tomada levou.

O objetivo deste treinamento é fazer com que o robô tome ações que maximizem sua recompensa após treinado. A cada iteração, a probabilidade do robô tomar ações aleatórias diminui, assim como a taxa de aprendizado.

2) Avoid Obstacles com Q-Learning: Nesta etapa do projeto, buscamos treinar o robô para ser capaz de desviar de obstáculos. É importante ressaltar que esta implementação não leva em conta um destino final, apenas faz com que o robô não colida com possíveis objetos no seu caminho.

Para isso, assim como na etapa anterior para implementar o *Go To Goal* utilizamos Q-Learning. Como o objetivo agora é evitar obstáculos, consideramos nos estados o valor de leitura de N sensores. Para cada sensor, temos a possibilidade do sensor estar captando algum objeto à sua frente, ou não. Por esse motivo, temos um total de 2^N estados. Neste trabalho, realizamos a implementação do algoritmo com $N = 2$.

Para fazer um mapeamento da leitura dos sensores para um estado, representamos cada sensor com um *bit* de uma sequência, de modo que se o sensor estiver captando um objeto, o *bit* mapeado tem o valor um, caso contrário, zero. A Figura 5 ilustra dois casos. No primeiro, nenhum sensor está captando objetos, portanto o estado mapeado é o estado zero. Ademais, no segundo caso, apenas o sensor da esquerda está captando um objeto, portanto este é o estado 2. Note que o *bit* menos significativo representa o sensor mais à direita e o mais significativo, o sensor à esquerda.

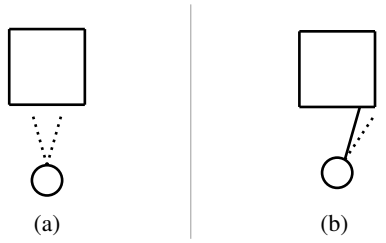

Figura 5: Exemplos de possíveis leituras dos sensores no cenário. A linha pontilhada indica que o sensor não está captando nenhum objeto e a linha contínua indica que o sensor está captando o objeto.

Posteriormente definimos o conjunto de estados. Das mesma forma que fizemos para o *Go To Goal*, demos ao robô as três possibilidades: andar para frente; virar à direita; virar à esquerda.

Para as recompensas, definimos um conjunto que busca beneficiar o robô sempre que ele consegue andar para frente e o penaliza caso ele bata em algum obstáculo. Neste sentido, as recompensas são atribuídas da seguinte forma: $+10$ caso tome a ação de andar para frente; 0 caso ele tome a ação de virar para um dos lados; -100 caso colida com alguma coisa.

Para o treinamento, aplicamos taxa de aprendizado, α, e fator de desconto, γ, constantes, com valores de 0.5 e 0.8, respectivamente. Além disso, no início do algoritmo, o robô tinha uma probabilidade de 90% de tomar uma ação aleatória e, conforme as iterações baixavam, este valor decrementava. Assim como no treinamento para o método Go To Goal, os parâmetros foram escolhidos com a partir de testes e selecionados aqueles com melhores resultados.

B. Lógica Fuzzy

O que é central na lógica *fuzzy* é que, diferentemente dos sistemas lógicos clássicos, ela visa modelar os modos imprecisos de raciocínio que desempenham um papel essencial na capacidade humana de tomar decisões racionais em um ambiente de incerteza e imprecisão. Essa capacidade depende, por sua vez, de nossa capacidade de inferir uma resposta aproximada a uma pergunta com base em um estoque de conhecimento inexato, incompleto ou totalmente não confiável [8].

Para isso, as preposições na lógica *fuzzy* não assumem somente valores de verdadeiro ou falso, mas podem assumir valores intermediários. Por exemplo, sendo verdadeiro como

o valor 1 e falso como um valor 0, uma preposição pode ter valor 0.5, o que significa que ela é 0.5 verdadeira.

Nos modelos usados neste trabalho, tanto as informações de entrada, *Antecedentes*, quanto as informações de saída, *Consequentes*, são codificados e processados usando lógica *fuzzy*.

1) Go To Goal com Lógica Fuzzy: A função Go To Goal implementada utilizando lógica Fuzzy é responsável pela movimentação do robô a partir de um ponto inicial até um ponto destino no cenário (gol adversário). A ideia desta função é fazer o robô rotacionar em direção ao ponto destino e ir seguindo em frente naquela direção, podendo efetuar algumas rotações de menor velocidade ao longo do caminho que está percorrendo. As definições para a lógica Fuzzy que foram feitas são as seguintes:

Antecedentes: O antecedente "distance" é responsável por elencar a distância da localização atual do robô e o destino. Ele possui 4 níveis de distância, que são: "none", quando robô está no ponto destino; "small", quando o robô encontra-se em uma distância relativamente próxima do ponto objetivo; "medium", que indica uma distância maior que "small" mas menor que "big", indica uma distância grande, podendo assim aumentar a velocidade do robô ao máximo.

O antecedente "angular distance" se responsabiliza por definir o valor de rotação que deve ser feito para que o Pioneer esteja de frente para a direção correta do ponto objetivo. Possui 4 níveis de rotação: "small from left", que indica uma rotação a esquerda com movimento em ambas as rodas; "smaller from left" se trata de uma rotação mais lenta e sutil à esquerda, com movimento apenas em uma roda. "small from right" e "smaller from right" se tratam das mesmas definições, agora aplicadas para a direta, ao invés de esquerda.

Consequentes: as ações a serem tomadas se tratam da velocidade que será aplicada em cada roda do robô. Aumentar a velocidade em uma das rodas e reduzir na outra faz com que ele efetue uma curva mais sutil. Girar uma roda no sentido contrário da outra, com a mesma velocidade, fará o robô girar no próprio eixo para o lado da roda que está com velocidade negativa. Manter as duas rodas com mesma velocidade faz com que o robô siga reto na direção ao qual ele está de frente.

As regras estão definidas em três conjuntos principais, que se aplicam as rodas direita e esquerda, simultaneamente.

A primeira regra seta a velocidade das rodas esquerda e direita como zero, ou seja, o robô atingiu seu destino. As regras dois e três são utilizadas para distâncias longas do ponto atual do robô para o objetivo, de modo que ele irá efetuar curvas com velocidade em ambas as rodas (maior velocidade na roda inversa á direção da curva). Regras quatro e cinco se tratam de rotação no próprio eixo do robô, ou seja, uma roda parada e outra com velocidade baixa, essa regra se aplica predominantemente quando o robô está bem próximo na posição objetivo. As regras seis e sete são para seguir o objetivo, se o robô estiver distante e na direção correta, pode-se adicionar velocidade máxima em ambas as rodas, enquanto se a distâncias do objetivo for menor, é necessário que haja uma diminuição da velocidade. As regras foram mapeadas individualmente para a Tabela I.

Regra	**Distancia**	**Distancia Angular**	**Vel. Roda Direita**	**Vel. Roda Esquerda**
1	None	-	Zero	Zero
2	Big	Smaller From Left and Not Small From Right	High	Low
3	Medium	Smaller From Left and Not Small From Right	High	Low
4	Small	Small From Right	Low	Low
5	Small	Small From Left	Low	Low
6	-	Smaller From Left	Low	Zero
7	-	Smaller From Right	Zero	Low
8	-	Small From Left	High	High
9	-	Small From Right	High	High

Tabela I: Regras de Lógica Fuzzy para Go To Goal

2) Avoid Obstacles com Lógica Fuzzy: Para o decidir as ações do robô usando Lógica *Fuzzy*, também foi necessário desenvolver um conjunto de regras que faz com que o robô desvie de obstáculos. Essas regras não levam o robô ao seu destino final, somente evitam que ele colida com quaisquer obstáculos que existam em sua rota.

Para isso, definimos um modelo de lógica *Fuzzy*. As entradas do modelo são dadas como Antecedentes (ou *Antecedent*) e as saídas como Consequentes (ou *Consequent*). Além disso, criamos regras para associar os Antecedentes com os Consequentes e decidir quais ações devem ser tomadas.

Existem três dados necessários como **Antecedentes**, que são as medições dos três sensores frontais, um frontal esquerdo, um frontral direito e um último frontal central. Como o robô não conta com um sensor frontal central, a medição para o sensor frontal utilizada é o mínimo dos dois sensores frontais do robô Pioneer 3-DX. Esses dados são então classificados em dois conjuntos fuzzy: **perto** (ou *near*) e **longe** (ou *far*).

Existem duas informações dadas como saída do modelo fuzzy através dos **Consequentes**: a velocidade e a direção. O dado da velocidade pertence aos conjuntos **parado** (ou *no movement*), **lento** (ou *slow*) e **rápido** (ou *fast*). A direção pode pertencer aos conjuntos **esquerda** (ou *left*), **frente** (ou *forward*) ou **direita** (ou *right*).

Com as informações obtidas pelos sensores, o modelo de lógica *Fuzzy* decide qual a melhor ação que o robô deve tomar para evitar obstáculos. O conjunto de regras usadas para isso são exibidos na Tabela II.

Regra	Antecedentes			Consequentes	
	Sensor Esquerdo	Sensor Frontal	Sensor Direito	Velocidade	Direção
1	Perto	Perto	Perto	Parado	Esquerda
2	Longe	Perto	Longe	Lento	Frente
3	Perto	Longe	Perto	Lento	Frente
4	Longe	Longe	Longe	Rápido	Frente
5	Perto	Perto	Longe	Lento	Esquerda
6	Longe	Perto	Perto	Lento	Direita
7	Perto	Longe	Longe	Rápido	Frente
8	Longe	Longe	Perto	Rápido	Frente

Tabela II: Regras para Avoid Obtacles com *fuzzy*.

C. Máquina de estados para juntar os métodos Go To Goal e Avoid Obstacles

Como mostrado anteriormente, para os métodos utilizando Q-Learing e lógica *Fuzzy*, as funções responsável por fazer o robô ir até o objetivo e evitar obstáculos foram implementadas separadamente.

Como nosso objetivo é fazer o robô ir até um destino evitando todos os obstáculos à sua frente, precisamos de um método para juntar cada uma das funcionalidades implementadas. Neste sentido, desenvolvemos uma máquina de estados, que busca identificar possíveis obstáculos à frente do trajeto que está sendo percorrido pelo robô.

A máquina possui dois estados. O primeiro deles identifica que o caminho está livre, ou seja, o robô pode continuar utilizando a função de ir até o objetivo, pois não há nenhum obstáculo à sua frente nesse momento. Entretanto, pode acontecer do robô se deparar com um objeto inesperado à sua frente e, por esse motivo, deve ativar a função de desviar de obstáculos. Este é o segundo estado.

Para identificar em qual estado o robô se encontra atualmente, utilizamos oito sensores, como mostra a Figura 6. Na figura, os segmentos de reta em azul representam a leitura dos sensores. Na nossa lógica, se algum dos sensores considerados captar um objeto a uma distância de pelo menos 0.5 unidades, então a máquina leva para o estado *desviar obstáculos*. Caso contrário, ou seja, nenhum sensor está captando a uma distância de pelo menos 0.5 unidades, a máquina leva para o estado *ir ao objetivo*.

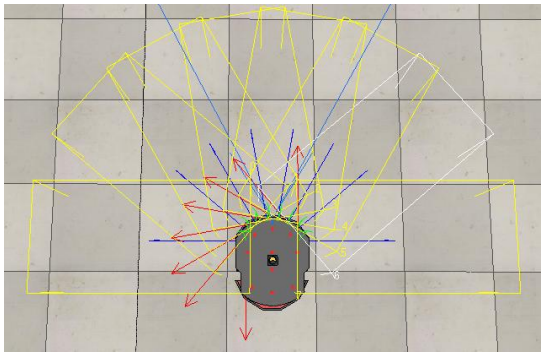

Figura 6: Sensores identificados para determinar qual estado o robô se encontra.

D. Deep Q-Network (DQN)

DQN, diferente da técnica de Q-Learning em que as iterações buscam atualizar a tabela Q, é uma técnica que possui um modelo de inferência (rede neural) que gera valores para cada uma das ações possíveis. Um benefício ao adicionar redes neurais no lugar de uma tabela Q, é a facilidade em aumentar significativamente a complexidade do ambiente, sem necessariamente exigir mais memória. Na Figura 7 é apresentada uma comparação entre a técnica a Q-Learning e a *Deep Q-Network*.

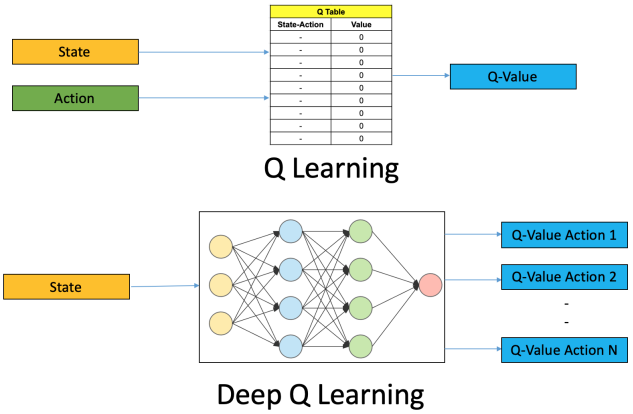

Figura 7: Comparação das técnicas Q-Learning e *DQN* [9].

1) Go To Goal + Avoid Obstacles com DQN: Na implementação da técnica DQN, algumas dependências foram cruciais. Para definir o modelo de rede neural, foi usada a biblioteca do *keras* e, para criar a instância da DQN, usamos o módulo *keras-rl*. Após instalar as dependências necessárias, iniciamos a implementação criando a classe *EnvRobot* que começa com um robô, um ponto objetivo e um conjunto de ações: virar à direita, virar à esquerda e ir para frente. Para ser um ambiente válido e aplicável à instância DQN do módulo *keras-rl*, o objeto *EnvRobot* necessita definir a função de *reset()* e *step(action)*.

A função *reset()*, como o próprio nome já diz, serve para reiniciar o ambiente em cada rodada. A função *step(action)* é usada a cada iteração, recebe uma ação, predita pelo modelo, aplica a ação, calcula e retorna o ganho ou recompensa da ação. O cálculo do ganho na função implementada foi feito pensando nas duas ações Go To Goal e Avoid Obstacles. Após algumas iterações observando o comportamento do robô, definimos a função de recompensa, ver Equação 1.

$$Recompensa = \Delta_a + \Delta_d - 2 \times flag \tag{1}$$

Nesta função, $\Delta_a = old\_angle - new\_angle$ é a diferença de ângulo do estado do robô antes e depois de aplicar a ação, $\Delta_d = old\_distance - new\_distance$ é a diferença de distância para o objetivo antes e depois de aplicar a ação e $flag$ indica se o robô está diante de algum obstáculo. Foi usado $2 \times flag$ na ideia de penalizar o modelo caso tenha obstáculo na frente do robô e ele não desvie.

Após o ambiente estar bem definido, o próximo passo foi pensar e definir o modelo. O modelo definido foi uma rede neural convolucional, que permite usar praticamente qualquer ambiente e tamanho, e com quase qualquer tipo de tarefa visual, pelo menos as que possam ser representadas visualmente. Apesar de não usarmos nada de tarefa visual, esse tipo de rede nos chamou a atenção por ser mais robusta. Deste modo, criamos o modelo, após algumas mudanças, com três camadas densas de 32 dimensões, com a função de ativação *relu* entre elas, adicionamos uma camada com a mesma dimensão do conjunto de ação, ou seja, 3 dimensões. Finalizando com a

função de ativação *linear*. A Figura 8 apresenta o resumo do modelo completo.

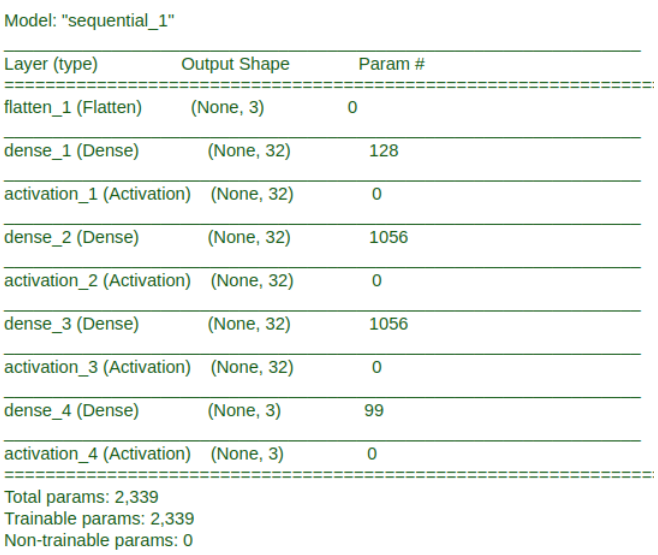

Figura 8: Modelo definido.

Na sequência, usando a função *DQNAgent()* do módulo *keras-rl*, criamos o objeto DQN da técnica. A função *DQNAgent* recebe como parâmetro 3 primordiais objetos: o modelo (definido previamente), uma política e uma memória. Definimos nossa política como *BoltzmannQPolicy* que transforma o comportamento de exploração do agente em um espectro entre escolher a ação aleatoriamente (política aleatória) e sempre escolher a ação mais ideal (política gananciosa) e nossa memória como Memória Sequencial, pois queremos armazenar o resultado das ações que realizamos e as recompensas que obtemos por cada ação.

Em seguida, usamos o otimizador *Adam* da biblioteca do *keras*, juntamente com a métrica de acurácia para compilar o agente com os modelos que foram explorados no treinamento e no teste. Ao final de cada treinamento salvamos os pesos em um arquivo e a cada nova rodada verificamos se há algum arquivo com o nome pré-definido (dqn_weights.h5f) na raiz do programa. Caso exista, os pesos do modelo são carregados antes de iniciar mais uma rodada.

Por fim, é hora de treinar o modelo. Para isso, usamos a função *fit()* passando como parâmetro 1000 para o número de passos e ativando o modo de exibição a cada 100 iterações. Apesar disso deixar o treinamento mais lento, foi usado para acompanhar o quão bem o modelo estava evoluindo. Após algumas rodadas de treinamento, usamos a função *test()* para testar nosso modelo de aprendizado por reforço.

E. Organização dos Códigos

Todos os códigos utilizados neste trabalho podem ser encontrados em um repositório público disponível em https://github.com/pedroonop/MO651/tree/master/P4.

Foram feitas alterações no arquivo *src/robot.py*, de modo que é possível se conectar e controlar dois robôs ao mesmo tempo em uma mesma cena. Para isso, os robôs, seus sensores e atuadores devem ter um identificador no final de seus nomes, e esse identificador deve ser passado como argumento no momento em que o robô é instanciado. Além disso, foram adicionados métodos de sincronização ao aplicar uma ação e para chamada de função interna da API do V-REP, no caso foi usado para chamar a função *reset()* ao aplicar a técnica DQN.

No diretório *codes/* estão os códigos para executar o treinamento do *Go To Goal* e *Avoid Obstacles* usando Q-Learning (respectivamente *GoToGoal.ipynb* e *AvoidObstacle.ipynb*) e suas respectivas matrizes Q (arquivos *Q_matrix_gtg.txt* e *Q_matrix_ao.txt*).

No mesmo diretório (*codes/*) estão os arquivos que controlam o robô para ir ao objetivo evitando obstáculos, usando lógica *fuzzy* (*fuzzy.ipynb*, que controla o robô, também para ir ao objetivo evitando obstáculos, usando *Q*-Learning (*Q-learning.ipynb*) e que controla os dois ao mesmo tempo (*Go To Goal.ipynb*).

Por fim, o código que executa o treino e os experimentos usando *DQN*-Learning também está nesse diretório (com o nome *DQNAgent.ipynb*).

IV. Divisão de Tarefas

De maneira geral, podemos dividir este trabalho em 6 atividades distintas:

1) Implementação *Go To Goal* e *Avoid Obstacle* usando DQN
2) Implementação *Go To Goal* e *Avoid Obstacle* usando *Q*-Learning
3) Implementação *Go To Goal* e *Avoid Obstacle* usando Lógica *Fuzzy*
4) Simulação e análise das implementações em conjunto
5) Elaboração da apresentação para o WTD
6) Escrita e revisão do relatório

Atividade	Carlos Araújo	Daiane Mendes	João Castilho	Pedro Olímpio
(1)	×	×		
(2)			×	×
(3)	×			×
(4)		×	×	
(5)	×	×	×	×
(6)	×	×	×	×

Tabela III: Distribuição das atividades

V. Resultados e Discussão

A seguir, apresentamos as análises dos resultados encontrados durante o treino e simulação dos diferentes métodos abordados neste trabalho.

Em todas as imagens desta seção, quando exibida uma linha com uma extremidade no robô, essa linha indica o caminho percorrido pelo robô durante a simulação.

A. Go To Goal com Q-Learning

Iniciamos o treinamento com um número indeterminado de iterações. Cada iteração iniciava com o robô em um ponto aleatório do ambiente, de modo a evitar que o treinamento ficasse tendencioso.

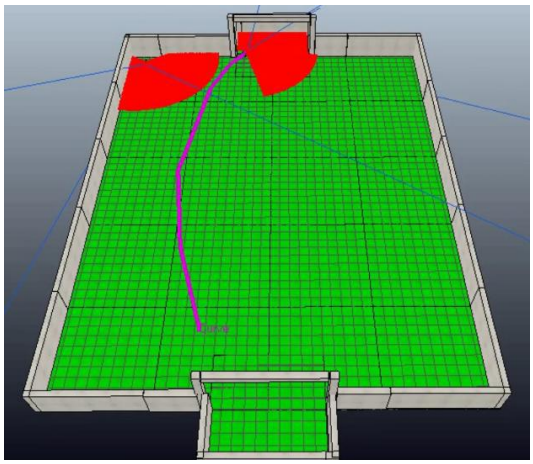

Figura 9: Caminho percorrido pelo robô até o objetivo com a lógica Q-Learning.

Após 1885 iterações executadas, percebemos que o algoritmo havia convergido de modo que, para qualquer ponto do cenário que o robô iniciava, ele ia até o objetivo. A Figura 9 ilustra o caminho percorrido pelo robô até o objetivo em um ponto do cenário.

Em alguns testes, a orientação inicial do robô era contrária ao objetivo e, ainda assim, o robô corrigia sua orientação e alcançava o objetivo.

B. Avoid Obstacles com Q-Learning

Assim como o método para ir ao objetivo, iniciamos o treinamento com um número indeterminado de iterações. O robô também iniciava cada iteração em um ponto aleatório do cenário.

Após 1284 iterações, o robô já era capaz de evitar obstáculos. Percebemos que no início do treinamento, o robô havia convergido de modo que se apenas o sensor da direita captava obstáculo, o robô virava à esquerda, e se apenas o sensor da esquerda captasse um obstáculo, o robô virava à direita. Entretanto, isso fazia com que o robô entrasse em um ciclo quando havia um obstáculo imediatamente à esquerda e à direita dele. Esse fator acarretou que, quando o robô se depara com obstáculos, ele vira sempre à esquerda. A Figura 10 ilustra o percurso do robô desviando obstáculos.

C. Junção dos métodos implementados com Q-Learning

Para juntar os métodos, utilizamos a máquina de estados explicada na Seção III. O resultado encontrado está ilustrado na Figura 11.

Pela Figura 11 é possível observar que o robô segue um trajeto que desvia dos obstáculos e vai até o objetivo da maneira esperada.

Mesmo o robô não tomando a menor rota, em partes porque vira sempre à esquerda quando encontra um obstáculo, o robô desviou perfeitamente de todos os objetos que encontrou no caminho e sempre se aproximando do objetivo (gol) quando não impedido por obstáculos.

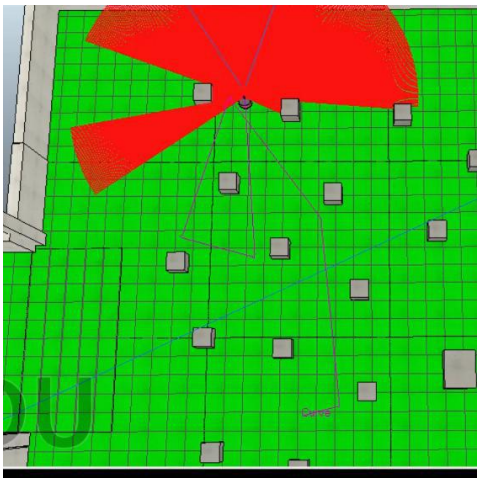

Figura 10: Robô evitando obstáculos com a lógica Q-Learning.

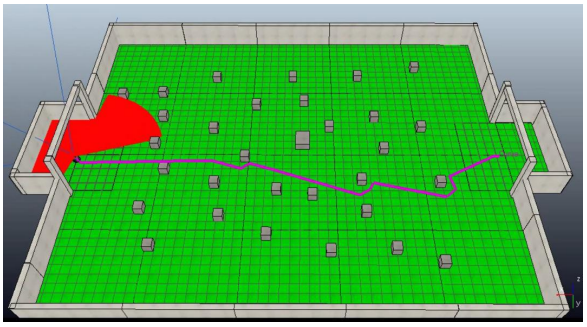

Figura 11: Execução do método ir até o objetivo junto com o método de evitar obstáculos com Q-Learning.

D. Go To Goal mais Avoid Obstacles com lógica fuzzy

Os resultados exibidos nesta subseção foram obtidos em outro trabalho prático da disciplina, pelo mesmo grupo do trabalho atual.

Quando simulado o algoritmo de alcançar um ponto objetivo com obstáculos no caminho, o robô evita os obstáculos e consegue alcançar o ponto objetivo. Como pode ser observado na Figura 12.

E. Go To Goal mais Avoid Obstacles com DQN

Durante os experimentos usando a técnica DQN, acompanhamos que nas primeiras rodadas o robô não conseguia avançar, porque apenas ficava girando em círculos. Entretanto, percebemos que aos poucos ele foi evoluindo e conseguindo desviar dos obstáculos e avançar para o ponto objetivo.

Na Figura 13, é exibido o resultado da primeira rodada usando a técnica DQN. Perceba que o ângulo do robô está na direção oposta do objetivo. Isso aconteceu porque na primeira rodada o robô ficou se movimentando em círculos, muito próximo do ponto inicial.

Com cerca de 5 rodadas o robô aprendeu a avançar em direção ao objetivo, desviando dos obstáculos durante o percurso. Na Figura 14, mostramos o robô quase chegando no objetivo em uma iteração da quinta rodada.

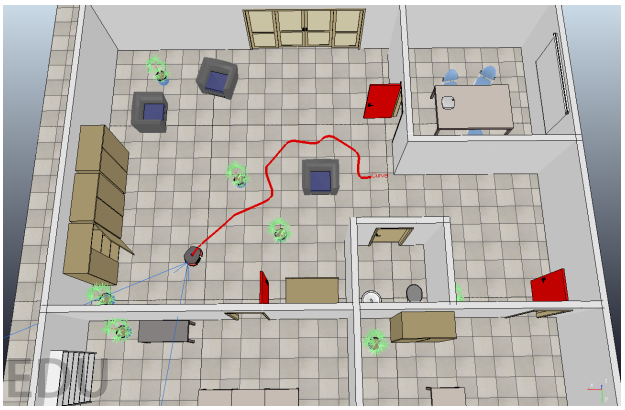

Figura 12: Execução do método ir até o objetivo junto com o método de evitar obstáculos com lógica *fuzzy*.

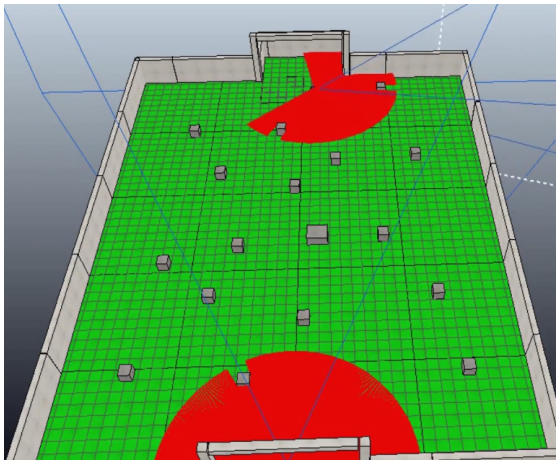

Figura 13: Primeira rodada com DQN.

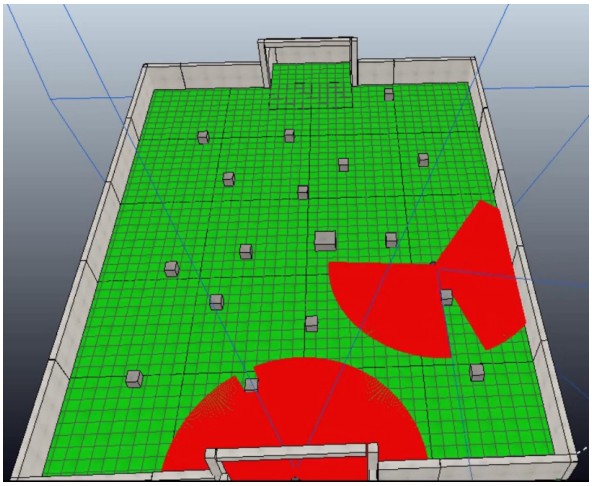

Figura 14: Quinta rodada com DQN.

VI. Conclusões

Um vídeo com a execução da simulação de diversos testes está disponível em https://youtu.be/Q6ETvQM3fdI.

Pelo vídeo e pelas análises da Seção V, é possível observar os resultados encontrados neste projeto. Foi empiricamente comprovado que o controle do robô usando os métodos de aprendizagem por reforço se mostraram mais eficientes – considerando o tempo levado para chegar ao objetivo – do que aqueles usando lógica fuzzy, principalmente quando utilizamos o método Q-Learning.

Além disso, essa maior eficiência e precisão nos movimentos permitiram o uso de velocidades mais elevadas quando o robô era controlado pelo método Q-Learning, pois dificilmente o robô colide com obstáculos ou encontra dificuldades em alcançar o objetivo (como não conseguir acertar o ângulo que aponta para o objetivo).

Referências

[1] R. Siegwart and I. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, 1st ed. Cambridge, Massachusetts: MIT Press, 2004. 1

[2] B. Wakileh and K. Gill, "Use of fuzzy logic in robotics," *Computers in Industry*, vol. 10, no. 1, pp. 35 – 46, 1988. 1

[3] H. Vashisth and Peng-Yung Woo, "Application of fuzzy logic to robotic control," in *Proceedings of the 1996 IEEE IECON. 22nd International Conference on Industrial Electronics, Control, and Instrumentation*, vol. 3, 1996, pp. 1867–1872 vol.3. 1

[4] C.-H. Chen, C.-C. Wang, Y. Wang, and P. Wang, "Fuzzy logic controller design for intelligent robots," *Mathematical Problems in Engineering*, vol. 2017, pp. 1–12, 09 2017. 1

[5] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, 09 2013. 1

[6] C. Gaskett, "Q-learning for robot control," 01 2002. 1

[7] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, "Benchmarking reinforcement learning algorithms on real-world robots," in *CoRL*, 2018. 1

[8] L. A. Zadeh, "Fuzzy logic," *Computer*, vol. 21, no. 4, pp. 83–93, 1988. 4

[9] "A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python," https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, acesso: 13-11-2019. 6