



UNIVERSIDADE FEDERAL DO PIAUÍ
CENTRO DE TECNOLOGIA
BACHARELADO EM ENGENHARIA ELÉTRICA
DISCIPLINA: INTELIGÊNCIA COMPUTACIONAL APLICADA
PROF. DR. JOSÉ MARIA PIRES DE MENEZES JÚNIOR
DISCENTE: ALINE CRISLAINY BARBOSA DA SILVA SOUSA
MATRÍCULA: 20229004604

Atividade 3 - Relatório Implementação de um Jogo da Velha em Python

Resumo - Este relatório apresenta a implementação de um jogo da velha em Python. O texto está organizado em três tópicos principais: a jogabilidade entre dois jogadores humanos, a interação entre um jogador humano e o “computador”, e, por fim, a implementação de uma interface gráfica para facilitar a interação com o usuário.

1. Jogador vs. Jogador

O código foi estruturado utilizando a programação orientada a objetos. Definiu-se uma classe do tipo Tabuleiro, que representa o objeto principal do jogo e contém métodos que auxiliam na verificação das jogadas durante a execução do código. Na fig.1 tem-se as bibliotecas utilizadas e a dimensão do tabuleiro.

```
#Bibliotecas utilizadas
import pygame
import sys
from PIL import Image

#dimensões do tabuleiro: 3x3
BOARD_SIZE = 3
```

Fig. 1 - Bibliotecas utilizadas e dimensão do tabuleiro.

O método `dentro_limites()` verifica se a jogada está dentro dos limites do tabuleiro. Caso os valores de `x` e `y` sejam menores que zero, o método retorna as coordenadas corrigidas como `(0, 0)`. Esse método foi implementado anteriormente, antes da criação de uma interface gráfica, sendo utilizado para tratar os dados enviados pelo usuário via terminal. Em seguida, o método `jogar()` é responsável por executar a jogada propriamente dita. Nele, verifica-se se o ponto enviado pelo jogador está vazio e dentro dos limites definidos para o tabuleiro. Caso essas condições sejam atendidas, o ponto é marcado com um “X” ou um “O” e o método retorna `True`. Caso contrário, retorna `False`.

O próximo método trata-se da verificação de vitória. O método `verificar_vitoria()` recebe como parâmetro uma variável `jogador`, que pode assumir os caracteres “X” ou “O”. Em seguida, são realizadas verificações em cada linha, coluna e diagonal do tabuleiro, buscando identificar alguma combinação que representa uma vitória para o jogador especificado. Caso seja encontrada uma combinação vencedora, o método retorna `True`; caso contrário, retorna `False`. Por fim, foi implementado o método `cheio()`, responsável por verificar se todas as posições do tabuleiro já foram ocupadas sem que tenha ocorrido uma vitória. Esse método é utilizado para declarar empate entre os jogadores. A implementação descrita é apresentada na fig. 2.

```

#na classe Tabuleiro estão as funções de jogar, verificar vitória e verificar se o tabuleiro está cheio,
# bem como a definição do tamanho do tabuleiro
class Tabuleiro:
    def __init__(self):
        #define o objeto tabuleiro como uma matriz 3x3 vazia
        self.tabuleiro = [["" for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]
    #verifica se a jogada está dentro dos limites do tabuleiro
    def dentro_limites(self, x, y):
        return 0 <= x < BOARD_SIZE and 0 <= y < BOARD_SIZE
    #método para verificar se a jogada está dentro dos limites do tabuleiro, se estiver é o local
    #estiver vazio, a jogada é realizada
    def jogar(self, jogador, x, y):
        ...
        Realiza a jogada do jogador na posição (x, y) se estiver dentro dos limites e a posição estiver vazia
        Parâmetros de Entrada: símbolo do jogador ("X" ou "O"), coordenadas x e y
        Saída: True se a jogada foi realizada, False caso contrário
        ...
        if self.dentro_limites(x, y) and self.tabuleiro[x][y] == "":
            self.tabuleiro[x][y] = jogador
            return True
        return False
    #método para verificar se um jogador venceu. O verifica linha a linha, coluna a coluna e as diagonais se estão
    #preenchidas pelo determinado símbolo do jogador
    def verificar_vitoria(self, jogador):
        ...
        Verifica se o jogador venceu, linha a linha, coluna a coluna e diagonais
        Entrada: símbolo do jogador ("X" ou "O")
        Saída: True se o jogador venceu, False caso contrário
        ...
        # linhas
        for i in range(BOARD_SIZE):
            if all(self.tabuleiro[i][j] == jogador for j in range(BOARD_SIZE)):
                return True
        # colunas
        for j in range(BOARD_SIZE):
            if all(self.tabuleiro[i][j] == jogador for i in range(BOARD_SIZE)):
                return True
        # diagonal principal
        if all(self.tabuleiro[i][i] == jogador for i in range(BOARD_SIZE)):
            return True
        # diagonal secundária
        if all(self.tabuleiro[i][BOARD_SIZE-1-i] == jogador for i in range(BOARD_SIZE)):
            return True
        return False

    def cheio(self):
        ...
        Verifica se o tabuleiro está cheio
        Saída: True se o tabuleiro está cheio, False caso contrário
        ...
        tab_cheio = all(self.tabuleiro[i][j] != "" for i in range(BOARD_SIZE) for j in range(BOARD_SIZE))
        return tab_cheio

```

Fig. 2 - Classe Tabuleiro.

No loop principal do código, define-se a sequência de execução da aplicação. A primeira parte da estrutura refere-se à implementação de uma interface gráfica, para cada evento do *pygame* – ou seja, uma entrada do usuário – verifica-se primeiramente se o botão de sair da interface foi pressionado. Caso essa condição seja verdadeira, a interface é encerrada. Em seguida, verifica-se se houve algum clique na tela e se ainda não há um vencedor (`vencedor == False`). Se essas condições forem atendidas, a posição do clique é capturada por meio da função `posicao_do_clique()`.

Por fim, chama-se o método `jogar()` e verifica-se se a jogada foi válida (retorno `True`). Caso a verificação seja válida, inicia-se o bloco condicional no qual o método `verificar_vitoria()` é utilizado para identificar possíveis combinações de vitória para o jogador atual. Caso não haja vitória, o método `cheio()` é chamado para verificar se o tabuleiro está completamente preenchido. Se essa condição for verdadeira, o jogo é encerrado com um empate. Se nenhuma dessas condições for satisfeita, o próximo jogador é então definido. A função `desenhar()`, presente nas duas versões apresentadas, é utilizada para atualizar e exibir a interface gráfica do jogo na tela. A implementação descrita é apresentada na fig. 3.

```

# Loop principal
while rodando:
    # eventos do pygame (entrada do usuário)
    for evento in pygame.event.get():# pega os eventos do pygame
        if evento.type == pygame.QUIT:# encerra o jogo se a janela for fechada
            rodando = False
            pygame.quit()
            sys.exit()
    #verifica se houve um clique do mouse e se não há vencedor ainda
    if evento.type == pygame.MOUSEBUTTONDOWN and not vencedor:
        x, y = evento.pos
        linha, coluna = posicao_do_clique(x, y)
        # verifica o jogador atual, o local do tabuleiro e realiza a jogada. Se a jogada for válida,
        # verifica se há um vencedor ou se o tabuleiro está cheio (empate). Alterna o jogador atual para
        # o próximo jogador
        if jogo.jogar(jogador_atual, linha, coluna):
            if jogo.verificar_vitoria(jogador_atual):
                vencedor = jogador_atual
            elif jogo.cheio():
                vencedor = "Empate"
            else:
                jogador_atual = "O" if jogador_atual == "X" else "X"
    #desenha o tabuleiro e as peças
    desenhar()

```

Fig. 3 - Loop principal para a implementação Jogador vs. Jogador.

2. Jogador vs. Computador

Na implementação do modo jogador versus computador, não houve alterações na classe `Tabuleiro`. Apenas foi adicionada uma nova classe para caracterizar o comportamento do computador, além de algumas modificações no loop principal do jogo, que serão apresentadas posteriormente.

Como pode ser observado no código na fig. 4, a classe `Computador` foi definida como filha da classe `Tabuleiro`. A implementação de herança foi necessária para reutilizar os métodos presentes na classe `Tabuleiro`, necessários na implementação da lógica de decisão do computador no jogo. Inicialmente, definiu-se o símbolo do computador como “O” e o do jogador humano como “X”. Em seguida, o método `verificar_possibilidades()` recebe como parâmetros o estado atual do tabuleiro e o jogador. Esse método define um vetor com todas as combinações possíveis de vitória e, com base no jogador informado, inicializa uma variável de contagem. Essa variável é responsável por contabilizar as possibilidades de vitória com determinada jogada. O método retorna o valor da variável contadora `v`. O método `verificar_possibilidades()` é utilizado dentro do método `melhor_jogada()`, responsável por determinar a jogada ideal do computador. A lógica implementada segue os princípios da árvore de decisão discutida em sala de aula. A escolha do computador baseia-se na diferença entre o número de vezes que ele pode vencer e o número de vezes que o jogador humano pode vencer, calculada através de simulações de jogadas.

O computador analisa as jogadas possíveis e toma decisões com base em prioridades. As prioridades seguem a seguinte sequência

- 1 - Computador vencer;
- 2 - Bloquear jogador humano;
- 3 - Escolher maior score.

```

# Classe que define um objeto Computadp. Nessa classe estão os métodos que definem como o computador deverá
# agir diante das jogadas do jogador humano
# Define como uma classe filha do tabuleiro, para que o computador seja capaz de utilizar o método verificar_vitoria,
# para ser capaz de simular as jogadas antes de realizá-las
class Computador(Tabuleiro):
    def __init__(self, simbolo="0"):
        self.simbolo = simbolo
        self.oponente = "X" # símbolo do jogador humano

    def verificar_possibilidades(self, tabuleiro, jogador):
        ...
        Verifica as possibilidades de vitória para o jogador
        Entrada: estado atual do tabuleiro, símbolo do jogador ("0" ou "X")
        Saída: número de possibilidades de vitória para o jogador'''
        #combinacoes de vitória possíveis
        combinacoes = [
            [(0,0), (0,1), (0,2)],
            [(1,0), (1,1), (1,2)],
            [(2,0), (2,1), (2,2)],
            [(0,0), (1,0), (2,0)],
            [(0,1), (1,1), (2,1)],
            [(0,2), (1,2), (2,2)],
            [(0,0), (1,1), (2,2)],
            [(0,2), (1,1), (2,0)]
        ]
        #verifica qual adversario
        adversario = "0" if jogador == "X" else "X"
        #inicializa o contador de possibilidades
        v = 0
        #verifica no tabuleiro as combinações possíveis de vitória e retorna o contador de possibilidades V
        for comb in combinacoes:
            marcas = [tabuleiro[i][j] for i, j in comb]
            if adversario not in marcas:
                v += marcas.count(jogador) # valoriza linhas boas

        return v

def melhor_jogada(self, tabuleiro):
    ''' Determina a melhor jogada para o computador com base na simulação das jogadas possíveis e
    na quantidade possível de vezes que ele pode ganhar
    Entrada: estado atual do tabuleiro
    Saída: coordenadas (x, y) da melhor jogada, priorizando primeiramente a vitória, depois o
    bloqueio do adversário e, por fim, a jogada com maior pontuação
    ...
    #inicializa variáveis para encontrar a melhor jogada, sua pontuação e a lista de probabilidades
    (variável) melhor_score: Literal[-999] qualquer jogada seja melhor, mesmo que ela seja menor que zero
    melhor_score = -999
    melhor_movimento = None
    jogada_vitoria = None
    jogada_bloqueio = None
    prob_xo = []

    # Percorre todas as posições do tabuleiro e analisa as possibilidades de vitória para ambos os jogadores
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tabuleiro.tabuleiro[i][j] == "":
                Vx = self.verificar_possibilidades(tabuleiro.tabuleiro, "X")
                Vo = self.verificar_possibilidades(tabuleiro.tabuleiro, "0")
                prob_xo.append((i, j, Vo - Vx))

    # Executa as simulações de jogadas para determinar a melhor jogada
    for i, j, score_atual in prob_xo:
        # simula a jogada do computador
        tabuleiro.tabuleiro[i][j] = self.simbolo
        venceu_comp = tabuleiro.verificar_vitoria(self.simbolo)
        tabuleiro.tabuleiro[i][j] = ""

        # simula a jogada do humano
        tabuleiro.tabuleiro[i][j] = "X"
        venceu_humano = tabuleiro.verificar_vitoria("X")
        tabuleiro.tabuleiro[i][j] = ""

        # Decide a melhor jogada com base nas simulações
        # guarda prioridade
        if venceu_comp:
            jogada_vitoria = (i, j)
        elif venceu_humano:
            jogada_bloqueio = (i, j)
        elif score_atual > melhor_score:
            melhor_score = score_atual
            melhor_movimento = (i, j)

    # decide na ordem de prioridade
    if jogada_vitoria:
        return jogada_vitoria # prioridade máxima
    elif jogada_bloqueio:
        return jogada_bloqueio
    else:
        return melhor_movimento

```

Fig 4 - Classe Computador.

Além disso, a estrutura do loop principal, apresentado na fig. 5, foi modificada com a adição de uma verificação para identificar quando se trata do jogador humano ("X"). Fora da estrutura `for`, é feita a verificação do computador e da variável `vencedor`. Nessa condição, o método `melhor_jogada()` da

classe Computador é chamado para que o computador selecione sua jogada. Em seguida, é executada uma estrutura de verificação semelhante àquela utilizada para o jogador humano, utilizando os métodos `cheio()` e `verificar_vitoria()` da classe Tabuleiro.

```
# Loop principal
while rodando:
    # eventos do pygame (entrada do usuário)
    for evento in pygame.event.get(): # pega os eventos do pygame
        if evento.type == pygame.QUIT: #encerra o jogo se a janela for fechada
            rodando = False
            pygame.quit()
            sys.exit()
    #verifica se houve um clique do mouse e se não há vencedor ainda
    if evento.type == pygame.MOUSEBUTTONDOWN and not vencedor:
        x, y = evento.pos
        linha, coluna = posicao_do_clique(x, y)

    # Jogada do humano (X)
    if jogador_atual == "X":
        if jogo.jogar("X", linha, coluna):
            if jogo.verificar_vitoria("X"):
                vencedor = "X"
            elif jogo.cheio():
                vencedor = "Empate"
            else:
                jogador_atual = "O"

    # Jogada do computador (O) se ele não tiver vencido ainda
    if jogador_atual == "O" and not vencedor:
        jogada = computador.melhor_jogada(jogo)
        if jogada:# se houver uma jogada possível
            i, j = jogada
            jogo.jogar("O", i, j)
            if jogo.verificar_vitoria("O"):
                vencedor = "O"
            elif jogo.cheio():
                vencedor = "Empate"
            else:
                jogador_atual = "X"
    #desenha o tabuleiro e as peças
    desenhar()
```

Fig. 5 - Loop principal para a implementação Jogador vs. Computador.

3. Interface Gráfica com bibliotecas Pygame e Pillow

Python oferece diversas bibliotecas que facilitam o desenvolvimento de aplicações visuais e interativas. Duas delas se destacam por suas funcionalidades complementares utilizadas na implementação do jogo da velha:

- *Pygame* é uma biblioteca baseada na SDL (Simple DirectMedia Layer) que permite criar jogos 2D de forma simples e eficiente. Com ela, é possível controlar gráficos, sons, eventos de teclado e mouse, além de gerenciar animações e colisões.
- *Pillow* é uma biblioteca de processamento de imagens que permite abrir, editar, salvar e transformar arquivos gráficos em diversos formatos.

A utilização do *Pygame* foi essencial para a criação da interface gráfica, implementação do tabuleiro, posicionamento dos símbolos de cada jogador na posição selecionada, bem como para definir parâmetros como o tamanho da tela. Já a biblioteca *Pillow* foi empregada na implementação de GIFs (imagens animadas) ao final do jogo, apresentando mensagens de vitória, empate ou derrota. A implementação da interface e o processamento dos GIFs em frames para exibição no *Pygame* é apresentado na fig. 7. O tabuleiro e as possíveis finalizações de partida estão ilustrados nas figs. 2 e 3.

```

# Interface implementada utilizando a biblioteca pygame e a biblioteca PIL para exibir gifs animados
# Inicialização do pygame e configuração da tela
pygame.init()
size=(600,600)
FORMAT = "RGBA"
LARGURA, ALTURA = 600, 600
TELA = pygame.display.set_mode((LARGURA, ALTURA))
pygame.display.set_caption("Jogo da Velha")

# Carregar imagens do tabuleiro do jogo e das peças X e O
tabuleiro_img = pygame.image.load("imagens/tabuleiro_jogo.png")
tabuleiro_img = pygame.transform.scale(tabuleiro_img, (LARGURA, ALTURA))

x_img = pygame.image.load("imagens/X.png")
x_img = pygame.transform.scale(x_img, (180, 180))

o_img = pygame.image.load("imagens/O.png")
o_img = pygame.transform.scale(o_img, (180, 180))

# Inicialização do jogo
# inicializa tabuleiro, computador, jogador atual, variável de controle do loop e vencedor
jogo = Tabuleiro()
computador = Computador(simbolo="O")
jogador_atual = "X"
rodando = True
vencedor = None

# Funções para converter imagens PIL para Pygame e para obter frames de gifs animados
def pil_to_game(img):
    """ Converte uma imagem PIL para um objeto de imagem Pygame
        Entrada: imagem PIL
        Saída: imagem Pygame"""
    data = img.tobytes("raw", FORMAT)
    return pygame.image.fromstring(data, img.size, FORMAT)

def get_gif_frame(img, frame):
    """ Obtém um frame específico de um gif animado
        Entrada: imagem PIL (gif), número do frame
        Saída: frame convertido para o formato RGBA"""
    img.seek(frame)
    return img.convert(FORMAT)

def init():
    ...
    Inicializa o Pygame e configura a tela
    Saída: objeto da tela do Pygame"""
    return pygame.display.set_mode(size)

def main(screen, path_to_image):
    """ Exibe um gif animado na tela do Pygame
        Entrada: objeto da tela do Pygame para o arquivo de imagem (gif)
        Saída: None"""
    gif_img = Image.open(path_to_image)
    if not gif_img.is_animated:
        print(f"Imagem em {path_to_image} não é um gif animado")
        return
    current_frame = 0
    clock = pygame.time.Clock()
    while True:
        frame = pil_to_game(get_gif_frame(gif_img, current_frame))
        frame = pygame.transform.scale(frame, size)

        screen.blit(frame, (0, 0))
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return

        current_frame = (current_frame + 1) % gif_img.n_frames
        pygame.display.flip()
        clock.tick(10)

```

```

def desenhar():
    """ Desenha o tabuleiro e as peças na tela do Pygame, e exibe mensagens de vitória ou empate junto com a animação correspondente
        Entrada: None
        Saída: None"""
    TELA.blit(tabuleiro_img, (0, 0))

    # desenha as peças no tabuleiro
    for j in range(BOARD_SIZE):
        for i in range(BOARD_SIZE):
            if jogo.tabuleiro[i][j] == "X":
                TELA.blit(x_img, (j * 200 + 10, i * 200 + 10))
            elif jogo.tabuleiro[i][j] == "O":
                TELA.blit(o_img, (j * 200 + 10, i * 200 + 10))

    # mensagem de vitória/empate/perdeu
    if vencedor: # o vencedor recebe o valor "X", "O" ou "Empate"
        if vencedor == "X":
            path = 'imagens/ganhouX.gif'
        elif vencedor == "O":
            path = 'imagens/xperdeu.gif'
        else:
            path = 'imagens/Empate.gif'

    main(TELA, path) # Exibe o gif correspondente ao resultado do jogo

    global rodando # controla o loop principal(global define a variável rodando como global)
    rodando = False
    pygame.time.delay(1000)
    pygame.display.update() #atualiza a tela

    # Função para determinar a posição do clique do mouse no tabuleiro
    def posicao_do_clique(x, y):
        """ Determina a posição (linha, coluna) do clique do mouse no tabuleiro
            Entrada: coordenadas x e y do clique
            Saída: tupla (linha, coluna)
            ...
        linha = y // 200
        coluna = x // 200
        return linha, coluna

```

Fig. 6 - Implementação de interface gráfica em Python utilizando Pygame e Pillow.

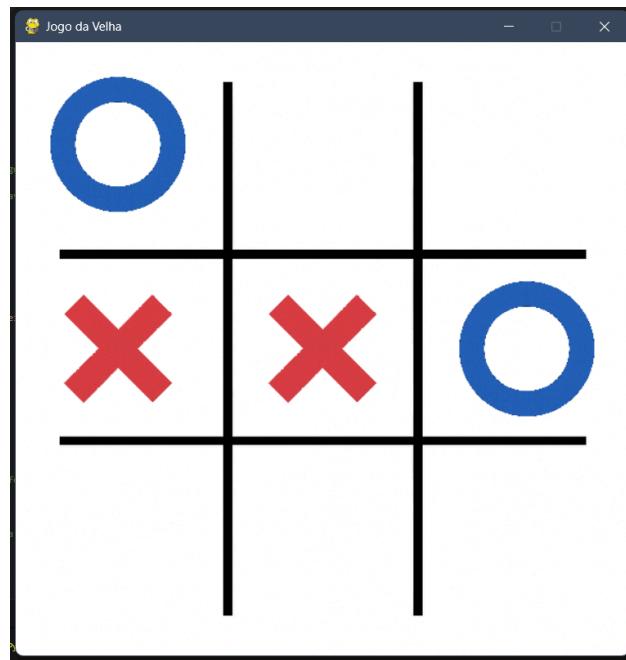


Fig. 7 - Tabuleiro apresentado na interface.



Fig. 8 - Frames das animações utilizadas na finalização de partidas da personagem Anya Forger(Anime “SPYxFamily”).