UNIVERSIDADE FEDERAL DO MARANHÃO
BACHARELADO EM CIÊNCIA E TECNOLOGIA
ENGENHARIA DA COMPUTAÇÃO
INTELIGÊNCIA ARTIFICIAL

Docente: Dr. Thales Levi Azevedo

LABIRINTO

Labirinto aplicado em python com a finalidade de implementar e entender algoritmo de busca de forma prática

Emanuel Lopes Silva - 2021017818
Letícia Delfino de Araújo - 2021061763
Thales Aymar Fontes - 2021018145

São Luís,MA
15/12/2024

Sumário

Metodologia..... 3
GerarMaze.py:..... 4
Breadth_First_Search.py:..... 5
agente.py:..... 7
main.py:..... 8

1 - Metodologia

O código foi dividido em 4 arquivos .py: GerarMaze.py para gerar um labirinto aleatório, que utiliza a técnica de geração de Recursive Backtracking ; Breadth_First_Search.py, para executar um algoritmo BFS em cima do labirinto criado com o objetivo de achar o caminho mais curto até a saída; agente.py, para fazer com que um agente percorra o caminho encontrado pelo BFS e deixe um rastro sobre ele; e um arquivo main.py, para executar os 3 arquivos anteriores em ordem.

2 - GerarMaze.py:

Primeiramente são definidas as configurações iniciais do labirinto e da janela que ele será exibido onde RES define o tamanho da janela,com WIDTH sendo a sua largura e HEIGHT sendo a sua altura, TILE é o tamanho de cada “unidade” do labirinto,e cols e rows o número de colunas e fileiras do labirinto, respectivamente.

Após isso, define-se a classe Célula e suas funções. Ela servirá para representar e controlar cada unidade no labirinto. Essa classe possui coordenadas x,y para indicar sua posição no labirinto, um dicionário booleano chamado walls, que mostra as quatro direções da célula (‘cima’, ‘esquerda’, ‘direita’,’baixo’) e se há uma parede nelas(True) ou não (False)., e também uma variável booleana visitado, para indicar se aquela célula já foi visitada na construção do labirinto. Suas funções são:

draw_current_cell: Desenha a célula atualmente em processamento

draw: Desenha as células já visitados, incluindo suas paredes

check_cells:

check_neighbors:Checa todas as células vizinhas não visitadas, coloca-as em uma lista e então retorna um vizinho aleatório dentro dessa lista.

Imagem 1 - Comandos principais do GerarMaze

```
1 usage  Emanuel
def draw_current_cell(self):
    x, y = self.x * TILE, self.y * TILE
    pygame.draw.rect(sc, pygame.Color('#067BC2'), rect: (x + 2, y + 2, TILE - 2, TILE - 2))

1 usage  Emanuel
def draw(self):
    x, y = self.x * TILE, self.y * TILE
    if self.visitado:
        col = (cwrap(self.x * 14), #coloque col na cor pra o fundo ficar doidão.
               cwrap(self.y * -20),
               cwrap(self.y * 30)) #caso queira usar o fundo cinza, pygame.Color('#1e1e1e')
        pygame.draw.rect(sc, pygame.Color('#3eb489'), rect: (x, y, TILE, TILE))
    if self.walls['cima']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), start_pos: (x, y), end_pos: (x + TILE, y), width: 3)
    if self.walls['esquerda']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), start_pos: (x, y + TILE), end_pos: (x, y), width: 3)
    if self.walls['direita']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), start_pos: (x + TILE, y), end_pos: (x + TILE, y + TILE), width: 3)
    if self.walls['baixo']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), start_pos: (x + TILE, y + TILE), end_pos: (x, y + TILE), width: 3)

4 usages  Emanuel
def check_cell(self, x, y):
    find_index = lambda x, y: x + y * cols
    if x < 0 or x > cols - 1 or y < 0 or y > rows - 1:  #comando geral pra checar as células
        return False
    return grid_cells[find_index(x, y)]
```

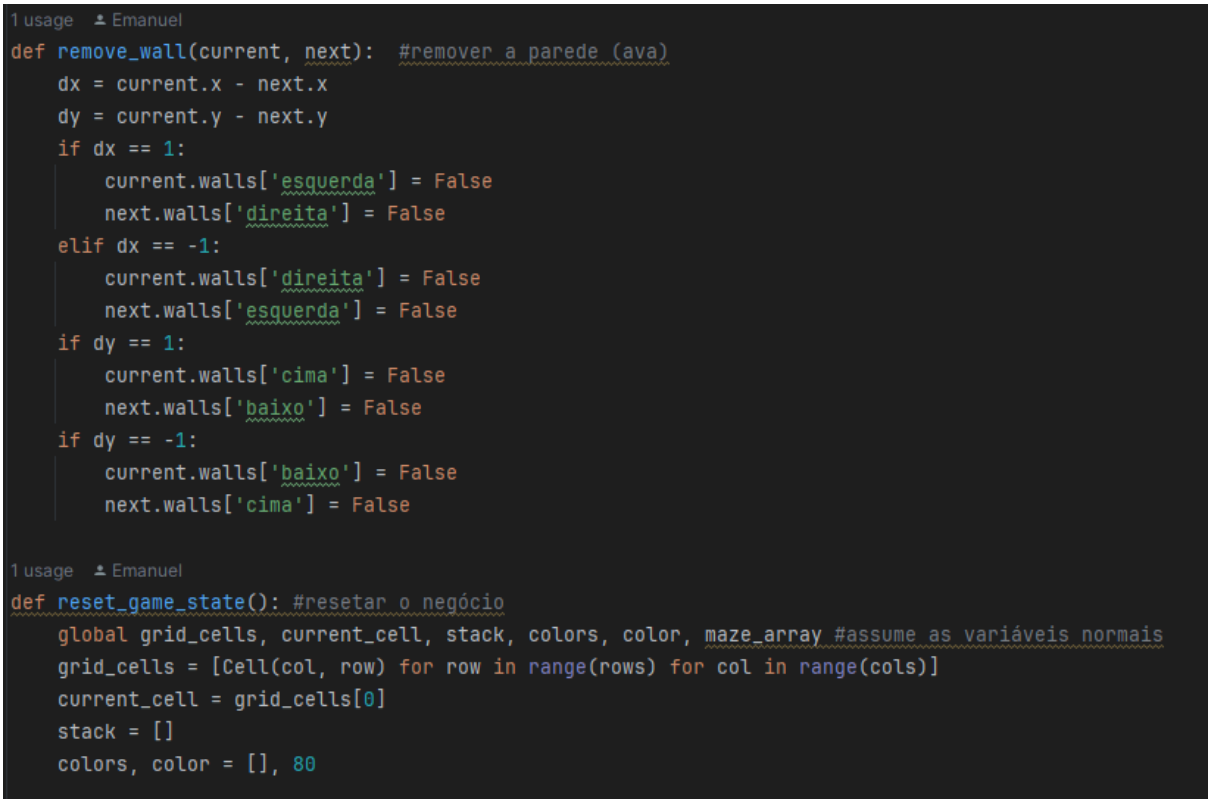
Autoria própria

Outras funções auxiliares:

remove_wall: Remove a parede entre duas células adjacentes.

reset_game_state: Reinicia o estado do jogo, incluindo a grid e a pilha do algoritmo.

Imagem 2 - Comandos auxiliares do GerarMaze



```
1 usage   Emanuel
def remove_wall(current, next):  #remover a parede (ava)
    dx = current.x - next.x
    dy = current.y - next.y
    if dx == 1:
        current.walls['esquerda'] = False
        next.walls['direita'] = False
    elif dx == -1:
        current.walls['direita'] = False
        next.walls['esquerda'] = False
    if dy == 1:
        current.walls['cima'] = False
        next.walls['baixo'] = False
    if dy == -1:
        current.walls['baixo'] = False
        next.walls['cima'] = False

1 usage   Emanuel
def reset_game_state(): #resetar o negócio
    global grid_cells, current_cell, stack, colors, color, maze_array #assume as variáveis normais
    grid_cells = [Cell(col, row) for row in range(rows) for col in range(cols)]
    current_cell = grid_cells[0]
    stack = []
    colors, color = [], 80
```

Autoria própria

grid_cells = [Cell(col, row) for row in range(rows) for col in range(cols)]: Comando de List Comprehension que permite a criação uma grid 2d, que é onde o labirinto é realizado. Basicamente, cria uma lista chamada grid_cells que contém todas as células do labirinto. Cada célula é uma instância da classe Cell, representando uma posição no grid.

current_cell = grid_cells[0] : Define current_cell como a primeira célula do grid

colors, color = [], 80

stack = [] : Permite as cores durante o labirinto ser criado. Basicamente para enfeitar um pouco.

O bloco if __name__ == "__main__" é o ponto de entrada principal do programa e define o comportamento do labirinto enquanto ele está sendo gerado. Ele utiliza um loop principal que controla a execução contínua até que o labirinto esteja completo. Nesse loop, a tela é preenchida com a cor de fundo e os eventos do Pygame, como fechar a janela ou pressionar a barra de espaço, são tratados. O labirinto é gerado dinamicamente utilizando a célula inicial (current_cell) e um algoritmo de busca em profundidade com backtracking, que explora vizinhos não visitados e retorna ao caminho anterior caso não haja mais vizinhos disponíveis. Durante o processo, as células visitadas são desenhadas na tela com cores diferenciadas, e as informações do labirinto (como coordenadas e paredes) são atualizadas em tempo real. Quando o labirinto é concluído, ele é salvo como uma imagem

(labirinto.png) e também em formato JSON (walls_data.json), encerrando o programa de forma ordenada.

Existe a opção mencionada no read.me no github de criar-se um labirinto completamente cromático, que foi feito para dar estilos diferentes ao labirinto gerado. Para isso, deve-se colocar no comando Draw a cor base “col” dentro do “self.visitado”.

3 - Breadth_First_Search.py:

O código inicia definindo o ponto de partida(start) como sendo a coordenada (0,0) e um ponto de chegada(goal) de coordenada aleatória, e então carrega o arquivo walls_data.json gerado pelo GerarMaze.py, que contém as posições das células do labirinto.

Ademais, o trecho do código “if os.file.isfile” tem como objetivo verificar se o arquivo walls_data.json existe e, caso contrário, o programa solicita que o usuário inicie o script GerarMaze para garantir que os dados necessários sejam criados.

get_neighbors: Identifica os vizinhos acessíveis da célula atual com base nas paredes e retorna uma lista com as coordenadas desses vizinhos.

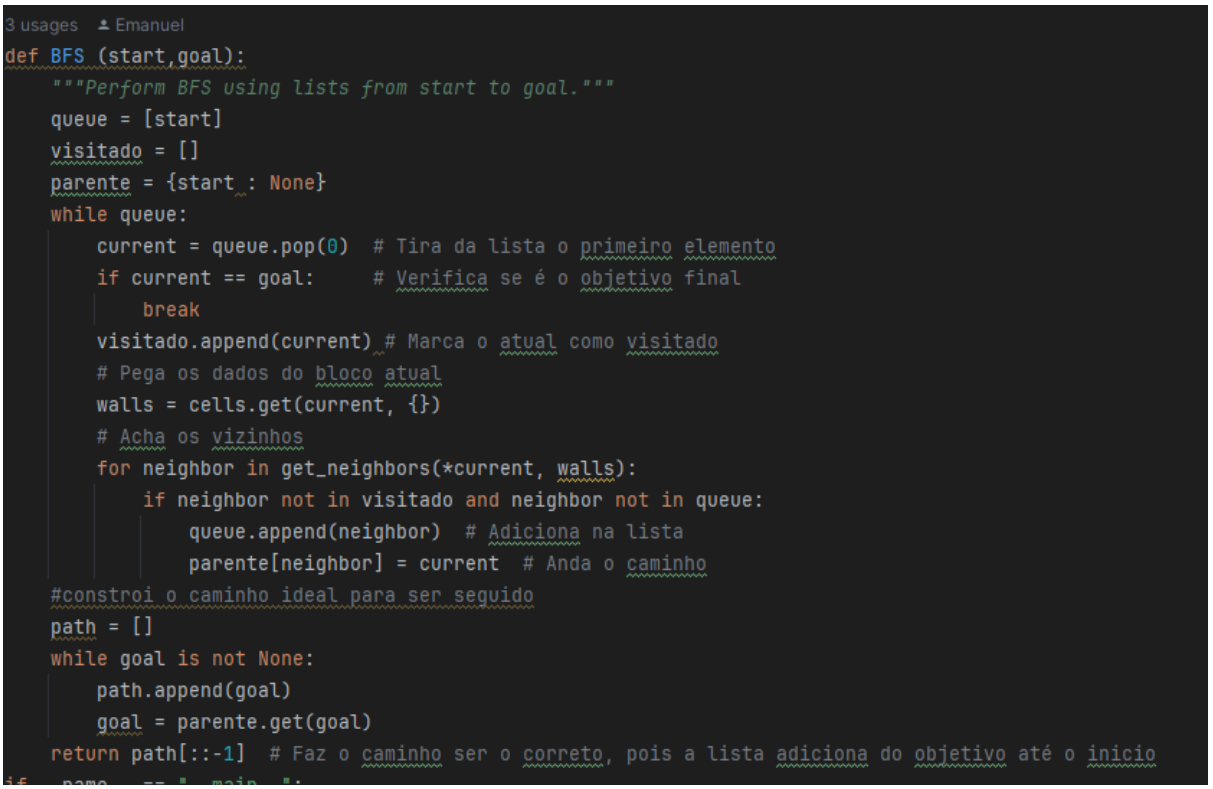
Imagem 3 -Código de obtenção de dados do JSON e Verificação de Vizinhos

```
# carrega o arquivo
with open('walls_data.json', 'r') as file:
    maze = json.load(file)

# organizar as celulas
cells = {(cell['x'], cell['y']): cell['walls'] for cell in maze}
1 usage  Emanuel *
def get_neighbors(x, y, walls):
    directions = {
        "cima": (x, y - 1),
        "baixo": (x, y + 1), #basicamnete as direções apresentadas.
        "esquerda": (x - 1, y),
        "direita": (x + 1, y),
    }
    neighbors = []
    for wall, (nx, ny) in directions.items():
        if not walls.get(wall, True):  # Sem uma parede, dá pra passar. É o que é possível andar.
            neighbors.append((nx, ny))
    return neighbors
```

Autoria própria

BFS: Função que roda o algoritmo BFS e percorre todas as células do labirinto até achar a saída. Inicialmente cria-se uma lista queue para conter as coordenadas das células a serem visitadas e adiciona-se apenas as coordenadas do ponto de partida (start). Após isso, a função retira o primeiro elemento da lista (elemento de índice 0) e visita a célula com as coordenadas correspondentes. Feito isso, a célula é marcada como visitada e checa-se quais dos seus vizinhos estão disponíveis, ou seja, quais ainda não foram visitados e nem foram adicionados na queue, e então adiciona-os na queue, marcando a célula atual como sendo a célula pai deles. Essa sequência de ações é repetida até se chegar ao objetivo, ou seja, até a célula com coordenadas goal ser atingida.

Imagem 4 - Código do BFS



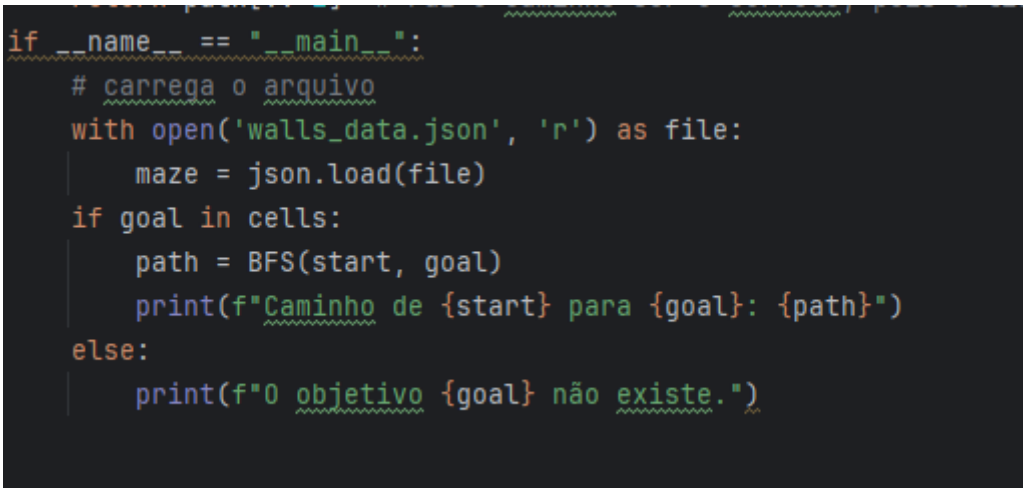
```
3 usages  Emanuel
def BFS (start,goal):
    """Perform BFS using lists from start to goal."""
    queue = [start]
    visitado = []
    parente = {start : None}
    while queue:
        current = queue.pop(0)  # Tira da lista o primeiro elemento
        if current == goal:     # Verifica se é o objetivo final
            break
        visitado.append(current) # Marca o atual como visitado
        # Pega os dados do bloco atual
        walls = cells.get(current, {})
        # Acha os vizinhos
        for neighbor in get_neighbors(*current, walls):
            if neighbor not in visitado and neighbor not in queue:
                queue.append(neighbor)  # Adiciona na lista
                parente[neighbor] = current  # Anda o caminho
    #constroi o caminho ideal para ser seguido
    path = []
    while goal is not None:
        path.append(goal)
        goal = parente.get(goal)
    return path[::-1]  # Faz o caminho ser o correto, pois a lista adiciona do objetivo até o inicio
```

Autoria própria

Em seguida, é criada uma lista path para armazenar o menor caminho do ponto de partida até o ponto de chegada. Para isso, adiciona-se as coordenadas do ponto de chegada a lista e define-se o novo ponto de chegada como sendo o pai do ponto de partida atual. Isso é feito até chegar na célula de ponto de partida, que não tem pai. Então, a função termina retornando a lista path em ordem inversa, que é exatamente o caminho ideal do labirinto.

Depois de definidas as funções, é criado um bloco de código para executá-las.

Imagem 5 - Código de execução do BFS



```
if __name__ == "__main__":
    # carrega o arquivo
    with open('walls_data.json', 'r') as file:
        maze = json.load(file)
    if goal in cells:
        path = BFS(start, goal)
        print(f"Caminho de {start} para {goal}: {path}")
    else:
        print(f"O objetivo {goal} não existe.")
```

Autoria própria

4 - **agente.py:**

No início do código, bibliotecas essenciais como pygame, json e time são importadas. As constantes definem o tamanho dos blocos ,como a TILE SIZE, que teve o nome modificado para não repetir completamente a GerarMaze.Py , as cores do agente, caminho, grade e fundo, além da taxa de atualização do FPS. Essas configurações ajudam a manter o layout visual uniforme e configurável. Os dados são importados do arquivo GerarMaze para que se mantenha padronizado e que não precise refazer mudanças desnecessárias em ambos os documentos.

A função load_maze() carrega o labirinto salvo no arquivo walls_data.json. Esse arquivo é gerado pelo código de geração do labirinto (GerarMaze.py). A função utiliza o módulo json para carregar os dados e transformá-los em um objeto Python (uma lista de células). Essa função permite que o labirinto comece a ser visualizado pelo usuário.

A função desenhar_grid_and_agent() é responsável por desenhar a grade do labirinto, com as paredes de cada célula baseadas nas informações do JSON, além do caminho percorrido pelo agente até o momento. Como diz o nome, também é responsável por desenhar o próprio agente, que se move ao longo do caminho.

Imagem 6 - Código de desenhar o Agente e a GRID

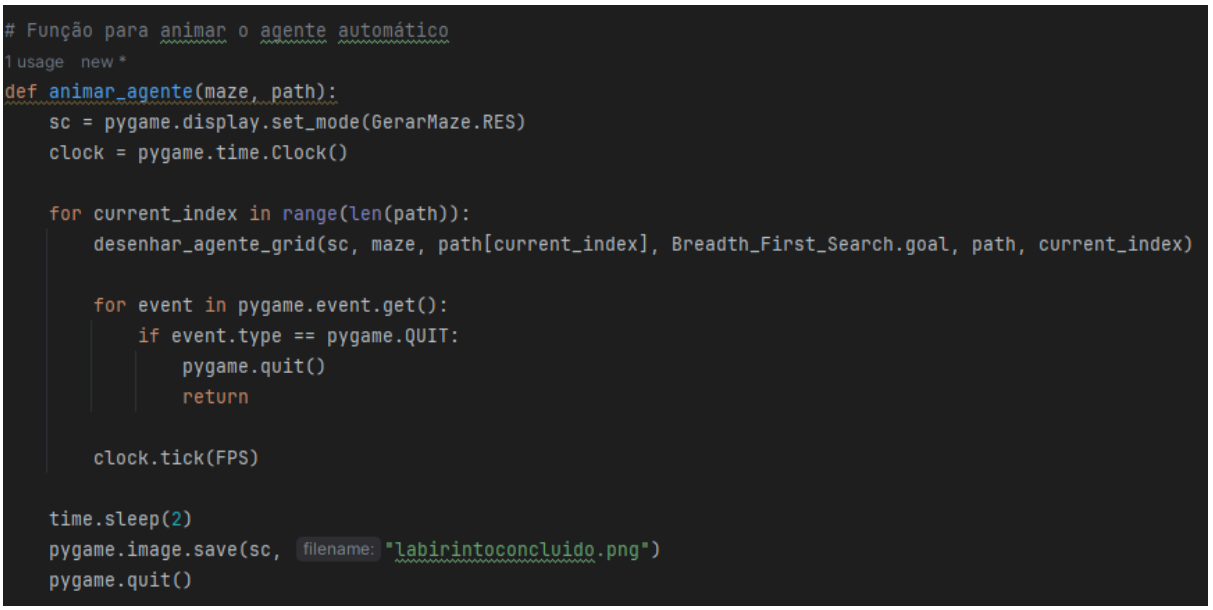
```python
# desenhaa o labirinto e o agente
2 usages  new *
def desenhar_agente_grid(screen, maze, agent_pos, goal_pos, path=None, current_index=0):
    screen.fill(BACKGROUND_COLOR)  # Limpa a tela
    for cell in maze:
        x, y = cell['x'], cell['y']
        walls = cell['walls']
        rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE)
        pygame.draw.rect(screen, GRID_COLOR, rect)  # Desenha a grid

        # Desenha as paredes
        if walls['cima']:
            pygame.draw.line(screen, pygame.Color('#1e4f5b'), start_pos: (x * TILE_SIZE, y * TILE_SIZE), end_pos: ((x + 1) * TILE_SIZE, y * TILE_SIZE), width: 2)
        if walls['baixo']:
            pygame.draw.line(screen, pygame.Color('#1e4f5b'), start_pos: (x * TILE_SIZE, (y + 1) * TILE_SIZE), end_pos: ((x + 1) * TILE_SIZE, (y + 1) * TILE_SIZE), width: 2)
        if walls['esquerda']:
            pygame.draw.line(screen, pygame.Color('#1e4f5b'), start_pos: (x * TILE_SIZE, y * TILE_SIZE), end_pos: (x * TILE_SIZE, (y + 1) * TILE_SIZE), width: 2)
        if walls['direita']:
            pygame.draw.line(screen, pygame.Color('#1e4f5b'), start_pos: ((x + 1) * TILE_SIZE, y * TILE_SIZE), end_pos: ((x + 1) * TILE_SIZE, (y + 1) * TILE_SIZE), width: 2)

    # Mostra o caminho percorrido para o agente automático
    if path:
        for idx in range(current_index):
            x, y = path[idx]
            rect = pygame.Rect(x * TILE_SIZE + 10, y * TILE_SIZE + 10, TILE_SIZE - 20, TILE_SIZE - 20)
            pygame.draw.rect(screen, pygame.Color('#3D348B'), rect, border_radius=20)

    # Desenha o agente
    agent_draw_pos = (agent_pos[0] * TILE_SIZE + TILE_SIZE // 2, agent_pos[1] * TILE_SIZE + TILE_SIZE // 2)
    pygame.draw.circle(screen, AGENT_COLOR, agent_draw_pos, TILE_SIZE // 4)

    # Desenha o objetivo
    gx, gy = goal_pos
    goal_rect = pygame.Rect(gx * TILE_SIZE + 5, gy * TILE_SIZE + 5, TILE_SIZE - 10, TILE_SIZE - 10)
    pygame.draw.rect(screen, pygame.Color('#F9627D'), goal_rect, border_radius=10)

    pygame.display.flip()  # Atualiza a tela
```

Autoria própria

Para cada célula no labirinto , as paredes são desenhadas conforme o estado de walls (cima, baixo, esquerda, direita), e o caminho percorrido é destacado em verde, que são as células visitadas pelo agente. O agente, representado por um círculo vermelho, é desenhado na posição atual definida por current_index, que costumeiramente é o ponto 0,0.

A função animate_agent() é o núcleo da animação. Ela desenha o movimento do agente ao longo do caminho (path), célula por célula. Inicialmente, ela carrega o tamanho da tela com base no número de células do labirinto e configura o Pygame para exibir a interface gráfica. Dentro de um loop, a função percorre cada posição do caminho (path) e utiliza a função draw_grid_and_agent para desenhar a grade, as paredes do labirinto, o caminho percorrido até o momento e o agente na posição atual. A animação é controlada pela taxa de quadros definida pelo clock.tick(FPS), garantindo suavidade no movimento.

Imagem 7 - Código de movimentar o agente automaticamente



```python
# Função para animar o agente automático
1 usage  new *
def animar_agente(maze, path):
    sc = pygame.display.set_mode(GerarMaze.RES)
    clock = pygame.time.Clock()

    for current_index in range(len(path)):
        desenhar_agente_grid(sc, maze, path[current_index], Breadth_First_Search.goal, path, current_index)

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return

        clock.tick(FPS)

    time.sleep(2)
    pygame.image.save(sc, filename: "labirintoconcluido.png")
    pygame.quit()
```

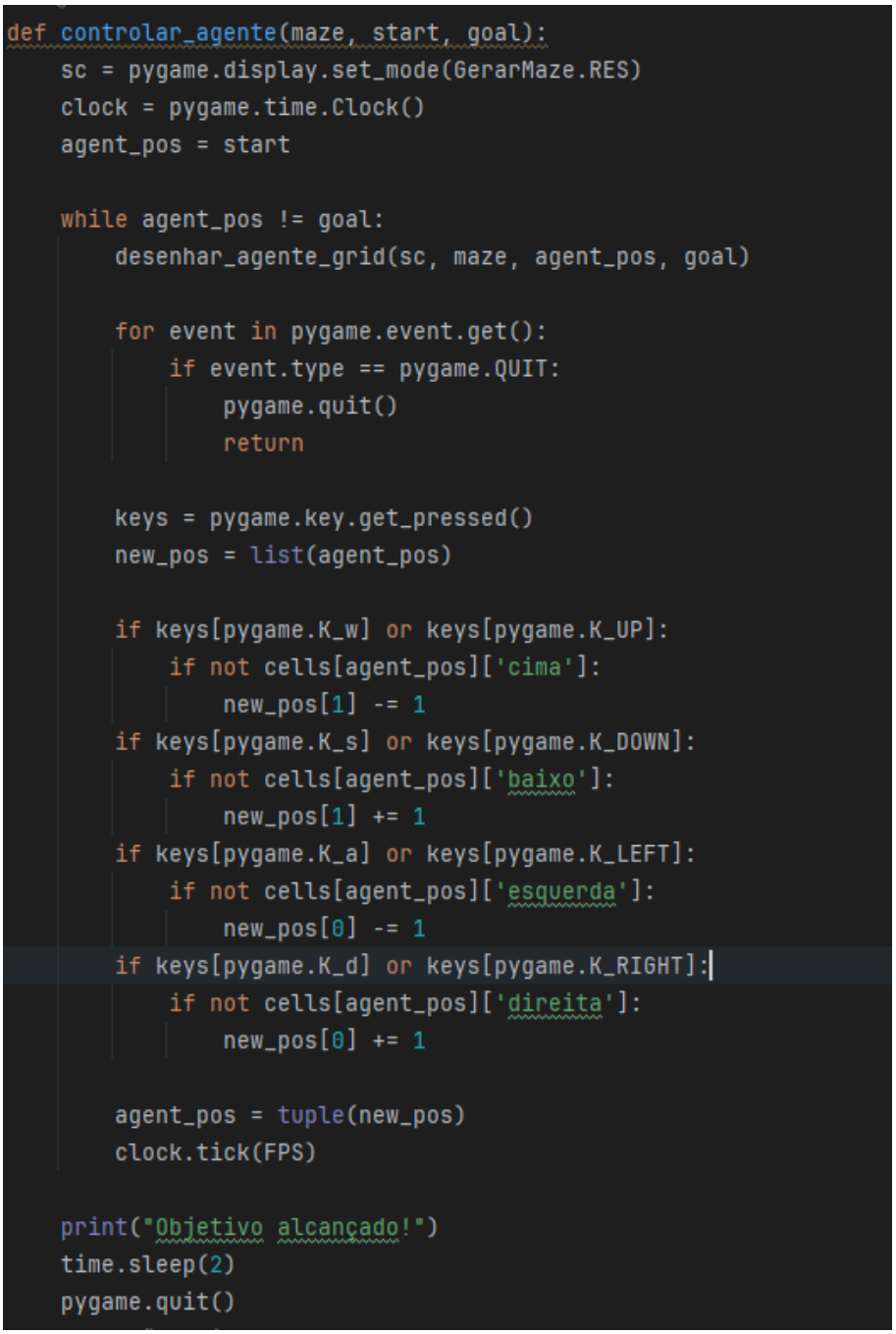
Autoria própria

Ademais, a função trata eventos como o fechamento da janela, permitindo que o programa seja encerrado adequadamente. Após o agente concluir o caminho, a função pausa por poucos segundos e encerra o Pygame, finalizando a animação.

Foi colocado para facilitar a visualização um comando de exportar a imagem do labirinto concluído. Isso não tem função real, mas serve para treinar a resolução do labirinto. Também serve para comparar se o labirinto que foi gerado é o correto. Pois em certas implementações, os dados presentes no JSON eram de labirintos antigos, mesmo o novo sendo gerado.

Após isso, houve a adição de um agente que poderia ser controlado manualmente, para uma maior diversidade de usos do programa

Imagem 8 - Código de movimentar o Agente manualmente



```
def controlar_agente(maze, start, goal):
    sc = pygame.display.set_mode(GerarMaze.RES)
    clock = pygame.time.Clock()
    agent_pos = start

    while agent_pos != goal:
        desenhar_agente_grid(sc, maze, agent_pos, goal)

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return

        keys = pygame.key.get_pressed()
        new_pos = list(agent_pos)

        if keys[pygame.K_w] or keys[pygame.K_UP]:
            if not cells[agent_pos]['cima']:
                new_pos[1] -= 1
        if keys[pygame.K_s] or keys[pygame.K_DOWN]:
            if not cells[agent_pos]['baixo']:
                new_pos[1] += 1
        if keys[pygame.K_a] or keys[pygame.K_LEFT]:
            if not cells[agent_pos]['esquerda']:
                new_pos[0] -= 1
        if keys[pygame.K_d] or keys[pygame.K_RIGHT]:
            if not cells[agent_pos]['direita']:
                new_pos[0] += 1

        agent_pos = tuple(new_pos)
        clock.tick(FPS)

    print("Objetivo alcançado!")
    time.sleep(2)
    pygame.quit()
```

Autoria própria

A função controlar_agente permite que o usuário controle manualmente um agente no labirinto com o teclado, movendo-o do ponto inicial até o objetivo. A cada iteração, a função desenha o labirinto e o agente na tela utilizando a função desenhar_agente_grid. Em seguida, utilizando-se do pygame, se captura os inputs diretos do teclado do usuário, para indicar para onde o agente vai, ao pressionar as setas ou o movimento clássico do WASD. Antes de atualizar a posição do agente , verifica se não há uma parede na direção desejada usando as informações de paredes armazenadas em cells[agent_pos]. Se o movimento for permitido, a posição é atualizada. O loop continua até que o agente alcance o objetivo, momento em que uma mensagem é exibida, e o programa aguarda brevemente antes de encerrar o Pygame.

Essa função proporciona uma interação direta com o labirinto, permitindo explorar seu layout manualmente. Ela foi adicionada como um extra, pois o usuário pode querer explorar o labirinto gerado.

5 - main.py:

O main apresentado no código é responsável por orquestrar a execução de diferentes etapas do programa, que incluem a geração do labirinto, o cálculo de um caminho usando busca em largura (BFS) e a animação de caminhada do agente. Basicamente, o Main roda os três arquivos anteriores em sequência, tornando a experiência mais fluida para o usuário. Ele é dividido em duas funções principais (main1 e main2), chamadas em sequência dentro do bloco principal if __name__ == "__main__".

O programa utiliza subprocessos advindos do módulo Subprocess para executar os scripts Python externos ao Main, mas ainda na mesma pasta (GerarMaze.py, Breadth_First_Search.py e agente.py) de maneira sequencial e integrada. Além disso, ele realiza a exibição de uma mensagem animada de "espera" enquanto as etapas estão em execução.

A função main1 é responsável por iniciar o processo de geração do labirinto. Ela exibe uma mensagem para o usuário utilizando a cor vermelha, indicando que o processo foi iniciado. Em seguida, utiliza o módulo subprocess para executar o script GerarMaze.py usando um interpretador Python localizado na pasta que originalmente foi criado usando uma venv. Isso garante que o labirinto será gerado corretamente com todas as dependências configuradas. A main1 é fundamental para isolar a responsabilidade de gerar o labirinto, deixando o restante do programa modular e limpo. Isso também permite que o resultado gerado pelos labirintos, como o JSON e as imagens, sejam propriamente geradas e depois carregadas pelos outros comandos.

A função main2 orquestra as etapas de cálculo do caminho no labirinto e a animação do agente. Primeiramente, ela executa o script Breadth_First_Search.py utilizando subprocess.run(), o que calcula o caminho entre a célula inicial e a célula meta no labirinto. Em seguida, após uma pequena pausa de 1 segundo , executa o script agente.py, que anima o movimento do agente ao longo do caminho calculado.

Entretanto, é bom informar que como o Objetivo do agente é randomizado para dar mais diversidade ao labirinto, o texto de saída ao rodar o BFS e o Agente são localizações diferentes.

A função waiting_dots tem como objetivo fornecer um feedback visual ao usuário enquanto o programa realiza operações demoradas, como a geração do labirinto. Ela exibe uma mensagem inicial seguida de uma animação de pontos, criando a ilusão de progresso. Foi utilizada apenas pois gera um bom indicativo visual de progresso, e deixa o uso do programa mais organizado.

Na função Main, temos a opção de adicionar qual modo usar o labirinto, no automático ou no manual, guiando o agente através de setas.

O código da função main está representado abaixo:

Imagem 9 - Código de movimentar o Agente manualmente

Autoria própria

6 - Referências:

Wilson, David Bruce (May 22–24, 1996). "Generating random spanning trees more quickly than the cover time". *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. Symposium on Theory of Computing. Philadelphia: ACM. pp. 296–303. CiteSeerX 10.1.1.47.8598. doi:10.1145/237814.237880. ISBN 0-89791-785-5.

Hendrawan, Yonathan. (2020). Comparison of Hand Follower and Dead-End Filler Algorithm in Solving Perfect Mazes. Journal of Physics: Conference Series. 1569. 022059. 10.1088/1742-6596/1569/2/022059.

Indriyono, Bonifacius & Widyatmoko,. (2021). Optimization of Breadth-First Search Algorithm for Path Solutions in Mazyin Games. International Journal of Artificial Intelligence & Robotics (IJAIR). 3. 58-66. 10.25139/ijair.v3i2.4256.