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

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 , as principais que estão representadas no Algoritmo 1, são:

draw_current_cell: Desenha a célula atualmente em processamento

draw: Desenha as células já visitados, incluindo suas paredes

check_cells: É usada para verificar se uma célula com coordenadas (x, y) está dentro dos limites válidos do grid e, se estiver, retornar essa célula. Caso contrário, retorna False, indicando que a célula escolhida não deve ser utilizada, pois está fora dos limites estabelecidos no WIDTH e HEIGHT precedentemente.

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

Algoritmo 1 : Comandos Principais do GerarMaze

```
1.  def draw_current_cell(self):
2.      x, y = self.x * TILE, self.y * TILE
3.      pygame.draw.rect(sc, pygame.Color('#067BC2'), (x + 2, y + 2, TILE - 2,
    TILE - 2))
4.
5.  def draw(self):
6.      x, y = self.x * TILE, self.y * TILE
7.      if self.visitado:
8.          col = (cwrap(self.x * 14),
9.                 cwrap(self.y * -20),
10.                cwrap(self.y * 30)) #caso queira usar o fundo cinza,
    pygame.Color('#1e1e1e')
11.         pygame.draw.rect(sc, pygame.Color('#3eb489'), (x, y, TILE, TILE))
```

```
    if self.walls['cima']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), (x, y), (x + TILE, y), 3)
    if self.walls['esquerda']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), (x, y + TILE), (x, y), 3)
    if self.walls['direita']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), (x + TILE, y), (x + TILE,
    y + TILE), 3)
    if self.walls['baixo']:
        pygame.draw.line(sc, pygame.Color('#1e4f5b'), (x + TILE, y + TILE), (x,
    y + TILE), 3)

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)]

def check_neighbors(self):
    neighbors = []
    cima = self.check_cell(self.x, self.y - 1)
    esquerda = self.check_cell(self.x - 1, self.y)
    direita = self.check_cell(self.x + 1, self.y)
    baixo = self.check_cell(self.x, self.y + 1)
    if cima and not cima.visitado:
        neighbors.append(cima)  #visita o de cima
    if esquerda and not esquerda.visitado:
        neighbors.append(esquerda)  #visita o da esquerda
    if direita and not direita.visitado:
        neighbors.append(direita)  #visita o da direita
    if baixo and not baixo.visitado:
        neighbors.append(baixo)  #visita o de cima
    return random.choice(neighbors) if neighbors else False
```

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.

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.

Algoritmo 2 : Comandos Auxiliares do GerarMaze

```
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

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


grid_cells = [Cell(col, row)
              for row in range(rows)
              for col in range(cols)]
current_cell = grid_cells[0]
colors, color = [], 80
stack = []
correndo = True
```

O bloco if __name__ == "__main__" é o ponto de entrada principal do programa, representado no Algoritmo 3, 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”.

Algoritmo 3 : Código do Bloco Principal do GerarMaze

```
if __name__ == "__main__":
  while correndo:
     sc.fill(pygame.Color('#D56062'))
     for event in pygame.event.get():
        if event.type == pygame.QUIT:  #fecha se eu apertar o x
           exit()
        elif event.type == pygame.KEYUP and event.key == pygame.K_SPACE: #resetar a
criação do mapa
           reset_game_state()
     [cell.draw() for cell in grid_cells]
     current_cell.visitado = True
     current_cell.draw_current_cell()
     [pygame.draw.rect(sc, colors[i],
                  (cell.x * TILE + 2, cell.y * TILE + 2,
                   TILE , TILE )) for i,
      cell in enumerate(stack)]
     # Descomente caso for usar Cinza!
     #eu tinha feito usando o cwrap mas ficou colorido demais, ai n sei se devo fazer daquele
jeito
     next_cell = current_cell.check_neighbors()
     if next_cell:
        next_cell.visited = True
        stack.append(current_cell)
        colors.append((cwrap(current_cell.x * 32), cwrap(current_cell.y * -12),
cwrap(current_cell.y * 42))) #isso dai serve pra pintar coloridinho os quadrados que passam
na criação
```

```
    #comente acima caso for usar o gradiente do fundo
    remove_wall(current_cell, next_cell)
    current_cell = next_cell
elif stack:
    current_cell = stack.pop()
else:
    # Quando o labirinto for concluído, saia do loop
    time.sleep(2)
    print("Labirinto concluído!")
    pygame.image.save(sc, "labirinto.png")
    correndo = False
    pygame.quit()
    exit()
pygame.display.flip()
clock.tick(100)
# exportar a imagem
Maze_array = [{'x': cell.x, 'y': cell.y, 'walls': cell.walls} for cell in grid_cells]
print(Maze_array)
file_path = 'walls_data.json'
# Salvar a Imagem
with open(file_path, 'w') as json_file:
    json.dump(Maze_array, json_file)
```

---

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.

Estes códigos estão representados no Algoritmo 4:

Algoritmo 4 : Código de obtenção de dados do JSON e Verificação de Vizinhos

---

```
1. if not os.path.isfile('walls_data.json'):
2.     print("O arquivo 'walls_data.json' não foi encontrado. Gerando agora...")
3.     time.sleep(3)
4.     # Executa o script GerarMaze.py para criar o arquivo
5.     subprocess.run([python_executable,"GerarMaze.py"])
```

```
    print('\035[31m' + "Rode o Gerar.Maze e feche! Ele precisa criar os arquivos necessários para o main rodar." + '\035[0m')

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

# organizar as células
cells = {(cell['x'], cell['y']): cell['walls'] for cell in maze}
def get_neighbors(x, y, walls):
    """Return accessible neighbors based on walls.""" #Chat gpt tá falando barras.
    directions = {
        "cima": (x, y - 1),
        "baixo": (x, y + 1), #basicamente 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
```

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.

Tal procedimento pode ser visto em formato de código claramente no Algoritmo 5:

Algoritmo 5 : Código do BFS

```
def BFS (start,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
```

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, que está representado no Algoritmo 6:

Algoritmo 6: Código do Bloco Principal 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}: {goal}")
    else:
        print(f"O objetivo {goal} não existe.")
```

---

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.

Algoritmo 7 : Código de desenhar o Agente e a GRID

```
1. def desenhar_agente_grid(screen, maze, agent_pos, goal_pos, path=None,
   current_index=0):
2.     screen.fill(BACKGROUND_COLOR)  # Limpa a tela
3.     for cell in maze:
4.         x, y = cell['x'], cell['y']
5.         walls = cell['walls']
6.         rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,
   TILE_SIZE)
7.         pygame.draw.rect(screen, GRID_COLOR, rect)  # Desenha a grid
8.
9.         # Desenha as paredes
10.        if walls['cima']:
11.            pygame.draw.line(screen, pygame.Color('#1e4f5b'), (x * TILE_SIZE, y
   * TILE_SIZE), ((x + 1) * TILE_SIZE, y * TILE_SIZE), 2)
12.        if walls['baixo']:
13.            pygame.draw.line(screen, pygame.Color('#1e4f5b'), (x * TILE_SIZE,
   (y + 1) * TILE_SIZE), ((x + 1) * TILE_SIZE, (y + 1) * TILE_SIZE), 2)
14.        if walls['esquerda']:
15.            pygame.draw.line(screen, pygame.Color('#1e4f5b'), (x * TILE_SIZE, y
   * TILE_SIZE), (x * TILE_SIZE, (y + 1) * TILE_SIZE), 2)
16.        if walls['direita']:
```

```
            pygame.draw.line(screen, pygame.Color('#1e4f5b'), ((x + 1) *
    TILE_SIZE, y * TILE_SIZE), ((x + 1) * TILE_SIZE, (y + 1) * TILE_SIZE),
    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
```

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.

Algoritmo 8 : Código de Movimentar o Agente Automaticamente

```
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, "labirintoconcluido.png")
    pygame.quit()
```

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

Algoritmo 9 : 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:
```

```
6.      desenhar_agente_grid(sc, maze, agent_pos, goal)
7.
8.      for event in pygame.event.get():
9.          if event.type == pygame.QUIT:
10.             pygame.quit()
11.             return
12.
13.     keys = pygame.key.get_pressed()
14.     new_pos = list(agent_pos)
15.
16.     if keys[pygame.K_w] or keys[pygame.K_UP]:
17.         if not cells[agent_pos]['cima']:
18.             new_pos[1] -= 1
19.     if keys[pygame.K_s] or keys[pygame.K_DOWN]:
20.         if not cells[agent_pos]['baixo']:
21.             new_pos[1] += 1
22.     if keys[pygame.K_a] or keys[pygame.K_LEFT]:
23.         if not cells[agent_pos]['esquerda']:
24.             new_pos[0] -= 1
25.     if keys[pygame.K_d] or keys[pygame.K_RIGHT]:
26.         if not cells[agent_pos]['direita']:
27.             new_pos[0] += 1
28.
29.     agent_pos = tuple(new_pos)
30.     clock.tick(FPS)
31.
32. print("Objetivo alcançado!")
33. time.sleep(2)
34. pygame.quit()
```

A função controlar_agente, representada no Algirtmo 9, 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.

Por fim, é criado um código permitindo ao usuário escolher entre dois modos para o agente do labirinto: modo automático (BFS) ou modo manual, que é controlado por teclas. Ele apresenta um menu com as opções, onde o usuário pode selecionar o modo digitando 1 ou 2. Em ambos os casos, as posições inicial e final são carregadas a partir do script de BFS. Se o usuário escolher o modo automático, a busca em largura é realizada para encontrar o caminho, que será animado na interface. No modo manual, o programa chama a função controlar_agente, permitindo ao usuário mover o agente pelo labirinto com o teclado até alcançar o objetivo.

Algoritmo 10 : Código do Bloco Principal do Agente

```
1.  if __name__ == "__main__":
2.      print("Escolha o modo do agente:")
3.      print("1 - Modo Automático (BFS)")
4.      print("2 - Modo Manual (Controle com Teclas)")
5.
6.      choice = input("Escolha 1 ou 2: ")
7.
8.      start = Breadth_First_Search.start
9.      goal = Breadth_First_Search.goal
10.
11.     if choice == '1':
12.         path = Breadth_First_Search.BFS(start, goal)
13.         if path:
14.             print(f"Caminho de {start} para {goal}: {path}")
15.             animar_agente(maze, path)
16.         else:
17.             print(f"O objetivo {goal} não existe.")
18.     elif choice == '2':
19.         controlar_agente(maze, start, goal)
20.     else:
21.         print("Escolha inválida.")
```

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 um ambiente virtual. 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.

Algoritmo 11 : Código das Funções usadas na Main

```
1. def mensagembase(n):
2.     print('\b' * n + ' ' * n + '\b' * n, end='', flush=True)
3. def waiting_dots(wait, ndots=4, interval=0.5, message="Gerando o Labirinto",
   final_message=""):
4.     start = time.time()  # Obtém o tempo atual
5.     print(message, end="", flush=True)  # Imprime a mensagem inicial
6.     while time.time() - start < wait:  # Loop enquanto o tempo decorrido for menor que
   o tempo de espera
7.         for _ in range(ndots):  # Loop para cada ponto a ser exibido
8.             print('.', end='', flush=True)  # Imprime um ponto
9.             time.sleep(interval)  # Aguarda um intervalo de tempo
10.        mensagembase(ndots)  # Limpa os pontos
11.        time.sleep(interval)  # Aguarda um intervalo de tempo
12.    if final_message is not None:  # Se houver uma mensagem final especificada
13.        mensagembase(len(message))  # Limpa a mensagem inicial
14.        print(final_message)  # Imprime a mensagem final
15. def main1():
16.    python_executable = sys.executable
17.    subprocess.run([python_executable, "GerarMaze.py"])
18. def main2():
19.    python_executable = sys.executable
20.    subprocess.run([python_executable, "Breadth_First_Search.py"])
21.    time.sleep(1)
22.    subprocess.run([python_executable, "agente.py"])
```

O código da função main está representado abaixo, no Algoritmo 12:

Algoritmo 12 : Código da Main

```
1. if __name__ == “__main__”:
2.
3.          if not os.path.isfile(‘walls_data.json’):
4.                  main1()
5.
6.
7.          else:
8.                  main1()
9.                  time.sleep(3)
10.                 # carrega o arquivo
11.                 with open(‘walls_data.json’, ‘r’) as file:
12.                         maze = json.load(file)
13.                 main2()
```

---

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.