Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Kayky Cristof Eduardo Domingos Silva" - "8118"

"Projeto 1 de inteligência artificial SIN-323"

Trabalho apresentado para obtenção de créditos na disciplina SIN-323 Inteligência Artificial da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pela Professora Larissa Ferreira Rodrigues Moreira.

Rio Paranaíba - MG

2024

1 RESUMO

Na primeira parte da disciplina de Inteligência Artificial (SIN-323), exploramos a Inteligência Artificial clássica, com um foco nos algoritmos de busca clássicos. Esses algoritmos são essenciais para resolver problemas onde um agente precisa encontrar uma solução dentro de um espaço de estados, realizando uma sequência de movimentos ou ações para alcançar o objetivo desejado.

Neste projeto, implementamos algoritmos de busca para resolver o problema de navegação em um labirinto. O objetivo foi aplicar os conceitos de **busca em profundidade**, **busca em largura** e **busca A***, utilizando a distância euclidiana e manhattan como heurística.

A seguir, detalharemos as metodologias de busca adotadas, as características dos algoritmos e como a implementação foi organizada para resolver o problema proposto. Além disso, discutiremos os resultados obtidos nas execuções dos algoritmos.

Sumário

1 RESUMO **1**

2 Tecnologias Utilizadas **3**

2.1 Escolha da Linguagem: Python 3

2.2 Biblioteca Pygame 3

3 Estrutura do problema **4**

3.1 Estados do problema 4

4 Criação do mapa **6**

4.1 Carregando a matriz 6

4.2 Criação do labirinto 6

5 Algoritmos de busca **9**

5.1 Busca em profundidade 9

5.2 Busca em largura 10

5.3 Busca gulosa 11

5.3.1 Distância Manhattan 12

5.3.2 Distância euclidiana 12

6 CONCLUSÃO **15**

6.1 Comparação entre os Algoritmos de Busca 15

6.2 Considerações finais 15

2 Tecnologias Utilizadas

Neste projeto, diversas tecnologias foram utilizadas para implementar algoritmos de busca clássicos e resolver o problema do labirinto. As principais ferramentas foram a linguagem de programação Python e a biblioteca Pygame.

2.1 Escolha da Linguagem: Python

A linguagem Python foi escolhida para o desenvolvimento deste projeto devido à sua facilidade na implementação de algoritmos de Inteligência Artificial. Entre os motivos para essa escolha, se destaca a simplicidade e clareza de sua sintaxe, o que facilita o processo de programação. Além disso, Python conta com uma vasta gama de bibliotecas específicas para Inteligência Artificial, que permitem a construção e otimização de algoritmos de busca de maneira eficiente. Por essas razões, Python se revela a escolha ideal para o desenvolvimento de soluções em Inteligência Artificial.

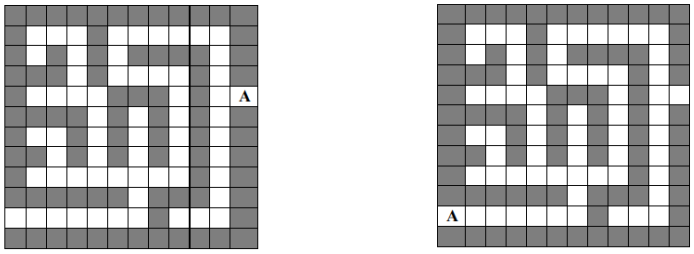
2.2 Biblioteca Pygame

A Pygame é uma biblioteca em Python voltada para o desenvolvimento de jogos e aplicações multimídia. Ela oferece funcionalidades como manipulação de imagens, controle de eventos e áudio, além de recursos gráficos essenciais para a criação de interfaces interativas.

No projeto, ela foi utilizada para a representação gráfica do labirinto. Ela permitiu a visualização dinâmica do processo de busca, exibindo o labirinto, os passos dos algoritmos e a posição do agente em tempo real.

3 Estrutura do problema

O labirinto foi representado como uma matriz, onde cada célula é um espaço do ambiente. A matriz possui o tamanho 12x12 e contém valores binários: um valor de 1 indica uma célula livre (onde o agente pode passar), enquanto o valor 0 indica uma célula bloqueada (onde o agente não pode passar).



(a) Estado Inicial (b) Estado Final

Figura 1: Labirinto com seu estado inicial e estado final

```
0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 1 1 1 1 1 1 0
0 1 0 1 0 1 0 0 0 0 1 0
0 0 0 1 0 1 1 1 1 0 1 0
0 1 1 1 1 0 0 0 1 0 1 1
0 0 0 0 1 0 1 0 1 0 1 0
0 1 1 0 1 0 1 0 1 0 1 0
0 0 1 0 1 0 1 0 1 0 1 0
0 1 1 1 1 1 1 1 1 0 1 0
0 0 0 0 0 0 1 0 0 0 1 0
1 1 1 1 1 1 1 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0
```

Figura 2: Matriz representando o mapa

3.1 Estados do problema

1. **Estado Inicial**: O estado inicial representa a posição de partida do agente no labirinto. No caso deste projeto, essa posição inicial foi definida na célula (4, 11) da matriz.

2. **Sucessor**: O sucessor representa todos os possíveis estados que podem ser alcançados a partir do estado atual. O agente pode se mover para células adjacentes (cima, baixo, esquerda e direita). Cada sucessor é um novo estado no qual o agente pode se encontrar após realizar um movimento.

3. **Teste de Objetivo**: O teste de objetivo verifica se o agente atingiu a posição desejada no labirinto, definida como a célula (10, 0). Quando o agente chega a essa célula, o teste de objetivo é bem-sucedido e o algoritmo de busca encerra a execução.

4. **Custo do Caminho**: O custo do caminho é a medida associada ao esforço necessário para ir de um estado ao outro. Cada movimento entre células adjacentes possui um custo fixo, definido como 1. O custo total do caminho, portanto, é dado pela soma dos movimentos realizados.

4 Criação do mapa

4.1 Carregando a matriz

O primeiro passo nesse projeto, foi construir a matriz de forma manual num arquivo.txt, definimos uma função que lê este arquivo, transforma os dados do txt em números inteiros e a salva em uma matriz python, para ser utilizada no projeto.

```python
def carregar_matriz():
    matriz = []
    try:
        with open("C:/Projeto/data/input/matriz.txt", 'r') as arquivo:
            for linha in arquivo:
                matriz.append([int(x) for x in linha.split()])
        return matriz
    except FileNotFoundError:
        print("Erro: O arquivo 'matriz.txt' não foi encontrado.")
        return None
```

Figura 3: Função que carrega a matriz para dentro do código:matriz.py

4.2 Criação do labirinto

Primeiramente definimos os componentes básicos para construção do labirinto.

```python
# Configurações iniciais
LARGURA_JANELA = 600
ALTURA_JANELA = 600
TAMANHO_CELULA = 50

# Cores
COR_PAREDE = (0, 0, 0)       # Preto para paredes
COR_CAMINHO = (255, 255, 255) # Branco para caminhos livres
COR_INICIO = (0, 255, 0)      # Verde para o ponto inicial
COR_DESTINO = (255, 0, 0)     # Vermelho para o destino
COR_VISITADO = (173, 216, 230) # Azul claro para células visitadas
COR_AGENT = (0, 0, 255)       # Azul para a posição atual do agente

# Posição inicial e destino
posicao_inicial = (4, 11)
posicao_destino = (10, 0)
```

Figura 4: Definição de componentes básicos do labirinto:config.py

Após definir os componentes básicos, inicializamos o Pygame, e utilizamos os componentes para definir os elementos do mapa, sendo estes: o agente, as paredes do labirinto, o caminho no qual o agente irá percorrer, o ponto de início, e o ponto de chegada.

```python
def inicializar_pygame():
    pygame.init()
    tela = pygame.display.set_mode((LARGURA_JANELA, ALTURA_JANELA))
    pygame.display.set_caption("Labirinto - Menu de Busca")
    return tela
```

Figura 5: Função que inicializa o Pygame:labirinto.py

A função inicializar_pygame é responsável pela configuração inicial da biblioteca Pygame, preparando o ambiente gráfico necessário para a execução do projeto. Agora iremos explicar passo a passo, como foi feita a construção do labirinto.

```python
def desenhar_labirinto(tela, matriz, visitados=set(), agente=None, passos=None, custos=None):
    fonte = pygame.font.SysFont("Arial", 16)

    for linha in range(len(matriz)):
        for coluna in range(len(matriz[linha])):
            x = coluna * TAMANHO_CELULA
            y = linha * TAMANHO_CELULA

            if matriz[linha][coluna] == 0:
                cor = COR_PAREDE
            elif (linha, coluna) in visitados:
                cor = COR_VISITADO
            else:
                cor = COR_CAMINHO

            pygame.draw.rect(tela, cor, (x, y, TAMANHO_CELULA, TAMANHO_CELULA))


            if custos and (linha, coluna) in custos:
                custo_texto = fonte.render(str(custos[(linha, coluna)]), True, (0, 0, 0))
                tela.blit(custo_texto, (x + TAMANHO_CELULA // 4, y + TAMANHO_CELULA // 4))
```

Figura 6: Parte 1 da função que cria o labirinto:labirinto.py

Esta função desenha cada célula da matriz que representa o labirinto, diferenciando visualmente as células que já foram visitadas, as paredes e os caminhos livres. Além disso, irá exibir os custos de cada célula visitada no algoritmo de busca gulosa.

A seguir, uma descrição detalhada dos elementos da função:

- **Parâmetros**:
  - `tela`: a superfície Pygame onde o labirinto será desenhado.
  - `matriz`: uma matriz que representa o layout do labirinto.
  - `visitados`: Iremos exibir por onde o agente passar(visitar) cada nó.
  - `custos` (somente no algorimo de busca gulosa): um dicionário que armazena o custo associado a cada célula do labirinto.
- **Loop Principal**: A função percorre todas as células da matriz, identificando se cada célula representa um caminho, uma parede ou uma célula já visitada. Cada tipo de célula é desenhado com uma cor específica:

  - Células de parede são desenhadas com a cor `COR_PAREDE`.
  - Células já visitadas são desenhadas com a cor `COR_VISITADO`.
  - Células que representam caminhos livres utilizam a cor `COR_CAMINHO`.

- **Exibição dos Custos**: Se a busca gulosa for executada a função renderiza o valor do custo na célula correspondente.

```
x_inicial, y_inicial = posicao_inicial[1] * TAMANHO_CELULA, posicao_inicial[0] * TAMANHO_CELULA
pygame.draw.rect(tela, COR_INICIO, (x_inicial, y_inicial, TAMANHO_CELULA, TAMANHO_CELULA))

x_destino, y_destino = posicao_destino[1] * TAMANHO_CELULA, posicao_destino[0] * TAMANHO_CELULA
pygame.draw.rect(tela, COR_DESTINO, (x_destino, y_destino, TAMANHO_CELULA, TAMANHO_CELULA))

if agente:
    x_agente, y_agente = agente[1] * TAMANHO_CELULA, agente[0] * TAMANHO_CELULA
    pygame.draw.rect(tela, COR_AGENT, (x_agente, y_agente, TAMANHO_CELULA, TAMANHO_CELULA))

fonte_passos = pygame.font.SysFont("Arial", 20)
texto_passos = fonte_passos.render(f"Passos: {passos}", True, (0, 0, 0))
tela.blit(texto_passos, (10, 10))
```

Figura 7: Parte 2 da função que cria o labirinto:labirinto.py

Esta parte do código desenha o ponto de início, o ponto de destino e a posição do agente no labirinto, além de um contador de passos para acompanhamento da busca gulosa.

Abaixo, uma explicação detalhada de cada elemento:

- **Ponto de Início**: A variável `posicao_inicial` contém as coordenadas iniciais do agente. Essas coordenadas são multiplicadas pelo tamanho da célula (`TAMANHO_CELULA`) para calcular a posição exata na tela em pixels. Em seguida, um retângulo é desenhado usando a cor `COR_INICIO` para representar visualmente o ponto inicial da busca.

- **Ponto de Destino**: De maneira semelhante, a variável `posicao_destino` armazena as coordenadas de destino no labirinto. Essas coordenadas também são convertidas para pixels e, então, um retângulo é desenhado na cor `COR_DESTINO` para marcar o objetivo da busca.

- **Agente**: Se a posição do agente (`agente`) for fornecida, a função calcula sua posição na tela em pixels e desenha um retângulo na cor `COR_AGENT`, representando onde o agente está no labirinto em tempo real.

- **Contador de Passos**: Exibe o contador de passos durante a execução da busca gulosa.

5 Algoritmos de busca

5.1 Busca em profundidade

O primeiro algoritmo no qual implementamos foi a busca em profundidade.

```python
def busca_profundidade(matriz, inicio, destino, tela, desenhar_labirinto):
    visitados = set()
    pilha = [inicio]
    passos = 0

    while pilha:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        no_atual = pilha.pop()

        if no_atual not in visitados:
            visitados.add(no_atual)
            passos += 1

        tela.fill((255, 255, 255))
        desenhar_labirinto(tela, matriz, visitados, agente=no_atual, passos=passos)
        pygame.display.flip()

        if no_atual == destino:
            print(f"Objetivo alcançado em {passos} passos!")
```

Figura 8: Função que implementa a busca em profundidade: buscas.py

A Busca em Profundidade é um algoritmo que explora o labirinto buscando caminhos até o destino, priorizando o aprofundamento em cada ramificação antes de retornar para explorar outros caminhos.

No início, são inicializadas as seguintes estruturas:

- Um conjunto para armazenar os nós já visitados.
- Uma pilha para organizar os nós pendentes de exploração. A pilha permite que o último nó adicionado seja o próximo a ser processado.
- Um contador para acompanhar o número de passos dados pelo agente durante a busca.

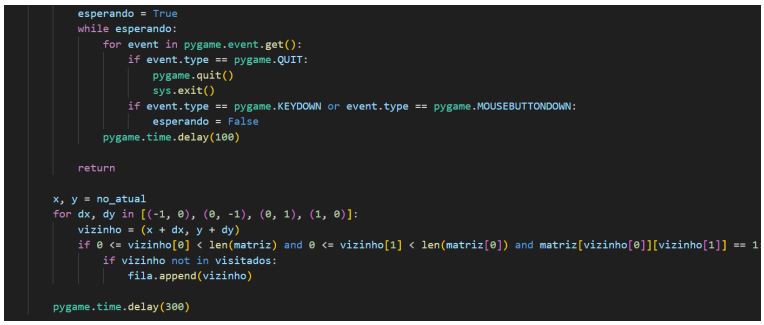
O algoritmo segue dentro de um laço principal, que continua enquanto houver elementos na pilha. A cada iteração, o nó no topo da pilha é retirado para processamento. Se o nó ainda não foi visitado, ele é marcado como visitado e o contador de passos é incrementado.

A interface gráfica é atualizada continuamente para refletir o estado atual da busca. A posição do agente, os passos realizados e os nós visitados são exibidos no labirinto em tempo real.

Caso o nó atual seja o ponto de destino, a busca é encerrada, e o programa exibe o número de passos necessários para alcançar o objetivo.

A busca em profundidade é eficiente para explorar labirintos pequenos ou quando não há necessidade de encontrar o caminho mais curto, já que ela não garante a solução mais otimizada.

No código abaixo, é implementado de forma igual em todos os códigos.



```
        esperando = True
        while esperando:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.KEYDOWN or event.type == pygame.MOUSEBUTTONDOWN:
                    esperando = False
            pygame.time.delay(100)

        return

    x, y = no_atual
    for dx, dy in [(-1, 0), (0, -1), (0, 1), (1, 0)]:
        vizinho = (x + dx, y + dy)
        if 0 <= vizinho[0] < len(matriz) and 0 <= vizinho[1] < len(matriz[0]) and matriz[vizinho[0]][vizinho[1]] == 1
            if vizinho not in visitados:
                fila.append(vizinho)

    pygame.time.delay(300)
```

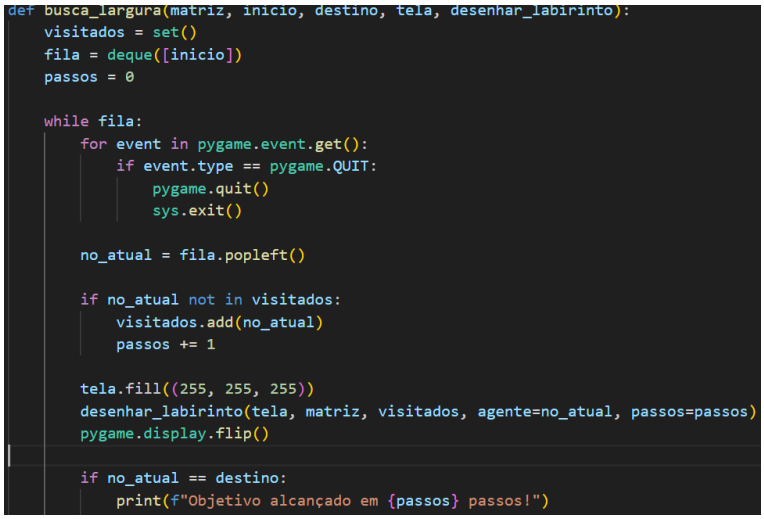
Figura 9: Parte em comum para todas as buscas: buscas.py

Uma parte do funcionamento dos algoritmos de busca (profundidade, largura e gulosa) é implementada de maneira idêntica, representando um bloco de lógica compartilhada entre eles. Este bloco realiza as seguintes funções:

- **Controle de interação com o usuário**: Após cada etapa do algoritmo, o programa aguarda uma interação do usuário, pressionando uma tecla.
- **Exploração de vizinhos**: Para cada nó visitado, o algoritmo verifica as células adjacentes em quatro direções (direita, baixo, cima, esquerda). Apenas os vizinhos válidos (que não são paredes e estão dentro dos limites da matriz) são considerados para os próximos passos. Este processo garante que o agente explore o labirinto corretamente.
- **Atraso visual (*delay*)**: Um pequeno atraso é adicionado entre as iterações para que o usuário consiga acompanhar visualmente o progresso da busca em tempo real.

5.2 Busca em largura

O segundo algoritmo que foi implementado foi a busca em largura.



```python
def busca_largura(matriz, inicio, destino, tela, desenhar_labirinto):
    visitados = set()
    fila = deque([inicio])
    passos = 0

    while fila:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        no_atual = fila.popleft()

        if no_atual not in visitados:
            visitados.add(no_atual)
            passos += 1

        tela.fill((255, 255, 255))
        desenhar_labirinto(tela, matriz, visitados, agente=no_atual, passos=passos)
        pygame.display.flip()

        if no_atual == destino:
            print(f"Objetivo alcançado em {passos} passos!")
```

Figura 10: Implementação da busca em largura: buscas.py

A Busca em Largura é um algoritmo que explora o labirinto de maneira sistemática, garantindo que todos os nós de um determinado nível sejam visitados antes de avançar para o próximo.

No início, são inicializadas as seguintes estruturas:

- Um conjunto para armazenar os nós já visitados, evitando repetições durante a busca.
- Uma fila para organizar os nós que ainda precisam ser explorados.
- Um contador para rastrear o número de passos realizados durante o processo.

A busca ocorre dentro de um laço principal, que continua enquanto houver elementos na fila. A cada iteração, o próximo nó é removido da fila para processamento. Se o nó ainda não foi visitado, ele é marcado como visitado e o contador de passos é incrementado.

A interface gráfica é atualizada continuamente para refletir o estado atual da busca, mostrando o agente, os passos realizados e os nós visitados no labirinto.

A busca em largura se destaca por garantir a solução mais curta em termos de número de passos, sendo especialmente eficiente para labirintos simples ou sem pesos.

5.3 Busca gulosa

A Busca Gulosa é um algoritmo que utiliza uma abordagem heurística para explorar o labirinto, priorizando os caminhos que aparentam estar mais próximos do destino. Abaixo

iremos abordar e explicar cada heurística utilizada.

5.3.1 Distância Manhattan

```python
def heuristica_manhattan(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

Figura 11: Implementação da distância manhattan: buscas.py

A heurística de Manhattan é uma técnica utilizada para estimar a distância entre dois pontos em um espaço baseado em uma grade. Ela calcula a soma das diferenças absolutas entre as coordenadas horizontais e verticais dos dois pontos.

$$\text{Distância Manhattan} = |x_1 - x_2| + |y_1 - y_2|$$

5.3.2 Distância euclidiana

```python
def heuristica_euclidiana(a, b):
    return sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)
```

Figura 12: Implementação da distância euclidiana: buscas.py

A heurística Euclidiana é utilizada para calcular a menor distância entre dois pontos em um plano bidimensional, considerando uma linha reta como trajeto. Ela é baseada na fórmula da distância Euclidiana.

$$\text{Distância Euclidiana} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Abaixo, segue a implementação da busca gulosa:

```python
def busca_gulosa(matriz, inicio, destino, tela, desenhar_labirinto, heuristica):
    visitados = set()
    fila_prioridade = []
    heapq.heappush(fila_prioridade, (heuristica(inicio, destino), inicio))
    custos = {inicio: heuristica(inicio, destino)}
    passos = 0

    while fila_prioridade:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        _, no_atual = heapq.heappop(fila_prioridade)

        if no_atual in visitados:
            continue

        visitados.add(no_atual)
        passos += 1

        tela.fill((255, 255, 255))
        desenhar_labirinto(tela, matriz, visitados, agente=no_atual, passos=passos, custos=custos)
        pygame.display.flip()

        if no_atual == destino:
            print(f"Objetivo alcançado em {passos} passos!")
```

Figura 13: Implementação da busca gulosa: buscas.py

No início, são inicializadas as seguintes estruturas:

- Um conjunto para armazenar os nós já visitados.
- Uma fila de prioridade (min-heap) que organiza os nós com base nos valores calculados pela função heurística. Os nós com menor custo estimado são processados primeiro.
- Um dicionário para armazenar os custos estimados de cada nó em relação ao destino.
- Um contador para rastrear o número de passos realizados durante a busca.

A busca ocorre dentro de um laço principal, que continua enquanto houver elementos na fila de prioridade. A cada iteração:

- O nó com o menor custo heurístico é removido da fila para ser processado.
- Se o nó já foi visitado, ele é ignorado para evitar redundância.
- Caso contrário, ele é marcado como visitado, e o contador de passos é incrementado.

Durante a execução, a interface gráfica é continuamente atualizada para refletir o estado atual do labirinto, mostrando o agente, os passos realizados e os custos estimados de cada nó visitado.

Se o nó atual for o destino, a busca é encerrada, e o programa exibe o número de passos necessários para alcançar o objetivo.

A Busca Gulosa é eficiente em encontrar caminhos, mas pode falhar em identificar a solução mais curta se a heurística subestimar ou superestimar o custo real.

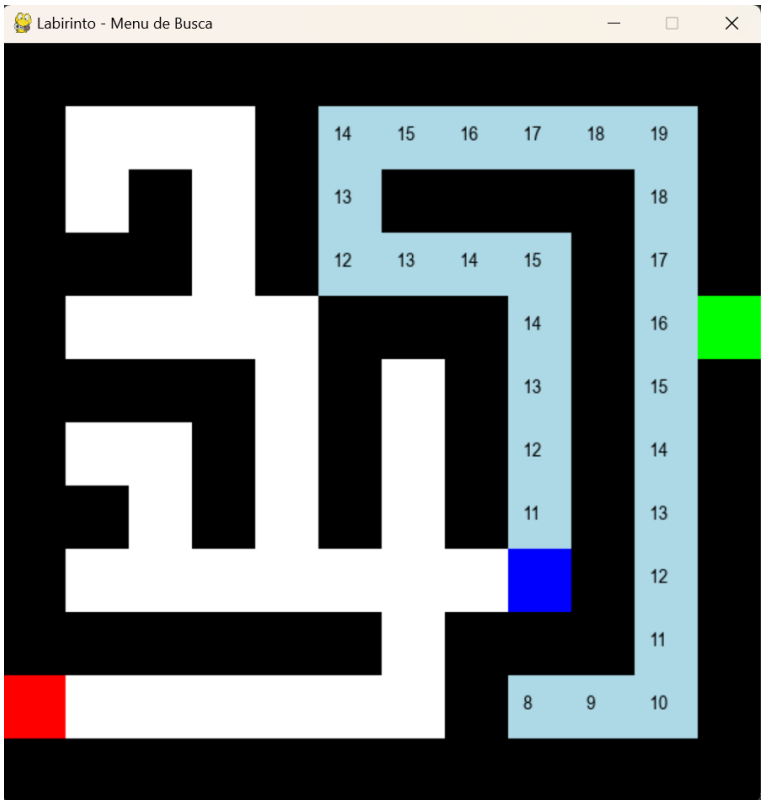


Figura 14: Execução da busca gulosa no Pygame: buscas.py

6 CONCLUSÃO

Neste trabalho, foram implementados três algoritmos clássicos de busca para resolução do problema de navegação em um labirinto: a Busca em Profundidade, a Busca em Largura e a Busca Gulosa heurísticas de Manhattan e Euclidiana.

6.1 Comparação entre os Algoritmos de Busca

- **Busca em Profundidade**: Explora os caminhos em profundidade, ou seja, ela vai o mais longe possível por uma ramificação. Este algoritmo é simples e consome menos memória, pois utiliza uma pilha para armazenar os nós a serem explorados. No entanto, ela não garante a solução mais curta.

- **Busca em Largura**: Esse algoritmo é mais eficaz quando o objetivo é encontrar o caminho mais curto, mas consome mais memória, pois precisa armazenar todos os nós da camada atual em uma fila. Em labirintos grandes, isso pode levar a uma maior utilização de recursos, mas, em contrapartida, garante que o caminho encontrado será o mais curto.

- **Busca Gulosa**: A Busca Gulosa, utilizando heurísticas como a Distância de Manhattan ou Euclidiana, é focada em encontrar o objetivo de forma rápida, priorizando os nós que parecem mais próximos do destino. Este algoritmo é eficiente em termos de tempo, pois reduz a quantidade de nós explorados ao guiar a busca diretamente para áreas promissoras do labirinto. No entanto, a escolha da heurística pode impactar a eficiência. Apesar de ser rápida, a Busca Gulosa não garante a solução ótima, pois pode ser enganada por uma heurística imprecisa.

6.2 Considerações finais

Com base nos resultados obtidos, a **Busca Gulosa** foi o algoritmo mais eficiente neste experimento, completando a busca em 47 passos. Esse desempenho pode ser atribuído à capacidade do algoritmo de priorizar os caminhos mais próximos do objetivo, utilizando uma heurística que guia a busca de forma mais eficaz.

A **Busca em Profundidade** e a **Busca em Largura** apresentaram um desempenho pior, com 54 e 56 passos, respectivamente. A busca em profundidade, apesar de ser mais simples e consumir menos memória, não garante o caminho mais curto, o que a torna menos eficiente para encontrar a solução rápida. Já a busca em largura garante o camninho mais curto , porém utiliza mais memória.

Portanto, em termos de eficiência de passos, a Busca Gulosa foi a mais rápida para encontrar o objetivo, seguida pela Busca em Profundidade e, por último, pela Busca em Largura. No entanto, a escolha do algoritmo ideal depende do contexto e dos objetivos específicos do problema. Se a prioridade for encontrar o caminho mais curto, a busca em largura seria a melhor escolha, mas se a eficiência em tempo for mais importante e o caminho ótimo não for essencial, a Busca Gulosa se destacaria como a mais adequada.