Alunos:
Pedro Pastorelo Fernandes 10262502
Vinícius Leite Ribeiro 10388200
Luiz Guilherme Martins 10392171

Relatório Projeto 1 - Inteligência Artificial

Introdução:

O objetivo desse projeto consiste na análise de 4 algoritmos de busca para observar a eficiência e o desempenho de cada um diante de uma entrada que foi estruturada em forma de um grafo, representando um mapa com obstáculos.

Descrição das implementações:

I) O programa inicia lendo o arquivo de entrada que contém a representação do caminho a ser estruturado e que serve como objeto base para os testes dos algoritmos de busca. A função de leitura, presente no arquivo map_parse.py, retorna uma matriz que representa o mapa contido no arquivo de entrada

```
def read_filemap(file_path):

    with open(file_path, "r") as file:
        line = file.readline()
        line = line.strip()
        dimensions = line.split(' ')

        arr = []
        while line:
            line = file.readline()
            if not line:
                break
            arr.append( line_to_list(line) )

    return arr
```

II) O conteúdo da entrada é passada para a função **create_graph** para ser retornada em forma de grafo.

A primeira etapa dessa função consiste em instanciar um objeto da classe Graph.

Essa classe possui todos os métodos necessários para a criação e a manipulação do grafo.

A estrutura do objeto consiste em:

- um dicionário responsável pelo armazenamento das coordenadas de cada nó, coordenadas dos possíveis caminhos para cada nó
- um atributo para representar o nó de início
- um atributo para representar o nó de destino

```
class Graph(object):

    def __init__(self):
        self.struct = dict()
        self.start_node = ''
        self.end_node   = ''
```

III) Após a criação do objeto, inicia-se o processo de criação do grafo
Esse processo é separado em duas etapas:

- A primeira consiste no armazenamento dos nós no grafo

```
for i, row in enumerate(map_array):
    for j, value in enumerate(row):

        name = G.node_name( map_array, (i,j) )

        if value != '-':
            G.insert_node(name)

        if value == '#':
            if G.start_node == '':
                G.set_start(name)
            else:
                print("ERRO: Mapa contem mais de um ponto inicial.")
                return None
        elif value == '$':
            if G.end_node == '':
                G.set_end(name)
            else:
                print("ERRO: Mapa contem mais de um ponto final.")
                return None
```

- A segunda consiste no armazenamento das arestas no grafo

```
for i, row in enumerate(map_array):
    for j, value in enumerate(row):

        if value != '-':
            for a in side_adjacents(map_array, (i,j)):
                ai = a[0]
                aj = a[1]
                if map_array[ai][aj] != '-':
                    G.insert_arc(G.node_name(map_array, (i,j)),
                                 G.node_name(map_array, a),
                                 WEIGHT_SIDE )

            for a in diag_adjacents( map_array, (i,j) ):
                ai = a[0]
                aj = a[1]
                if map_array[ai][aj] != '-':
                    G.insert_arc(G.node_name(map_array, (i,j)),
                                 G.node_name(map_array, a),
                                 WEIGHT_DIAG )
```

A partir disso, já temos o mapa estruturado em forma de grafo, o que nos permite iniciar o processo de análise dos algoritmos de busca.

depth_first

É um método recursivo de um objeto do tipo **Graph** que realiza uma busca em profundidade em um grafo**.**

Recebe dois parâmetros em sua chamada: start e end, que representam o ponto de início e fim. E também possui mais 2 parâmetros, que serão utilizados no momento das chamadas recursivas, representando o caminho, path, e uma flag para indicar se o nó já foi visitado, visited.

Vale ressaltar que esse algoritmo não considera o peso das arestas.

```
def depth_first(self, start, end, path=None, visited=None):
    if path is None:
        path = []

    if visited is None:
        visited = []

    visited.append(start)

    if start == end:
        path.append(start)
        return path

    node_edges = self.struct[start]
    for edge in node_edges:
        dest_node =  edge[0]

        if dest_node not in visited:

            subpath = self.depth_first(dest_node, end, path, visited)

            if subpath != None:
                path.append(start)
                return path
```

breadth_first:

Método iterativo de um objeto do tipo **Graph** que realiza uma busca em largura em um grafo.

Recebe apenas dois parâmetros em sua chamada: **start** e **end** que representam os pontos de início e de fim do caminho.

Vale ressaltar que esse algoritmo não considera o peso das arestas.

```
def breadth_first(self, start, end):
    queue = [[start]]

    while queue:
        path = queue.pop(0)

        node = path[-1]
        if node == end:
            return path

        node_edges = self.struct[node]
        for edge in node_edges:
            dest_node = edge[0]

            enqueue = list(path)
            enqueue.append(dest_node)

            queue.append(enqueue)
```

A*

Método iterativo de um objeto do tipo **Graph** que realiza uma busca informada utilizando a heurística de distância entre o nó atual e o nó de destino e peso do caminho percorrido.

Recebe 2 parâmetros: **start** e **end** que representam os pontos de início e de fim do caminho.

Pode ser considerado um Djikstra melhorado, pois ele busca o caminho mais eficiente sem a necessidade de percorrer o grafo todo

```
def a_star(self, start, end):
    pp = PP(depth=4)

    node_data  = {}
    fringe     = []
    path       = []
    done       = []
    found = False

    g = 0
    f = g + euclidian_distance(start, end)
    start_node = {
        start: {
            "g": g,
            "f": f,
            "predecessor": None
        }
    }

    fringe.append( (start, f) )
    node_data.update(start_node)

    while fringe:
        fringe.sort(key=lambda tup: tup[1])
        current_node = fringe.pop(0)[0]

        if current_node == end:
            found = True
            break

        if current_node in done:
            continue

        h = euclidian_distance(current_node, end)

        node_edges = self.struct[current_node]
        for edge in node_edges:
            dest_node   = edge[0]
            edge_weight = edge[1]

            if dest_node in done:
                continue
            if dest_node not in fringe:
                fringe.append( (dest_node, f) )

            g = node_data[current_node]['g'] + edge_weight
            f = g + h


            ###
            adjacent = {
                dest_node: {
                    "g": g,
                    "f": f,
                    "predecessor": current_node
                }
            }

            ###
            if dest_node not in node_data.keys():
                node_data.update(adjacent)
            elif f < node_data[dest_node]['f']:
                node_data[dest_node] = adjacent[dest_node]

        done.append(current_node)

###
if found:
    path.append(end)
    current_node = node_data[end]

    while current_node['predecessor'] != None:
        pred = current_node['predecessor']
        path.append(pred)
        current_node = node_data[pred]

    return path[::-1]

else:
    return None
```

Best First

Um método iterativo do tipo **Graph** que realiza uma busca informada semelhante ao algoritmo A*, porém, utiliza uma heurística mais simples que não consideram o peso do caminho percorrido até o nó atual, apenas entre o nó atual e o nó final

```python
def best_first(self, start, end):
    node_data  = {}
    open_nodes = []
    path       = []
    done       = []
    found = False

    h = heuristic_calc(start, end)
    start_node = {
        start: {
            "h": h,
            "predecessor": None
        }
    }

    open_nodes.append( (start, h) )
    node_data.update(start_node)

    while open_nodes:
        open_nodes.sort(key=lambda tup: tup[1])
        open_nodes = open_nodes[::-1]
        current_node = open_nodes.pop(0)[0]

        if current_node == end:
            found = True
            break

        if current_node in done:
            continue

        dist = heuristic_calc(current_node, end)

        node_edges = self.struct[current_node]
        for edge in node_edges:
            dest_node   = edge[0]
            edge_weight = edge[1]

            if dest_node in done:
                continue

            if dest_node not in open_nodes:
                open_nodes.append( (dest_node, h) )

            h = dist + edge_weight

            adjacent = {
                dest_node: {
                    "h": h,
                    "predecessor": current_node
                }
            }

            if dest_node not in node_data.keys():
                node_data.update(adjacent)
            elif h < node_data[dest_node]['h']:
                node_data[dest_node] = adjacent[dest_node]

        done.append(current_node)

    if found:
        path.append(end)
        current_node = node_data[end]

        while current_node['predecessor'] != None:
            pred = current_node['predecessor']
            path.append(pred)
            current_node = node_data[pred]

        return path[::-1]

    else:
        return None
```

Heurística

Foram implementadas duas funções de heurística para os algoritmos de busca informada. A primeira, euclidian_distance(), calcula a distância em linha reta entre o ponto atual e o ponto final. A outra, leg_distance(), calcula a distância pela soma do tamanho dos catetos do triângulo formado entre o ponto atual e o ponto final.

```
# calculo da heurisitca para um no
def heuristic_calc(node_a, node_b):
    return euclidian_distance(node_a, node_b)

# heuristica de distancia em linha reta
def euclidian_distance(node_a, node_b):
    a = literal_eval(node_a)
    b = literal_eval(node_b)

    hor = abs(b[0] - a[0])
    ver = abs(b[1] - a[1])

    return sqrt(hor**2 + ver**2)

# heuristica de distancia da soma dos catetos
def leg_distance(node_a, node_b):
    a = literal_eval(node_a)
    b = literal_eval(node_b)

    hor = abs(b[0] - a[0])
    ver = abs(b[1] - a[1])

    return hor + ver
```

Média de cada algoritmo

Nome	Proporção	Tempo
Nome do algoritmo	Proporção de cada teste	Média em segundos

	13:15|	200:40|
Depth First	$4{,}5 \times 10^{-4}$	
Breadth First	$3{,}5 \times 10^{-3}$	2.75
Best First	2.5×10^{-2}	6.45
A*	3.4×10^{-3}	4.20

Conclusão

Após a execução de todos os algoritmos, tomada dos tempos médios de execução e a comparação dos dados obtidos, podemos concluir que o melhor algoritmo varia de acordo com o caso. Dependendo da topologia do mapa apresentado, um algoritmo pode ser mais eficiente que outro ou vice versa. Contudo, o algoritmo que, em média, apresenta melhores valores e, portanto pode ser considerado mais genérico é o A*. A heurística utilizada pelo mesmo é mais complexas e mais abrangente.