Alunos: Harã Heique e Jadson Pereira
Github: https://github.com/HaraHeique/pathfinding-Aestrela-IA

1. Como utilizar o algoritmo de busca A*

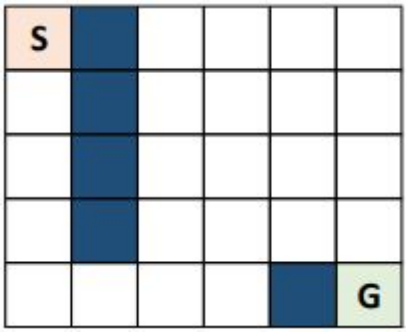
Neste tutorial iremos utilizar o algoritmo A* em uma matriz 2D para a partir de um ponto inicial atingir o ponto final pelo melhor caminho, os quais ambos são pré-definidos pelo usuário.

1.1 Teoria

Na ciência da computação, A* é um algoritmo de busca que foi escrito pela primeira vez em 1968 por Peter Hart, Nils Nilsson, e Bertram Raphael com finalidade de buscar um trajeto de um vértice inicial até um vértice final combinando a aproximação heurística e da formalidade do algoritmo de dijkstra.

2. Problema proposto

Nosso problema consiste em utilizar o algoritmo de A* para traçar a menor rota em uma matriz 2D 6x6 movimentando-se somente em linha reta com rotações de 90 graus. O ponto inicial e o ponto final foram predefinidos como S = (0,0) e G = (4,5).



Fonte: ANÁLISE DO ALGORITMO A* (A ESTRELA) NO PLANEJAMENTO DE ROTAS DE VEÍCULOS AUTÔNOMOS (Betina Carol Zanchi)

3. Implementação

Para a implementação do algoritmo de busca utilizamos a linguagem de programação Python que vem ganhando bastante popularidade no meio da IA, devido ser uma linguagem de script bem fácil de ser aprendida e consequentemente aumento o nível de produtividade.
Logo partir de uma matriz grid gerada através de um arquivo de entrada (txt) e aplicando o cálculo da distância de manhattan nesta matriz obtivemos uma matriz heurística que por sua vez foi utilizada para calcular o melhor caminho traçado.

3.1. Explicar os trechos mais importantes da implementação

Primeiramente a função principal que é responsável por chamar todos os módulos e funções do algoritmo, desde a leitura do arquivo com a matriz a ser lida como mapa até o retorno do melhor caminho traçado pelo algoritmo proveniente no módulo libraries.pathfindinHeuristic.

```python
import libraries.userInput as userInput
import libraries.utilities.matricesHandle as mtzHandle
import libraries.pathfindingHeuristic as heuristic
import os

def main(filesDir) :
    # Cria a matriz grid a partir de um arquivo
    mtzMap: list = mtzHandle.openMtzFromPathFile(filesDir + "map_default.txt")

    # Pega as informações dos pontos inicial e final inseridos pelo usuário
    mtzHandle.printar(mtzMap)
    pontoInicial: tuple = userInput.pontoInicial(mtzMap)
    pontoFinal: tuple = userInput.pontoFinal(mtzMap)

    # Cria a matriz de heurística baseado na matriz grid baseado no cálculo da distância de Manhatan
    mtzHeuristic: list = heuristic.matrizHeuristica(mtzMap, pontoFinal)
    mtzHandle.printar(mtzHeuristic)

    # Pega o melhor caminho traçado pelo algoritmo com base na matriz heurística
    caminhoTracado: list = heuristic.getCaminhoPercorrido(mtzMap, mtzHeuristic, pontoInicial, pontoFinal)
    print("\nCaminho Percorrido: {0}".format(caminhoTracado))

    # Imprime o percurso percorrido pela matriz
    heuristic.drawMapTraced(mtzMap, caminhoTracado)
    mtzHandle.printarStr(mtzMap)

    return 0

if __name__ == '__main__':
    filesDir = os.path.dirname(os.path.realpath('__file__')) + "/src/files/"
    main(filesDir)
```

Figura 1 - Função principal da aplicação.

Essa parte do código pega o ponto selecionado pela entrada inserida pelo usuário, onde além de retornar o ponto também faz validações da entrada do usuário que são: checar se o ponto inserido segue o formato pré-definido, ou seja, (x,y); checar se o ponto viola os limites da matriz; checar se o ponto selecionado é uma obstáculo no mapa.

```python
def __getPontoSelecionado(mtzMap: list, ptoInput: str) -> tuple :
    if (not __isPtoInputValid(ptoInput)) :
        print("O ponto não segue o formato(x,y) desejado.")
        return None

    ptoInput = ptoInput.split(',')
    ptoSelecionado: tuple = (int(ptoInput[0]), int(ptoInput[1]))

    if (not __isValidPto(mtzMap, ptoSelecionado)) :
        print("O ponto inserido viola os limites da matriz.")
        return None

    if (not __isObstaculo(mtzMap, ptoSelecionado)) :
        print("O ponto selecionado é um obstáculo. Favor selecionar os de valor 0")
        return None

    return ptoSelecionado
```

Figura 2 - Função de pegar e validar o ponto selecionado pelo usuário.

A função a seguir constrói a matriz heurística, dado um ponto final, ela percorre ponto a ponto calculando a distância usando o método do cálculo de distância de Manhattan.

```python
# Retorna a matriz heurística baseado na matriz passada que é o mapa
def matrizHeuristica(mtz: list, pontoFinal: tuple) -> list:
    matEuri: list = []
    lin: int = len(mtz)
    col: int =  len(mtz[0]) if (lin > 0) else 0

    for i in range(lin):
        matEuri.append([])

        for j in range(col):
            # Caso seja diferente de zero é porque é um obstáculo logo seta um valor negativo
            if(mtz[i][j] != 0):
                matEuri[i].append(-1)
            else:
                # Faz o cálculo da distância de Manhattan do ponto corrente com o ponto final
                matEuri[i].append((abs(i-pontoFinal[0])+(abs((j-pontoFinal[1])))))

    return matEuri
```

Figura 3 - Função de construção da matriz heurística baseado na distância de Manhattan.

Perceba que é retornado uma nova matriz que é usada como base para definir o melhor caminho a ser percorrido até chegar ao ponto final.
O código a seguir, seleciona a lista de caminhos baseado na matriz heurística, onde enquanto o ponto corrente (ptoCurrent) for diferente do ponto final (ptoFinal) é chamado a função __nextPonto que é responsável por pegar o próximo ponto/célula do mapa percorrido.

```python
def getCaminhoPercorrido(mtzMap: list, mtzHeuristica: list, ptoInicial: tuple, ptoFinal: tuple) -> list:

    if (ptoInicial == ptoFinal):
        return [ptoInicial]

    #CUSTO: int = 0 # Será o custo de percorrer para qualquer caminho
    ptoCurrent: tuple = ptoInicial
    found: bool = False
    pathTraced: list = [ptoInicial]

    while (not found):
        # Guarda o ponto anterior para que seja comparado com o novo ponto corrente/atual
        ptoAnteriorTraced = pathTraced[-2] if (len(pathTraced) > 1) else pathTraced[-1]
        ptoCurrentAntes = ptoCurrent
        ptoCurrent = __nextPonto(mtzHeuristica, ptoAnteriorTraced, ptoCurrent)

        if (ptoCurrent == ptoCurrentAntes):
            raise Exception("Não é possível sair do ponto {0}. Favor checar o código".format(ptoCurrent))

        # Adiciona o novo ponto ao caminho percorrido pelo algoritmo
        pathTraced.append(ptoCurrent)

        found = True if (ptoCurrent == ptoFinal) else False

    return pathTraced
```

Figura 4 - Função de retorno do caminho percorrido pelo algoritmo.

A figura a seguir demonstra a função que é responsável por pegar o próximo ponto/célula e retorná-la como uma tupla. Perceba que nela são guardados na lista chamada *direcoes*,

que age como uma espécie de cache, as tuplas com as coordenadas das células cima, baixo, esquerda e direita respectivamente.

```python
def __nextPonto(mtzHeuristica: list, ptoAnterior: tuple, ptoCurrent: tuple) -> tuple:

    LINMAX = len(mtzHeuristica)-1
    COLMAX = len(mtzHeuristica[0])-1 if (LINMAX > 0) else 0

    # As direções são respectivamente cima, baixo, esquerda, direita
    direcoes = [(ptoCurrent[0]-1, ptoCurrent[1]),
                (ptoCurrent[0]+1, ptoCurrent[1]),
                (ptoCurrent[0], ptoCurrent[1]-1),
                (ptoCurrent[0], ptoCurrent[1]+1)]

    # Seta o valor como NULL inicialmente até que pegue o primeiro ponto que seja válido
    menorCustoFuncaoHeuristica = None

    # Percorrendo pelas direções e checando qual delas será a selecionada como próximo caminho corrente
    for direcao in direcoes:
        # Checando se quebra os limites da matriz
        if (direcao[0] < 0 or direcao[0] > LINMAX or direcao[1] < 0 or direcao[1] > COLMAX):
            continue

        # Checa se o ponto da direção não é um obstáculo ou se já foi percorrido
        if (ptoAnterior == direcao or mtzHeuristica[direcao[0]][direcao[1]] < 0):
            continue

        # Caso a célula da matriz heurística na determinada direção seja menor do que o ponto corrente
        if (menorCustoFuncaoHeuristica == None or mtzHeuristica[direcao[0]][direcao[1]] < menorCustoFuncaoHeuristica):
            ptoCurrent = direcao
            menorCustoFuncaoHeuristica = mtzHeuristica[direcao[0]][direcao[1]]

    return ptoCurrent
```

Figura 5 - Função de retorno do próximo ponto selecionado pelo algoritmo.

Note que ao percorrer as direções o algoritmo checa qual delas possui o menor valor de F = função heurística + custo, e o retorna assim que finaliza a execução da função.

4. Resultados

Com base no algoritmo de busca A* que desenvolvemos, concluímos que o melhor caminho para o problema proposto foi o caminho ilustrado a seguir.

```
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (3, 3), (3, 4), (3, 5), (4, 5)]
```

Ao executar o código é exibido a matriz grid ao usuário e questionado os pontos iniciais e finais. Como falado antes, com essas informações é montada a matriz heurística (exibida após os dados) que é fundamental para que o algoritmo A* seja eficaz.

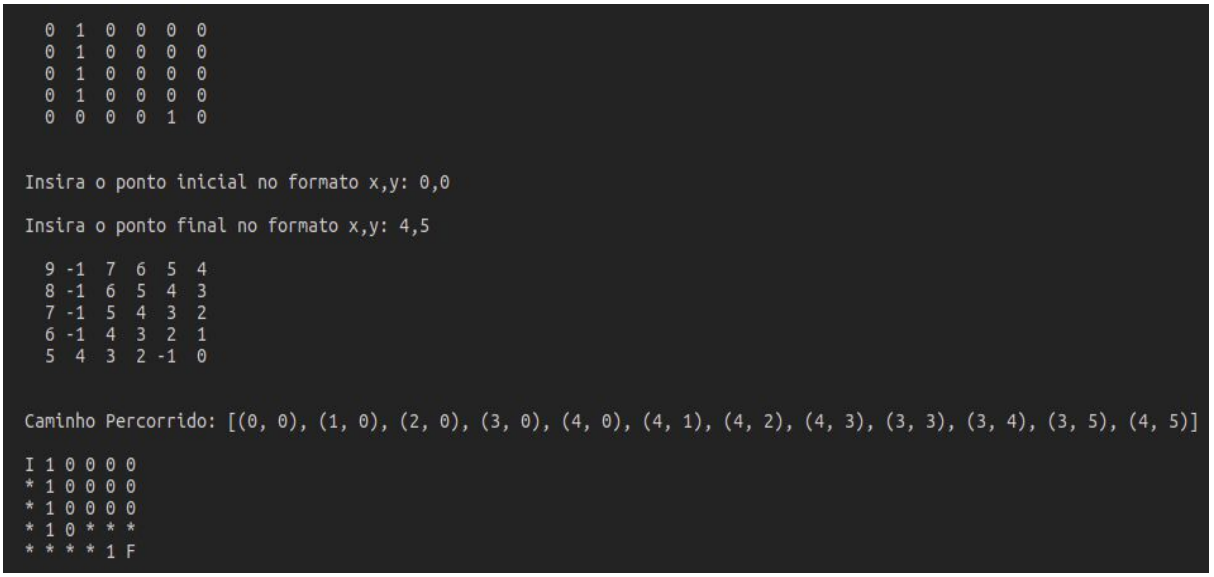


Figura 6 - Resultado final da execução da main.py