ANTÔNIO CARLOS DURÃES DA SILVA
JOEL WILL BELMIRO

RELATÓRIO – IMPLEMENTAÇÃO DO ALGORITMO DE BUSCA A*

1 EXPLICAÇÃO TEÓRICA DO ALGORITMO

O Algoritmo A* tem como objetivo encontrar um caminho entre em pontos (nós ou vértices), fazendo uso de heurísticas para reduzir a quantidade de operações que serão necessárias para se ter um resultado e assim se pode tratar grandes quantidades de possibilidades de caminho em tempo hábil (computacionalmente), sendo este resultado, o caminho o mais próximo do que seria o melhor caminho. Por conta dessa heurística, não pode-se afirmar que o caminho escolhido é o melhor, pois para isso, seria necessário passar por todos caminhos possíveis e verificar o menor.

O algoritmo cria um plano, representado como uma matriz ou grafo, e verifica qual o ponto adjacente (em relação ao atual) tem o menor peso, sendo este, calculado através da heurística escolhida.

Como heurística, usa-se para cada ponto, o custo de movimentação do ponto atual a outro ponto adjacente a ele, somado ou a distância a partir do ponto adjacente ao ponto final ou a distância a partir do ponto adjacente ao ponto atual. A distância pode ser calculada de duas formas: distância euclidiana ou distância de Manhattan.

2 PROBLEMA PROPOSTO:

Dado uma matriz, lida através de um arquivo-texto, com suas colunas separadas por um espaço em branco. Cada linha do arquivo é uma linha da matriz, contendo para cada posição 0s e 1s, onde 0 representa uma posição livre para passagem e 1 representa um obstáculo, além de um ponto inicial e um ponto final. Deseja-se encontrar o melhor caminho para se chegar ao objetivo, ou identificar que não existe um caminho que ligue os dois pontos.

3 IMPLEMENTAÇÃO:

- Código no GitHub: https://github.com/duraes-antonio/IA_trab1_A_Star

Foram implementados os dois tipos de cálculo de distância, apesar de ser usado apenas o de Manhattan, o outro cálculo pode ser usado caso queira.

Texto 1: Métodos para calcular a distância

```
def calc_dist_manhattan(ponto_1: Ponto, ponto_2: Ponto) -> int:
    return abs(ponto_1.x - ponto_2.x) + abs(ponto_1.y – ponto_2.y)

def calc_dist_euclid(ponto_1: Ponto, ponto_2: Ponto) -> float:
    return ((ponto_1.x - ponto_2.x) ** 2 + (ponto_1.y - ponto_2.y) ** 2) ** (0.5)
```

A função “calc_trajeto” é realmente quem aplica o algoritmo A*. Ela recebe como parâmetro, o plano (um objeto do tipo PlanoCartesiano), o ponto inicial e final (objetos do tipo Ponto) e o símbolo que representa um bloqueio/obstáculo dentro do plano (importante para o algorítimo identificar quais pontos são obstáculos).

A primeira coisa a ser feita de fazer com que o ponto atual seja o ponto inicial, depois inicializa uma lista vazia que representa com os pontos em que todos os vizinhos já foram visitados (pts_fechados) e uma lista com os pontos que devem ser verificados (pts_abertos). Depois se pega a lista de pontos que são obstáculos e por fim define-se uma flag para saber se encontrou o caminho ao sair do loop ou não.

*Texto 2: Assinatura e trecho inicial do A**

```
def calc_trajeto(plano: PlanoCartesiano, pt_inicial: Ponto, pt_final: Ponto,
                 simbolo_bloqueio: object) -> List[PontoAStar]:

   # Defina como o primeiro ponto corrente, o ponto inicial recebido;
   pt_atual = pt_inicial

   # Armazene os nós já percorridos (p/ evitar laços s/ fim e mov. desnecessários)
   pts_fechados: List[PontoAStar] = []
   pts_abertos: List[PontoAStar] = [pt_atual]

   # Armazene os pontos que representam obstáculos;
   obstaculos = plano.get_pts_bloqueados(simbolo_bloqueio)
   encontrou = False
```

Chega-se então ao loop principal, que será executado enquanto existir pontos que devem ser verificados ou até se o ponto atual ser o ponto final. Dentro do loop é feita uma ordenação (crescente) dos pontos abertos em relação ao resultado da heurística de cada um, e então se atribui ao ponto atual o que possui o menor valor. Após isso, se adiciona o ponto atual a lista de pontos fechados e verifica se o ponto atual é igual ao ponto final, pois caso seja, muda-se a flag e quebra o loop.

Os três próximos passos são para se saber quem são os vizinhos do ponto atual, tanto na horizontal quanto na vertical, mas sem que eles sejam obstáculos ou estejam na lista de pontos fechados, pois isso evita loops infinitos e também a escolha de um obstáculo como um caminho.

Texto 3: Ordenação da lista de pontos abertos, obtenção e limpeza da lista de vizinhos

```
# Enquanto houver pontos candidatos a serem verificados;
while (pts_abertos):

   # Obtenha o ponto da lista de abertos c/ o menor custo final;
   pts_abertos = sorted(pts_abertos, key=lambda pt: pt.f)
   pt_atual: PontoAStar = pts_abertos.pop(0)

   # Adicione o ponto atual na lista de pontos percorridos;
   pts_fechados.append(pt_atual)

   if pt_atual == pt_final:
      encontrou = True
      break

   # Obtenha os pontos vizinhos verticais e horizontais do ponto atual;
   vizinhos: List[PontoAStar] = plano.get_pts_adj_horiz_vert(pt_atual.x, pt_atual.y)

   # Remova os obstáculos dos pontos vizinhos;
   vizinhos = [pt for pt in vizinhos if pt not in obstaculos]

   # Remova os pontos já percorridos da lista de pontos vizinhos;
   vizinhos = [pt for pt in vizinhos if pt not in pts_fechados]
```

Logo em seguida, entra-se em um loop que passa por vizinho, e que é peça fundamental para se obter o melhor caminho. É neste momento que os pontos

começarão a ter os valores de G e H calculados ou recalculados, e terão como pai o ponto atual, caso o G até ele seja o menor ou caso ele nunca tenha sido aberto antes.

Texto 4: Cálculo de G, H e F, preenchimento da lista de abertos e fechados

```
# Para cada vizinho do ponto atual;
for pt_viz in vizinhos:

   # Se o vizinho atual já estiver na lista de candidatos;
   if (pt_viz in pts_abertos):

      # Calcule a dist. G do ponto atual até o vizinho atual;
      g_dist = pt_atual.g + calc_dist_manhattan(pt_atual, pt_viz)

      # Se G do ponto atual até o viz. for menor que o G antigo do vizinho;
      if (g_dist < pt_viz.g):

         # Defina o ponto atual como pai do ponto vizinho E atualize seu G;
         pt_viz.pt_pai = pt_atual
         pt_viz.g = g_dist

   # Senão, atualize as distâncias do ponto vizinho e defina seu ponto pai;
   else:

      pt_viz.pt_pai = pt_atual

      # Calcule sua distância (valor G) até o ponto atual;
      pt_viz.g = pt_atual.g + calc_dist_manhattan(pt_atual, pt_viz)

      # Calcule a dist. H do vizinho até o objetivo;
      pt_viz.h = calc_dist_manhattan(pt_viz, pt_final)
      pts_abertos.append(pt_viz)
```

Após o loop principal finalizar, ele chega nesse trecho de código, que caso tenha sido encontrado um caminho, ele passa por todos pontos pais, a partir do último (que é o ponto atual), e o adiciona na lista de saída e depois atribui ao ponto atual o pai dele. No fim desse processo se tem o caminho de trás para frente, bastando apenas revertê-lo para se obter o caminho no sentido correto

Texto 5: Agrupamento dos pontos do trajeto final, ordenação e retorno

```
saida = []

if encontrou:

   # Adicione o pai do ponto no trajeto final até que um ponto sem pai seja encontrado;
   while (pt_atual.pt_pai):
      saida.append(pt_atual.pt_pai)
      pt_atual = pt_atual.pt_pai

   # Remova o ponto inicial do trajeto; Por fim inverta o trajeto;
   saida.remove(pt_inicial)
   saida.reverse()
   saida.append(pt_final)

return saida
```

4 EXEMPLO DE CHAMADA

Logo abaixo há um exemplo de chamada ao programa. Para colorir a saída no terminal foi utilizado uma lib para o Python, chamada de Colorama. Portanto, é importante executar a aplicação ao menos uma vez como administrador ou root (sudo, no caso de sistemas Unix).

Texto 6: Exemplo de chamada e execução

```
x@pc IA_trab1_A_Star/src $ python3 main.py

Digite ou cole o path do arquivo contendo o plano:
plano.txt

Entre com o valor x, y do ponto INICIAL (Ex.: '2 3'):
0 0

Entre com o valor x, y do ponto FINAL (Ex.: '2 3'):
5 -4

                                  0   1   2   3   4   5
                                  —   —   —   —   —   —
                                0| *   ■   -   -   -   -
                               -1| -   ■   -   -   -   -
                               -2| -   ■   -   -   -   -
                               -3| -   ■   -   -   -   -
                               -4| -   -   ■   -   ■   @
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
NÃO HÁ CAMINHOS POSSÍVEIS!!!

* - PONTO INICIAL
@ - PONTO FINAL
- - PONTO COMUM
■ - OBSTÁCULOS
▲ - TRAJETO FINAL
```

5 RESULTADOS

Texto 7: Exemplo 1 - Caminho encontrado pelo algorítimo

```
        0   1   2   3   4   5   6   7
        —   —   —   —   —   —   —   —
    0| -   ■   ■   ■   ■   ■   ■   -
   -1| -   -   -   ■   -   ■   -   ■
   -2| -   ■   -   ■   ■   -   -   -
   -3| -   ■   -   ■   -   ■   @   -
   -4| -   -   -   -   -   ■   ▲   -
   -5| ■   -   -   -   ■   ■   ▲   -
   -6| ■   ■   ■   -   -   ■   ▲   -
   -7| *   ►   ►   ►   ►   ►   ►   -
   -8| -   -   ■   -   ■   ■   -   ■
   -9| -   -   -   -   ■   -   ■   -
- - - - - - - - - - - - - - - - - - - -
PONTOS percorridos (X, Y):
[(1, -7), (2, -7), (3, -7), (4, -7), (5, -7), (6, -7), (6, -6), (6, -5), (6, -4), (6, -
3)]
- - - - - - - - - - - - - - - - - - - -
Distância: 10

* - PONTO INICIAL
@ - PONTO FINAL
- - PONTO COMUM
■ - OBSTÁCULOS
▲ - TRAJETO FINAL
```

Texto 8: Exemplo 2 - Caminho encontrado pelo algorítimo

```
        0   1   2   3   4   5
        —   —   —   —   —   —

    0| -   ■   -   -   -   @
   -1| *   ■   ■   -   -   ▲
   -2| ▼   ■   -   -   ■   ▲
   -3| ▼   ■   -   ▲   ►   ►
   -4| ▼   ►   ►   ►   ■   -
- - - - - - - - - - - - - - - - - - - -
PONTOS percorridos (X, Y): [(0, -2), (0, -3), (0, -4), (1, -4), (2, -4), (3, -4), (3, -
3), (4, -3), (5, -3), (5, -2), (5, -1), (5, 0)]
- - - - - - - - - - - - - - - - - - - -
Distância: 12

* - PONTO INICIAL
@ - PONTO FINAL
- - PONTO COMUM
■ - OBSTÁCULOS
▲ - TRAJETO FINAL
```

Texto 6: Exemplo de caso onde não há caminhos até o nó final.

```
                          0   1   2   3   4   5
                          —   —   —   —   —   —
                       0| -   ■   -   -   -   -
                      -1| -   ■   ■   -   -   -
                      -2| -   ■   *   ■   ■   -
                      -3| -   ■   ■   -   -   -
                      -4| -   -   -   -   ■   @
- - - - - - - - - - - - - - - - - - - - - - - -
NÃO HÁ CAMINHOS POSSÍVEIS!!!

* - PONTO INICIAL
@ - PONTO FINAL
- - PONTO COMUM
■ - OBSTÁCULOS
▲ - TRAJETO FINAL
```