

Relatório de Implementação do Algoritmo A* (A Estrela)

Integrantes

- Anne Caroline Silva
- Mellyssa Mendes

1. Introdução

Neste relatório, vamos explicar como foi feita a implementação do Algoritmo A* (lê-se A-Estrela) para traçar o melhor caminho (utilizando heurísticas), partindo de um ponto inicial até o ponto final, onde o mesmo somente se movimenta em linha reta e rotaciona 90 graus, desviando de alguns obstáculos no caminho.

2. Explicação Teórica do Algoritmo

O Algoritmo A* tem como objetivo encontrar um caminho entre 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 os caminhos possíveis e verificar o menor.

O algoritmo A* avalia os nós através da combinação de $g(n)$ que é o custo para alcançar cada nó com a função $h(n)$ que é o menor custo partindo da origem para se chegar ao destino, matematicamente dado na equação: $F(n) = G(n) + H(n)$

onde,

- **G(n)**: custo do caminho do nó inicial para n;
- **H(n)**: função heurística que estima o custo do caminho mais barato de n para a meta;
- **F(n)**: n é o próximo nó no caminho.

As características definidoras do algoritmo A* são a construção de uma "lista fechada" para registrar áreas já avaliadas, uma lista aberta para registrar áreas adjacentes àquelas já avaliadas e o cálculo das distâncias percorridas desde o "ponto inicial" com distâncias estimadas até o "ponto objetivo".

A lista aberta, é uma lista de todos os locais imediatamente adjacentes a áreas que já foram exploradas e avaliadas (a lista fechada). A lista fechada é um registro de todos os locais que foram explorados e avaliados pelo algoritmo.

3. Problema Proposto

Implementar o algoritmo A* para resolver o seguinte problema:

Dado um mapa com obstáculos, o algoritmo deve traçar o caminho menos custoso, do ponto inicial até o ponto final.

Nosso mapa vem em um arquivo .txt, composto de 0's e 1's, onde 0 representa o caminho livre e 1 representa o obstáculo. Além disso, temos ponto de partida e chegada, que chamamos, respectivamente, de ponto inicial e ponto final. Nossa implementação devia obedecer algumas restrições:

- Leitura do mapa através de arquivo;
- Uso de diferentes mapas;
- Uso de heurística para reconhecer o caminho de menor custo;
- Locomover-se somente em linha reta ou em 90°;
- Ao final, mostrar um mapa com o caminho percorrido, juntamente com uma lista contendo as coordenadas utilizadas.

Como heurística, escolhemos a heurística de Manhattan, que tem esse nome pois define a menor distância entre quarteirões numa malha urbana reticulada ortogonal, como na própria zona da Cidade de Manhattan, EUA.

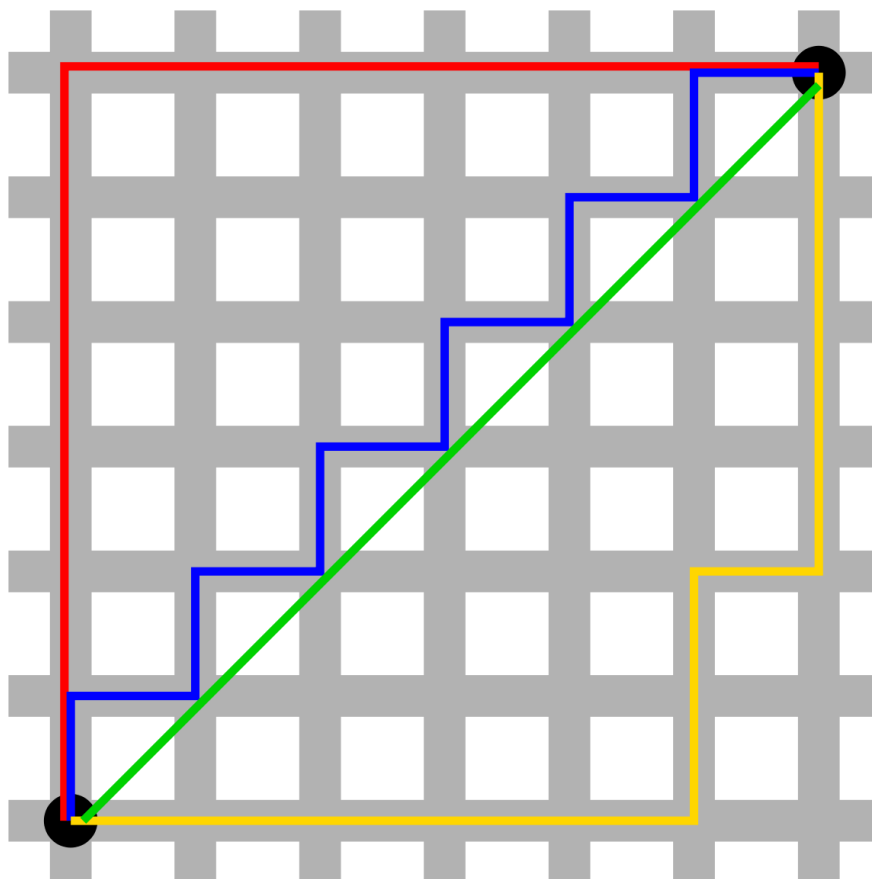


Figura 1: Representação da menor distância possível que um carro é capaz de percorrer numa malha urbana reticulada ortogonal

Na figura abaixo, temos a representação de um mapa, onde cada quadrado representa uma coordenada. O quadrado vermelho representa o ponto de partida, o verde o destino e os pretos são os obstáculos.

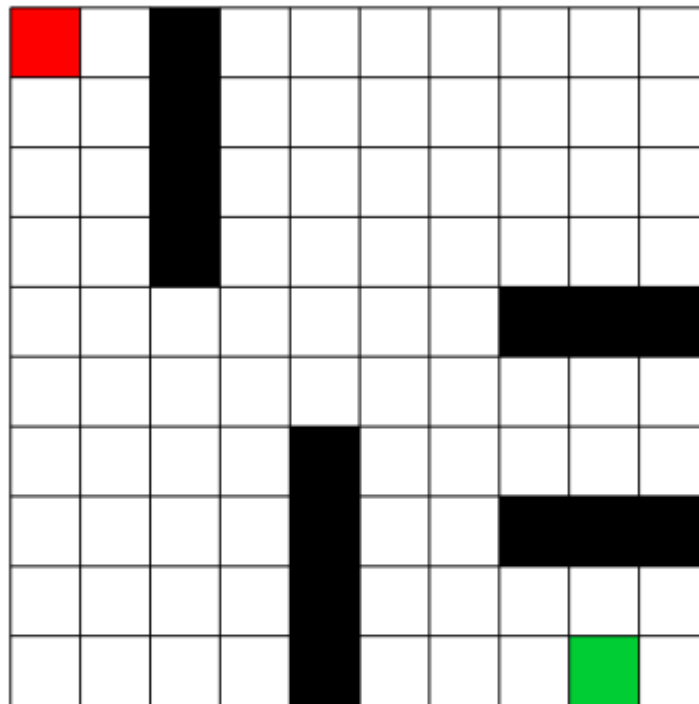


Figura 2: Exemplo de caminho a ser percorrido pelo algoritmo

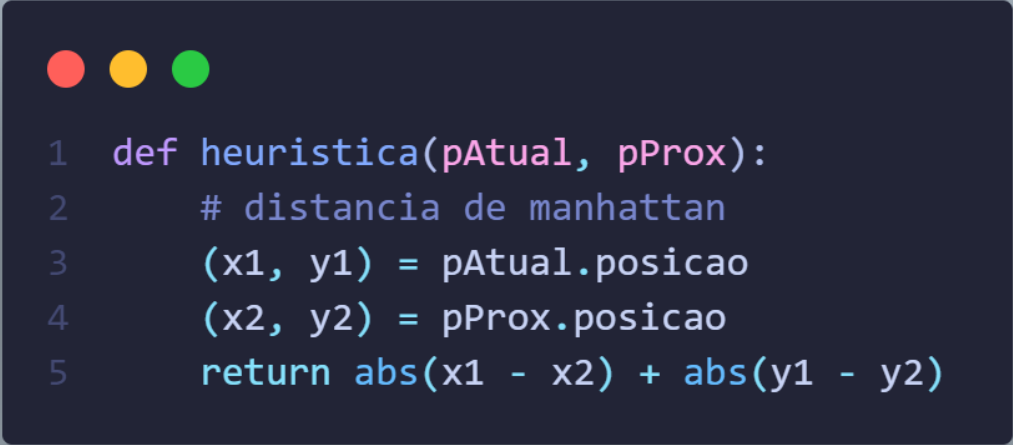
4. Implementação

Esse é um algoritmo guloso, em cada iteração ele faz a escolha que parece ser a melhor possível de acordo com algum critério (Best-First Search).

Para ser eficiente, o algoritmo armazena um conjunto de estados não explorados chamado de franja (do inglês, fringe). Inicialmente, a franja possui apenas o estado inicial. Na iteração seguinte, o estado inicial já foi explorado e a partir daí a franja armazena os estados sucessores do estado inicial. Com isso, o algoritmo vai montando um caminho passo-a-passo que permite escolher qual estado está mais próximo da meta.

5. Trechos mais importantes do código

Foi implementado o cálculo da distância de Manhattan, que aplica o algoritmo A*. Essa função recebe como parâmetro dois pontos e calcula a distância usando o método da distância de Manhattan.



```
1 def heuristica(pAtual, pProx):  
2     # distancia de manhattan  
3     (x1, y1) = pAtual.posicao  
4     (x2, y2) = pProx.posicao  
5     return abs(x1 - x2) + abs(y1 - y2)
```

A primeira coisa a ser feita é instanciar tanto o nó inicial quanto o final, porque esses nós serão utilizados no mapa.

Depois é criada uma lista vazia que representa os pontos em que todos os vizinhos já foram visitados (listaFechada) e uma lista com os pontos que devem ser verificados (listaAberta) e adiciona o nó inicial na lista de pontos a serem verificados. É definido também uma flag para saber se encontrou o caminho ao sair do loop e uma lista que vai armazenar o caminho que o algoritmo percorreu.

```
1 def astar(mapa, ini, fim):
2     # instancia o no inicial e final
3     noInicial = Node(None, ini)
4     noInicial.g = noInicial.h = noInicial.f = 0
5     noFinal = Node(None, fim)
6     noFinal.g = noFinal.h = noFinal.f = 0
7
8     # cria a "lista aberta" e a "lista fechada"
9     listaAberta = []
10    listaFechada = []
11
12    # adiciona o no inicial a lista aberta
13    listaAberta.append(noInicial)
14
15    # cria uma variavel flag para saber se o alvo foi encontrado
16    achou = False
17    # lista que guarda o caminho percorrido ate o no final
18    path = []
```

Chega-se então ao loop principal, que será executado enquanto existir pontos que devem ser verificados. 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 à 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.



```
1 while achou == False:
2     # pesquisa o no com menor F da lista aberta
3     noCorrente = min(listaAberta)
4     # remove o no corrente da lista aberta
5     index = listaAberta.index(noCorrente)
6     listaAberta.pop(index)
7     # adiciona o no corrente na lista fechada
8     listaFechada.append(noCorrente)
```

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 menor ou caso ele nunca tenha sido aberto antes.



```
1 for pos in [(-1, 0), (0, 1), (1, 0), (0, -1)]:
2     # pega as posicoes de todos os nos adjacentes
3     posAdj = (noCorrente.posicao[0] + pos[0], noCorrente.posicao[1] + pos[1])
4     # cria um no com as posicoes adjacentes
5     noAdjacente = Node(None, posAdj)
6
7     if posAdj[0] < 0 or posAdj[1] < 0 or posAdj[0] >= len(mapa) or posAdj[1] >= len(mapa[-1]):
8         continue
9     if mapa[posAdj[0]][posAdj[1]] == 1 or noAdjacente in listaFechada:
10        continue
```

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 pontos verificados. No fim desse processo se tem o caminho de trás para frente, bastando apenas revertê-lo para se obter o caminho no sentido correto.



```
1     # condições de parada
2     if len(listaAberta) == 0 or noCorrente == noFinal:
3         no = noCorrente
4         while no is not None:
5             path.append(no.posicao)
6             no = no.pai # adiciona os pais na lista path
7         achou = True
8     return path[::-1] # retorna o caminho reverso
```

