

Busca A*

O algoritmo que possui a solução mais amplamente conhecida para problemas de busca é o A*. Este algoritmo avalia o custo para se alcançar um nó ($g(n)$) e o custo para ir do nó ao objetivo ($h(n)$), nos dando a função:

$$f(n) = g(n) + h(n).$$

Para encontrar o caminho de menor custo é feito o cálculo de $f(n)$ para cada nó, escolhendo assim os nós com menor $f(n)$ que possam ser visitados durante o processo.

Comparando o A* com o algoritmo de Dijkstra, é percebido que a diferença está na presença da função heurística que reduz a quantidade de nós expandidos através da priorização dos nós.

Uma desvantagem em se utilizar este algoritmo são as limitações da computação quando se trata de uma busca em que se tem muitos nós, isto é, problemas de larga escala. Além do tempo de processamento, existe o problema da capacidade de armazenamento, visto que o algoritmo armazena os nós visitados

Explicação

A seguir, uma breve explicação de como funciona o algoritmo.

Aqui é criada a classe Node que irá representar os nós que possuem sua posição no mapa além dos custos f , g e h :

```
class Node:
    """
    A node class for A* Pathfinding
    """

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position
```

Esta função é utilizada para montar a lista de nós que compõem o melhor caminho encontrado pelo A*:

```
def return_path(current_node):

    path = []
    current = current_node
    while current is not None:
```

```
        path.append(current.position)
        current = current.parent
    return path[::-1]`
```

Para definir a heurística foi feita a seguinte função, que aplica a heurística de Manhattan:

```
def manhattan(point,point2):
    return abs(point.position[0] - point2.position[0]) + abs(point.position[1]-point2.position[0])
```

Chegamos de fato no algoritmo A*, ele recebe como parâmetros o mapa representado por uma matriz, o ponto inicial e o ponto final. Existe ainda um quarto parâmetro indicando se é permitido movimentos na diagonal:

```
def astar(maze, start, end, allow_diagonal_movement = False):
```

Os nós são inicializados, assim como a lista aberta e a lista fechada. É adicionado o primeiro nó a lista aberta.

```
# Create start and end node
start_node = Node(None, start)
start_node.g = start_node.h = start_node.f = 0
end_node = Node(None, end)
```

```
end_node.g = end_node.h = end_node.f = 0

# Initialize both open and closed list
open_list = []
closed_list = []

# Add the start node
open_list.append(start_node)
```

Neste trecho são definidas posições dos nós vizinhos a serem visitados, isto é, a posição dos nós ao redor:

```
adjacent_squares = ((0, -1), (0, 1), (-1, 0), (1, 0),)
```

```
if allow_diagonal_movement:
```

```
adjacent_squares = ((0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1),
(1, 1),)
```

Buscando o nó objetivo - Este loop percorre a lista aberta verificando o nó de menor custo e o adicionando na lista fechada. Após isso é feita a verificação se o nó em questão é o nó objetivo, se sim é retornada a lista dos nós que compõem o caminho, caso contrário o nó é expandido e é feito o cálculo dos custos de seus filhos, sendo o de menor custo adicionado na lista aberta e retornando ao início do loop.

```
# Loop until you find the end
while len(open_list) > 0:
    outer_iterations += 1
```

```

# Get the current node
current_node = open_list[0]
current_index = 0
for index, item in enumerate(open_list):
    if item.f < current_node.f:
        current_node = item
        current_index = index

# Pop current off open list, add to closed list
open_list.pop(current_index)
closed_list.append(current_node)

# Found the goal
if current_node == end_node:
    return return_path(current_node)

# Generate children
children = []

for new_position in adjacent_squares: # Adjacent squares

    # Get node position
    node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

    # Make sure within range
    if node_position[0] > (len(maze) - 1) or node_

```

```

position[0] < 0 or node_position[1] > (len(maze[len(maze)-
1])) -1) or node_position[1] < 0:
    continue

    # Make sure walkable terrain
    if maze[node_position[0]][node_position[1]] !=
0:
        continue

    # Create new node
    new_node = Node(current_node, node_position)

    # Append
    children.append(new_node)

    # Loop through children
    for child in children:

        # Child is on the closed list
        if len([closed_child for closed_child in close
d_list if closed_child == child]) > 0:
            continue

        # Create the f, g, and h values
        child.g = current_node.g + 1

        #Changed the Euclidean heuristics to the Manha
ttan Heuristics

```

```

        # child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] - end_node.position[1]) ** 2)

        child.h = manhattan(child, end_node)
        child.f = child.g + child.h

        # Child is already in the open list
        if len([open_node for open_node in open_list if child == open_node and child.g > open_node.g]) > 0:
            continue

        # Add the child to the open list
        open_list.append(child)

```

No *main* passamos o grid com 0 representando posições vazias e 1 representando obstáculos para o agente. Além disso passamos a tupla representando a posição inicial do agente (start), e a tupla representando a posição final desejada do agente (end), assim como a execução do algoritmo e a impressão da lista dos nós do melhor caminho a ser seguido.

```

def main():

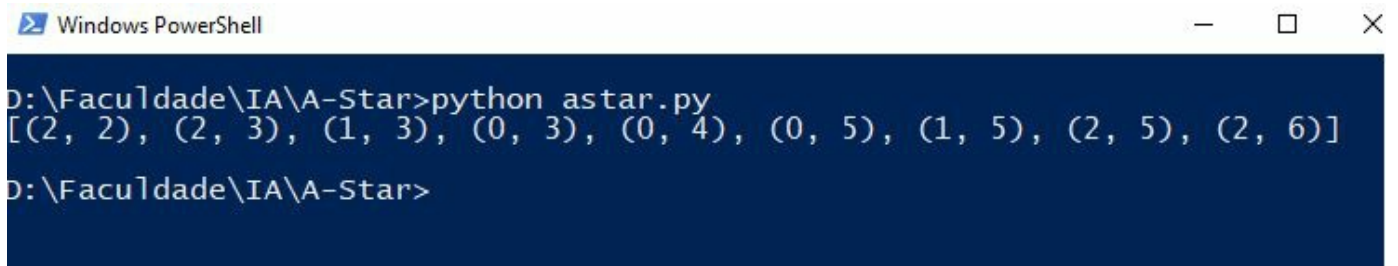
    maze = [[0,0,0,0,0,0,0,0],
            [0,0,0,0,1,0,0,0],
            [0,0,0,0,1,0,0,0],
            [0,0,0,0,1,0,0,0],

```

```
[0,0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0,0]]  
  
start = (2, 2)  
end = (2, 6)  
  
path = astar(maze, start, end)  
print(path)  
  
if __name__ == '__main__':  
    main()
```

Execução

Para a execução do algoritmo só é necessário invocar o interpretador python passando o arquivo [astar.py](#) como parâmetro de execução.



```
Windows PowerShell  
D:\Faculdade\IA\A-Star>python astar.py  
[(2, 2), (2, 3), (1, 3), (0, 3), (0, 4), (0, 5), (1, 5), (2, 5), (2, 6)]  
D:\Faculdade\IA\A-Star>
```

Referencia

[Código fonte](#)