Algoritmos de Busca em Python

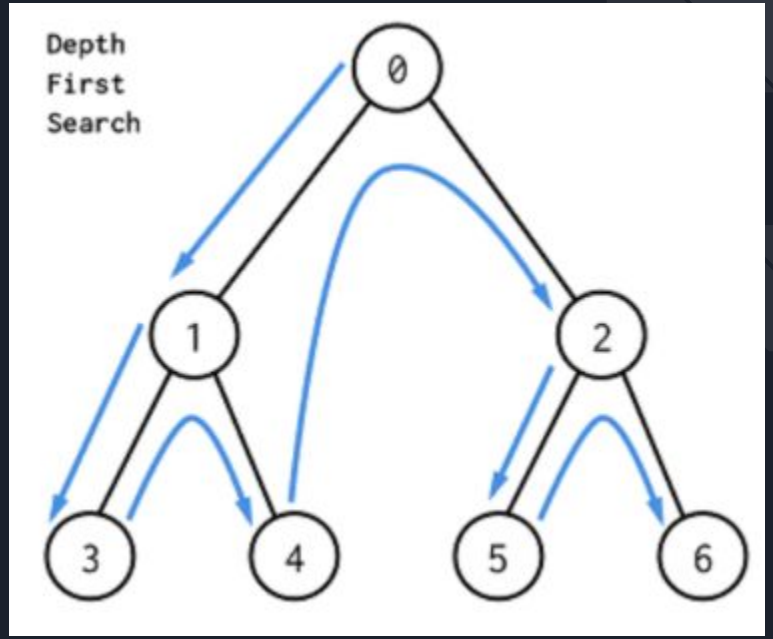
Algoritmo Busca em Profundidade

Vai até o último nó e depois voltando, utilizando de uma Pilha.

Dentre os algoritmos utilizados ele possui o maior tempo de execução, muitas vezes se tornando inviável.

Foca em grafos com diversas soluções.

Algoritmo Busca em Profundidade

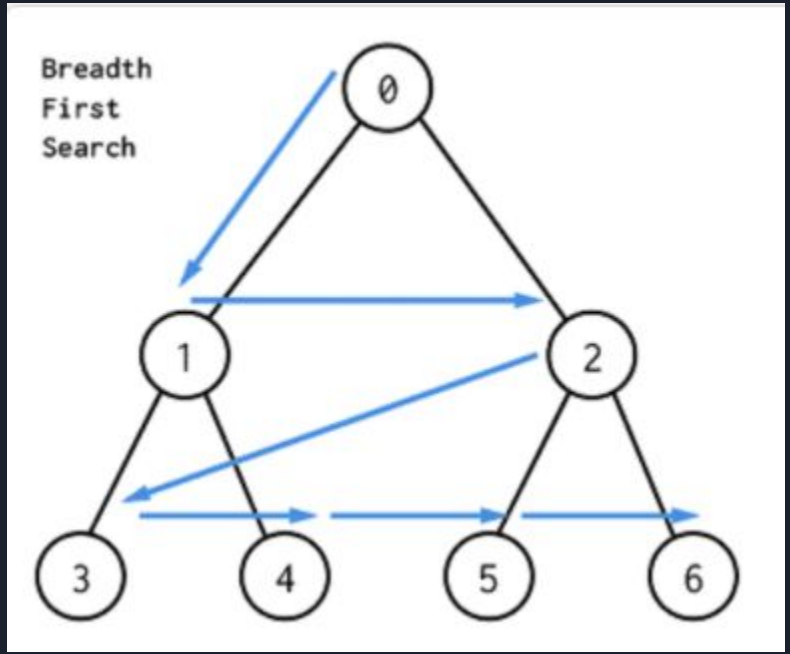


Algoritmo Busca em Largura

Avalia todos os vizinhos antes de prosseguir, e depois para cada vizinho olha seus vizinhos e assim por diante, sempre começando pelo primeiro vizinho encontrado. Utiliza de uma Fila para fazer isso.

Garante o menor caminho mas não o menor tempo de execução, seu uso vai depender do problema a ser solucionado.

Algoritmo Busca em Largura

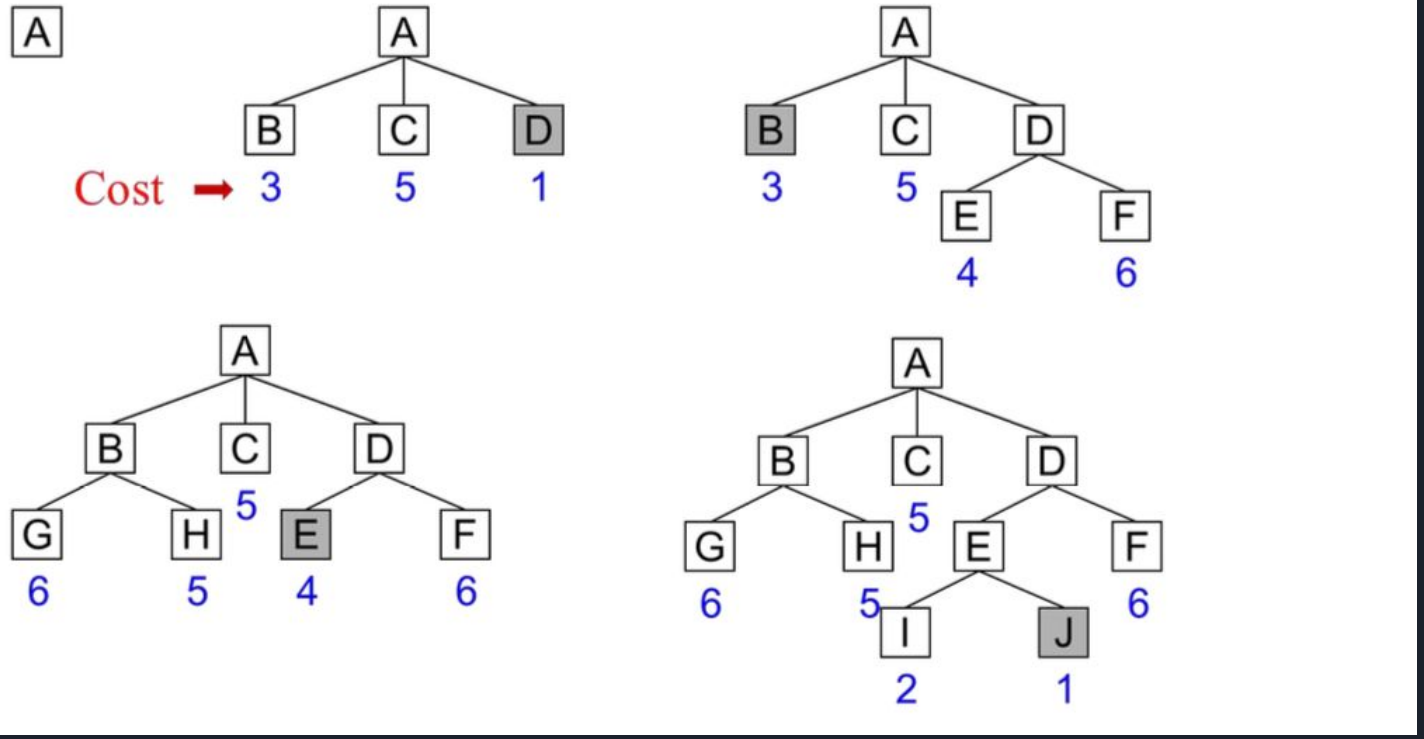


Algoritmo Best First

Similar ao algoritmo de busca em largura porém ele utiliza de uma heurística básica para decidir qual nó será avaliado primeiro. Essa heurística consiste em avaliar primeiro o nó com menor custo de locomoção.

Demonstrou ser mais rápido que o busca em largura mas com um caminho maior, vale ressaltar que estamos aplicando os algoritmos em um grafo knn, o que favorece os algoritmos que possuem heurísticas.

Algoritmo Best First



Algoritmo A

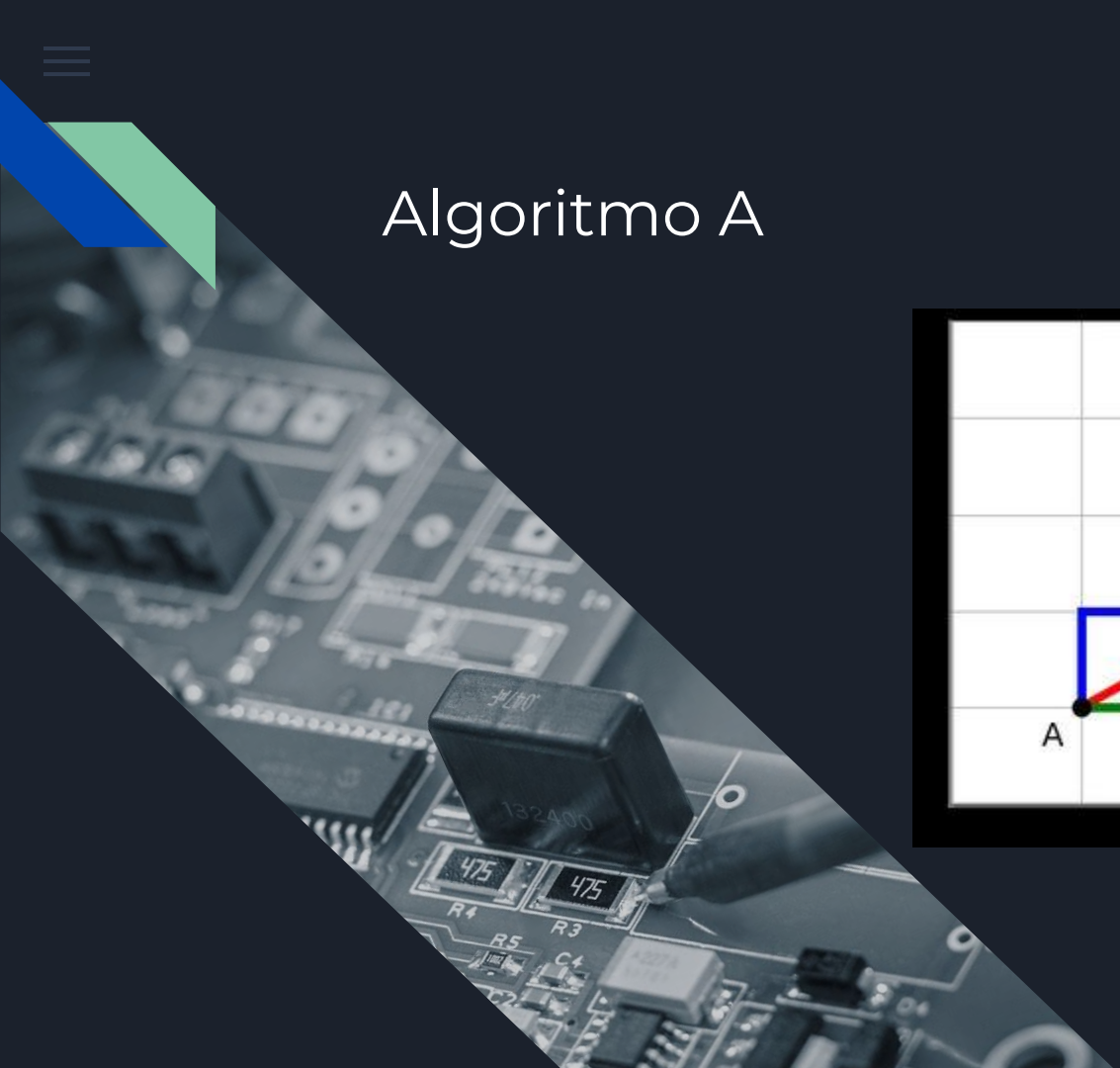
Uma melhoria da heurística do Best First

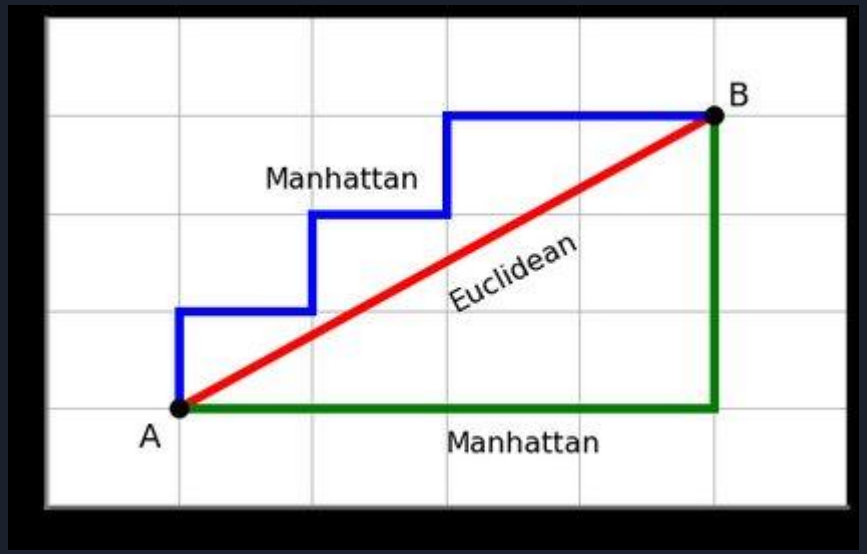
Ao invés de decidir o vizinho pelo custo, é escolhido o vizinho pela distância entre ele e o vértice objetivo.

Demonstrou ser mais rápido que os demais algoritmos já vistos e com um caminho menor que o Best First

Utilizamos a distância de Manhattan.

Algoritmo A





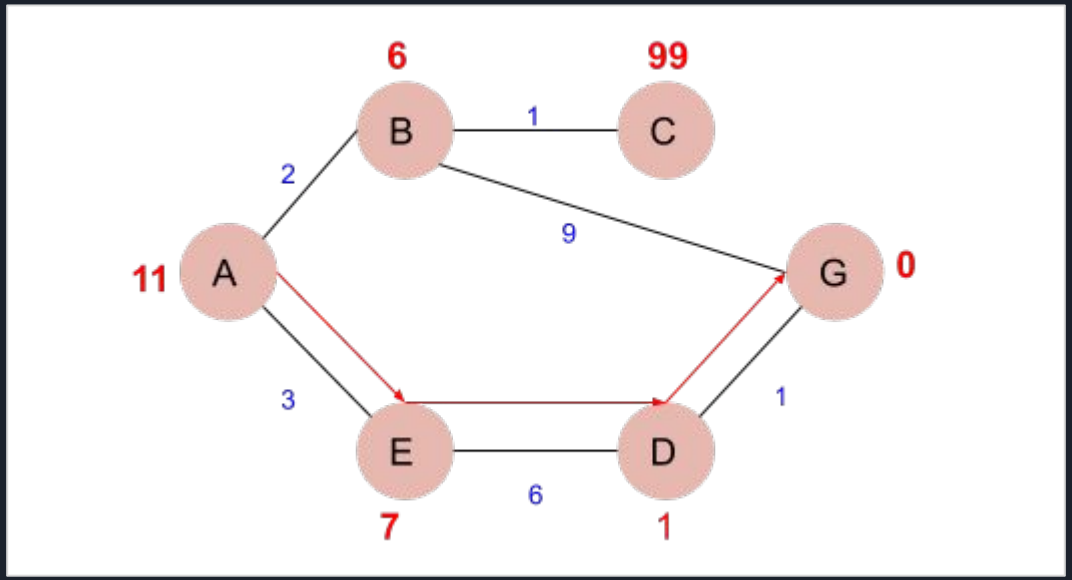
Algoritmo A*

O algoritmo mais rápido dentre os vistos, mas não necessariamente com o menor caminho.

O algoritmo está presente em GPS e jogos em uma grande quantidade

Utiliza a comparação da distância euclidiana entre o vértice atual com o vértice objetivo para tomada de decisão.

Algoritmo A*



Código

Grafo Knn

Para um grafo com 10 vértices e 3 vizinhos, é gerado uma lista de listas simulando uma matriz onde cada elemento Aij é a distância do vértice i ao vértice j, caso seja 0 eles não estão conectados.

```
Grafo knn gerado pela biblioteca:
[
 [0.0, 0.0, 0.0, 1.4142135623730951, 0.0, 0.0, 0.0, 1.0, 0.0, 3.0],
 [0.0, 0.0, 0.0, 0.0, 1.0, 3.1622776601683795, 0.0, 0.0, 0.0, 2.23606797749979],
 [0.0, 0.0, 0.0, 0.0, 0.0, 4.0, 1.0, 0.0, 3.1622776601683795, 0.0],
 [1.4142135623730951, 0.0, 0.0, 0.0, 0.0, 0.0,0.0, 1.0, 0.0, 2.23606797749979],
 [0.0, 1.0, 0.0, 4.123105625617661, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0],
 [0.0, 3.1622776601683795, 4.0, 0.0, 4.123105625617661, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 1.0, 0.0, 0.0, 5.0, 0.0, 0.0, 3.605551275463989, 0.0],
 [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.1622776601683795],
 [0.0, 0.0, 3.1622776601683795, 0.0, 0.0, 4.242640687119285, 3.605551275463989, 0.0, 0.0, 0.0],
 [0.0, 2.23606797749979, 0.0, 2.23606797749979, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0]
]
```

Grafo Knn

Modificando o grafo para facilitar a programação, agora temos uma lista de listas onde cada elemento da lista é uma lista contendo tuplas (a , b) onde a é o vértice ao qual o vértice do respectivo índice está conectado e b a distância entre eles

```
Grafo knn modificado para ser usado nos algortimos:
[
 [(3, 1.4142135623730951), (7, 1.0), (9, 3.0)],
 [(4, 1.0), (5, 3.1622776601683795), (9, 2.23606797749979)],
 [(5, 4.0), (6, 1.0), (8, 3.1622776601683795)],
 [(0, 1.4142135623730951), (7, 1.0), (9, 2.23606797749979)],
 [(1, 1.0), (3, 4.123105625617661), (9, 2.0)],
 [(1, 3.1622776601683795), (2, 4.0), (4, 4.123105625617661)],
 [(2, 1.0), (5, 5.0), (8, 3.605551275463989)],
 [(0, 1.0), (3, 1.0), (9, 3.1622776601683795)],
 [(2, 3.1622776601683795), (5, 4.242640687119285), (6, 3.605551275463989)],
 [(1, 2.23606797749979), (3, 2.23606797749979), (4, 2.0)]
]
```