Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Curso de Bacharelado em Sistemas de Informação

Algoritmos de Busca em Python para Grafos Knn

Heitor Camilo de Freitas e Oliveira
Pedro Fernando Christofoletti dos Santos

Monografia apresentada na disciplina SCC0630 - Inteligência Artificial, como requisito parcial para obtenção da aprovação na mesma.

São Carlos/SP, 06 de Julho de 2021

SUMÁRIO

Abstract: *The theme of this monograph is Artificial Intelligence and implementation of search algorithms with use of graphs. These algorithms will be introduced and explained in a way that is possible to measure their development, efficiency and compare them. They will use graphs to help represent the maps with some parameters. These algorithms were implemented in Python.*
Keywords: Search Algorithms, Graphs, Python.

Resumo: *O tema dessa monografia é Inteligência Artificial e implementação de algoritmos de busca com uso de grafos, esses algoritmos serão introduzidos e explicados, de forma que seja possível comparar a sua eficiência. Utilizarão os grafos como uma forma de representar os mapas com alguns parâmetros. Os algoritmos têm sua implementação feita em Python.*
Palavras-chave: Algoritmos de busca, Grafos, Python.

1. Introdução

A Inteligência Artificial está cada vez mais presente em diversas áreas, produtos e softwares que são muito utilizados todos os dias. Um software como o Google Maps, que é muito utilizado, precisa fazer cálculos para decidir a menor distância entre os pontos definidos pelo usuário, assim de acordo com cada ponto, as ruas mais próximas deverão ser avaliadas, da mesma forma como as demais, encontrando o melhor caminho. Cálculos como esse podem ser realizados por alguns algoritmos, sendo alguns deles a Busca em Profundidade, Busca em Largura, Busca Best First, algoritmo A e A*.

Os algoritmos implementados são comparados de diversas formas, de acordo com cada entrada pré-estabelecida, complexidade de tempo e espaço, tempo de execução e outros fatores. O KNN (K Nearest Neighbor) é um algoritmo usado no campo de Data Mining e Machine Learning baseado em distâncias e considera a proximidade entre dados para realizar previsões e classificações, como regressão. O KNN calcula a distância euclidiana entre os pontos e utiliza essa para classificá-los. O algoritmo pode ser usado para gerar gráficos expondo a distribuição dos dados, pode ser utilizado para distribuir dados que expõe a eficiência de outros algoritmos, por exemplo.

Os grafos são um conjunto de vértices e arcos, cada arco é associado a dois vértices, sendo um a ponta inicial e outro a final. Grafos são utilizados para representar esquemas nos quais os objetos possuem uma relação de peso, podendo ser de custo, tempo, distância, esses objetos podem ser cidades, pontos em um mapa, por exemplo.

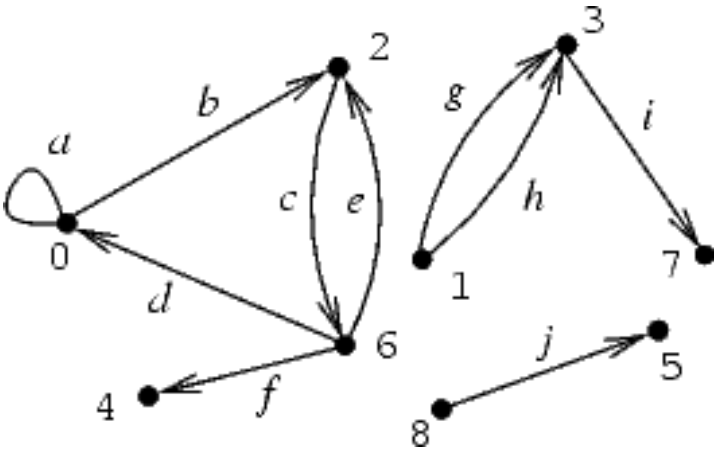


Figura 1 - Exemplo de grafo. Disponível em: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.ime.usp.br%2F~pf%2Falgoritmos_em_grafos%2Faulas%2Fgrafos.html&psig=AOvVaw0-7bLGSgQmEbsY0vwHbkxL&ust=1625593406339000&source=images&cd=vfe&ved=0CAoQjRxqFwoTCPic4L29zPECFQAAAAAdAAAAABAD

2. Descrição da implementação

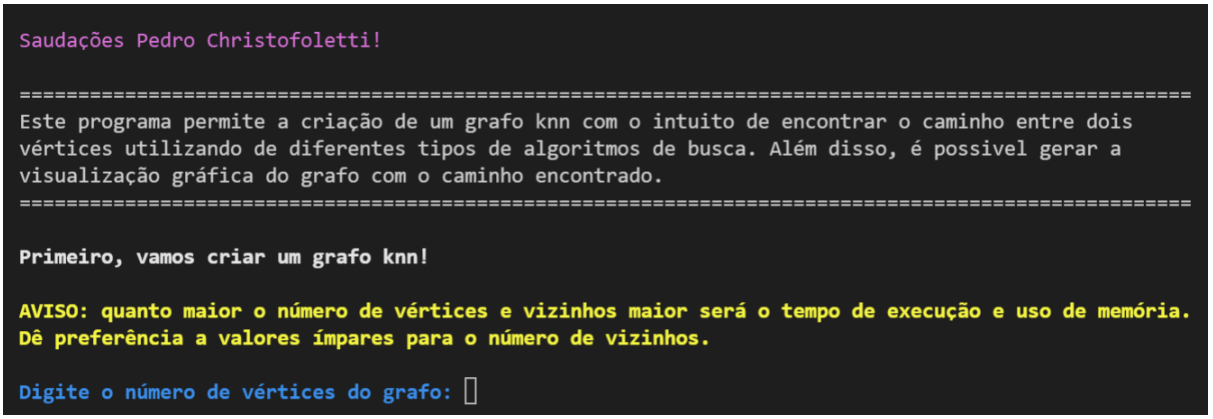
2.1. Bibliotecas usadas

Na implementação do projeto foram usadas as seguintes bibliotecas:

- **getpass**: para descobrir o nome de usuário do sistema e dar as saudações ao iniciar o programa.
- **copy**: para fazer uma cópia profunda do grafo knn.
- **numpy**: utilizado em vários momentos, para gerar arrays, calcular a distância euclidiana, gerar coordenadas aleatórias para os vértices do grafo knn.
- **sklearn**: gerar o grafo knn.
- **time**: calcular o tempo de execução.
- **matplotlib**: gerar a visualização do grafo e comparação dos tempos de execução.
- **queue**: para ordenar a fila usada nos algoritmos.

2.2. Passo a passo da execução

O primeiro passo para interagir com o programa é a criação do grafo knn, para isso é solicitado que o usuário informe o número de vértices a serem criados e o número de vizinhos que cada vértice terá.

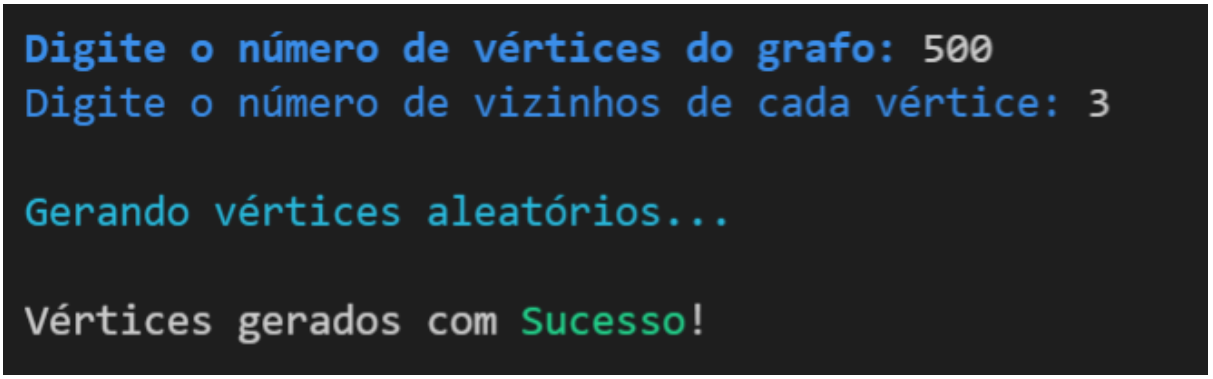


Após inseridas as informações necessárias para criar o grafo, utilizamos a função randint do numpy para gerar vértices aleatórios.

Código:

```
numberOfVertices = int(input(f'{bcolors.OKBLUE}Digite o número de vértices do grafo: {bcolors.ENDC}'))
k = int(input(f'{bcolors.OKBLUE}Digite o número de vizinhos de cada vértice: {bcolors.ENDC}'))

vertexArray=np.random.randint(numberOfVertices, size=(numberOfVertices, 2))
```

Saída:



Após gerados os vértices aleatórios, utilizamos a função kneighbors_graph da biblioteca sklearn para gerar o grafo knn.

Código:

```
knn_graph = kneighbors_graph(vertexArray, k, mode='distance')
knn_graph = knn_graph.toarray().tolist()
```

Saída:

```
Criando grafo knn...

Grafo knn criado com Sucesso!
```

O grafo criado fica no formato de uma matriz n x n, onde n é o número de vértices, e cada elemento X_{ij} contém a distância entre o vértice *i* ao vértice *j* caso sejam vizinhos, se não forem a distância é 0 por padrão. Utilizando dessa matriz será criada uma lista de listas contendo, onde cada índice da lista possui uma lista de vizinhos onde cada vizinho é uma tupla formada pelo par (v,c) onde v é o vértice e c é a distância.

Código:

```
for vertex in range(len(knn_graph)):
    for neighbour in range(len(knn_graph[vertex])):
        if knn_graph[vertex][neighbour]:
            knn_graph[vertex][neighbour] = (neighbour,knn_graph[vertex][neighbour])
        else:
            knn_graph[vertex][neighbour] = -1
    knn_graph[vertex] = list(filter(lambda v: v != -1, knn_graph[vertex]))
```

Após criada a lista de listas, é solicitado do usuário o vértice inicial e o vértice final para encontrar o caminho entre eles.

Código:

```
startVertex = int(input(f'{bcolors.OKBLUE}Vértice inicial: {bcolors.ENDC}'))
endVertex = int(input(f'{bcolors.OKBLUE}Vértice final: {bcolors.ENDC}'))
```

Saída:

```
Agora vamos tentar encontrar o caminho entre dois vértices!

Vértice inicial: 0
Vértice final: 300
```

Após escolha dos vértices, é solicitado que o usuário informe qual algoritmo de busca deseja que seja executado para encontrar o caminho, existe a possibilidade de que seja executado por todos.

Saída:

```
Agora escolha qual algoritmo para realizar a busca (número correspondente):

1- Busca em largura
2- Busca best first
3- Busca A
4- Busca A*
5- Busca em profundidade
6- Todos os anteriores

Opção:
```

Escolhida a opção, o algoritmo tentará encontrar o caminho. Caso encontre, mostrará o tamanho do caminho, o tempo de execução, o caminho, e dará a opção de visualizar o grafo com o caminho encontrado.

Código:

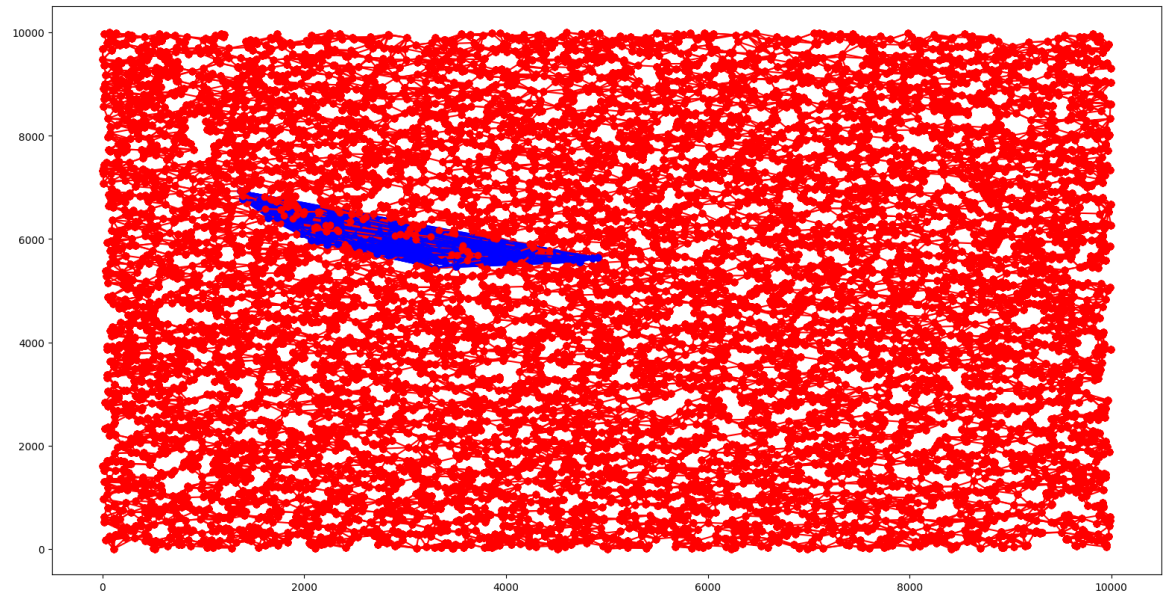
```python
def findPathAndInfo(option, knn_graph, startVertex, endVertex,
vertexArray):
    title = search_algorithms[option]['title']
    print(f'\n{bcolors.OKCYAN}Procurando caminho usando
{title}...{bcolors.ENDC}')
    start = time.time()
    path = search_algorithms[option]['func'](knn_graph, startVertex,
endVertex, vertexArray)
    stop = time.time()
    execTime = stop - start
    if path: # Path found
        print(f'\nCaminho {bcolors.OKGREEN}encontrado{bcolors.ENDC}!\n')
        print(f'{bcolors.HEADER}Tamanho do caminho:
{bcolors.ENDC}{len(path)}')
        print(f'{bcolors.HEADER}Tempo de execução:
{bcolors.ENDC}{execTime} segundos')
        print(f'{bcolors.HEADER}Caminho encontrado: \n{bcolors.ENDC}')
        print(f'{bcolors.OKCYAN}INICIO {bcolors.ENDC}-> ', end='')
        for vertex in path:
            print(f'{vertex} -> ', end='')
        print(f'{bcolors.OKCYAN}FIM{bcolors.ENDC}\n')
        return path
    else: # Path not found
        print(f'\nCaminho {bcolors.FAIL}não encontrado{bcolors.ENDC}.')
        return None
```

```python
def plotGraphWithPath(graph, dotsArray, path):
    print(f'{bcolors.OKCYAN}\nPlotando grafo knn...{bcolors.ENDC}')
    for x in range(len(dotsArray)):
        for y in range(len(dotsArray)):
            if x in path and y in path:
```

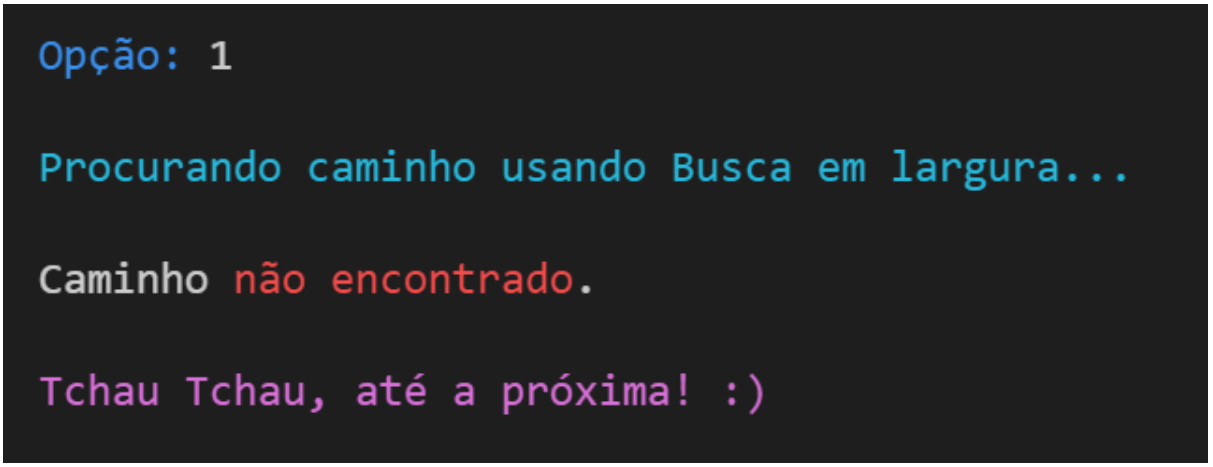
```
                plt.plot([dotsArray[x][0], dotsArray[y][0]],
[dotsArray[x][1], dotsArray[y][1]], 'bo-')
            elif graph[x][y]:
                plt.plot([dotsArray[x][0], dotsArray[y][0]],
[dotsArray[x][1], dotsArray[y][1]], 'ro-')
    plt.show()
```

Saída quando encontrado exemplo:

```
Opção: 1

Procurando caminho usando Busca em largura...

Caminho encontrado!

Tamanho do caminho: 35
Tempo de execução: 0.8250465393066406 segundos
Caminho encontrado:

INICIO -> 0 -> 1127 -> 1907 -> 2492 -> 4109 -> 941 -> 1682 -> 9216 -> 6137 ->
2963 -> 7520 -> 4563 -> 8276 -> 5329 -> 4394 -> 6849 -> 1575 -> 611 -> 1772 ->
 5897 -> 64 -> 6696 -> 3608 -> 1050 -> 1208 -> 2476 -> 2580 -> 4878 -> 4192 ->
 2866 -> 4146 -> 3955 -> 8670 -> 3625 -> 5668 -> FIM

Deseja visualizar o grafo com o caminho encontrado? (s/n): s

Plotando grafo knn...
```



Saída quando não encontrado exemplo:



3. Resultados

Os algoritmos de busca podem ser divididos entre informados e não informados. Os informados usam conhecimento sobre o domínio além do problema e heurísticas, enquanto os não informados somente usam a definição do problema. Entre os algoritmos não informados estão busca em largura, busca em profundidade e busca uniforme. Já entre os informados esta A*.

3.1 Busca em Largura

O algoritmo de busca em largura, também conhecido com breadth-first-search(BFS), utiliza grafos, onde são estabelecidos vértices e arestas. Um vértice principal primeiramente é escolhido, depois as arestas em contato com ele serão encontradas, levando aos vértices mais próximos, a uma distância k. Logo após, as arestas novas de cada vértice encontrado levará a outros vértices a uma distância k+1, isso se repetirá até que se ache todos os vértices e o grafo esteja completo.

Na busca em largura os vértices devem apresentar cores de acordo com sua classificação, os brancos não foram descobertos ainda, cinza são somente examinados enquanto os pretos foram examinados assim como seus vizinhos. Os vértices cinzas são guardados em uma fila, antes de entrar na fila são brancos e depois de sair são pretos.No pior dos casos, o algoritmo precisa percorrer todos os vértices e arestas, deixando um gasto superior de tempo. Sua complexidade de tempo é de O(V+A), onde V é a quantidade de vértices enfileirados e removidos uma vez e A são todas as arestas utilizadas, a complexidade de espaço é O(|V|).

A vantagem desse algoritmo é de que caso cada ação tenha custo idêntico, sua solução consumirá uma quantidade igual de recursos o algoritmo é considerado completo (encontrará o vértice desejado) e ótimo (encontrará a melhor sequência até o vértice desejado). Entretanto o algoritmo também deixa a desejar enquanto realiza uma busca incessante e para isso deve armazenar todos os vértices já visitados, evitando passar por eles novamente, o que pode complicar a memória RAM de acordo com o tamanho do problema.

Código:

```
def breadthFirstSearch(graph, startNode, endNode, vertexArray):
    explored = []
    queue = [[startNode]]
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour[0])
                queue.append(new_path)
                if neighbour[0] == endNode:
                    return new_path
            explored.append(node)
    return None
```

3.2 Busca em Profundidade

O algoritmo de busca em profundidade, também conhecido como depth-first-search(DFS), utiliza a ideia de sempre partir do vértice descoberto mais recentemente, até que não haja mais vizinhos. Nesse caso os vértices recebem 2 rótulos, d, que indica o momento no qual o vértice foi descoberto e tornou-se cinza, e f, que indica o momento em que seus vizinhos foram identificados e tornou-se preto. O vértice branco pode ser até d, cinza entre d e f e preto a partir de f.

Depois de identificar o vértice, seus adjacentes são encontrados, o tempo de visita é marcado e se passa para o próximo vértice. O processo continua até que todos os vértices atingíveis pelo primeiro vértice sejam encontrados. Por conta de descer um caminho muito grande e necessitar ficar indo e voltando nesse caminho, sua solução não é ótima. Suas complexidades de tempo e espaço são O(A+V) e O(bm), respectivamente, onde b é o número de caminhos alternativos e m é a profundidade máxima da árvore de busca.

O algoritmo mostra maior eficiência quando existem várias soluções para o problema ou se vários caminhos levam a uma solução. O mesmo já se mostra mais falho quando visita todos os vértices do grafo, aumentando fortemente seu tempo de busca.

Código:

```
def depthFirstSearch(graph, startVertex, endVertex, vertexArray):
    stack = [(startVertex, [startVertex])]
    while stack:
        (vertex, path) = stack.pop()
        for node in set(list(map(lambda x: x[0], graph[vertex]))) -
```

```
set(path):
            if node == endVertex:
                return path + [node]
            else:
                stack.append((node, path + [node]))
    return None
```

3.3 Busca Best First

O algoritmo best-first search(BFS), utiliza a distância em linha reta entre o vértice atual e os os vértices vizinhos. Acha soluções rapidamente, entretanto nem sempre é a melhor, mas pode melhorar com uma boa função heurística. Esse algoritmo tem algumas variantes, como A* . Sua função é de avaliação é dada por f(n) = h(n), sendo h(n) o custo estimado de n até o objetivo e f(n) o valor da função.

O BFS tem melhores resultados quando comparado com algoritmos não informados, afinal desde o início otimiza o custo entre os vértices. O algoritmo acaba ficando somente atrás de suas próprias variações, que utilizam mais dados das distâncias para realizar decisões mais inteligentes.

Código:

```
def bestFirstSearch(graph, startVertex, endVertex, vertexArray):
    visited = [False] * len(graph)
    pq = PriorityQueue()
    pq.put((0, startVertex))
    path = []
    while pq.empty() == False:
        u = pq.get()[1]
        path.append(u)
        if u == endVertex:
            return path
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    return None
```

3.4 Busca A*

Esse algoritmo surgiu em 1968 por Peter Hart, Nils Nilsson e Bertram Raphael, um pouco depois viabilizou o surgimento do GPS, através de seu funcionamento, e é bem utilizado em plataformas de jogos para que bots consigam traçar caminhos até o usuário. Aqui a ideia é identificar o vértice para onde se deseja traçar o caminho, identificar os vizinhos de onde iniciará o caminho,

calcular as distâncias em linha reto entre os vértices vizinhos e o vértice desejado e então utilizar uma combinação entre as distâncias início-vizinho e vizinho-final para decidir qual o vizinho se deve ir para obter o menor caminho, repetindo isso algumas vezes até chegar no vértice desejado.

Sua função de avaliação é dada por f(n) = g(n) + h(n), g(n) é o custo até o momento para alcançar n. A* apresenta soluções ótimas com heurística admissível. Todos os nós ficam na memória e a complexidade do espaço pode se tornar exponencial no pior caso, entretanto é um dos melhores algoritmos, sua complexidade de tempo já é muito boa, visto que utiliza a distância euclidiana para o vértice desejado e também a distância para o vértice vizinho para fazer a melhor escolha.

Código:

```
def aStarSearch(graph, startVertex, endVertex, vertexArray):
    visited = [False] * len(graph)
    pq = PriorityQueue()
    pq.put((0, startVertex))
    path = []
    while pq.empty() == False:
        u = pq.get()[1]
        path.append(u)
        if u == endVertex:
            return path
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                point1 = np.array(vertexArray[v])
                point2 = np.array(vertexArray[endVertex])
                euclideanDistance = np.linalg.norm(point1 - point2)
                pq.put((c + euclideanDistance, v))
    return None
```

3.5 Busca A

A busca A é um algoritmo semelhante ao best-first, mas quase igual ao A*. A maior diferença entre os dois algoritmos é a forma como calculam as distâncias, entre vértice vizinho e vértice desejado. Essa diferença no cálculo das distâncias deixa o A para trás, visto que a distância euclidiana é mais eficiente que a distância Manhattan, fazendo com que seu uso seja menor do que o de seu irmão.

Código:

```
def aSearch(graph, startVertex, endVertex, vertexArray):
    visited = [False] * len(graph)
    pq = PriorityQueue()
    pq.put((0, startVertex))
    path = []
```

```
    while pq.empty() == False:
        u = pq.get()[1]
        path.append(u)
        if u == endVertex:
            return path
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                point1 = np.array([1] + vertexArray[v])
                point2 = np.array([1] + vertexArray[endVertex])
                manhattanDistance = abs(point1[0] - point2[0]) +
abs(point2[1] - point2[1])
                pq.put((c + manhattanDistance, v))
    return None
```

k=3 e v=500

A*:
Tamanho do caminho: 8
Tempo: 0.0009984970092773438 segundos

A:
Tamanho do caminho: 8
Tempo: 0.0009963512420654297 segundos

Best-first:
Tamanho do caminho: 241
Tempo: 0.0009970664978027344 segundos

Busca em largura:
Tamanho do caminho: 4
Tempo: 0.0 segundos

Busca em profundidade
Tamanho do caminho: 7
Tempo: 0.0 segundos

k=5 e v=5000

A*:
Tamanho do caminho: 98
Tempo: 0.0029897689819335938 segundos

A:

Tamanho do caminho: 261
Tempo: 0.006017446517944336 segundos

Best-first:

Tamanho do caminho: 2889
Tempo: 0.014960527420043945 segundos

Busca em largura:

Tamanho do caminho: 75
Tempo: 0.5436265468597412 segundos

Busca em profundidade

Tamanho do caminho: indeterminado
Tempo: indeterminado

k=7 e v=10000

A*:

Tamanho do caminho: 87
Tempo: 0.011479377746582031 segundos

A:

Tamanho do caminho: 206
Tempo: 0.011479377746582031 segundos

Best-first:

Tamanho do caminho: 5947
Tempo: 0.05133962631225586 segundos

Busca em largura:

Tamanho do caminho: 62
Tempo: 1.9495022296905518 segundos

Busca em profundidade

Tamanho do caminho: indeterminado
Tempo: indeterminado

4. Discussão

Sem muito esforço é possível visualizar que o tempo gasto em buscas de profundidade são enormes, tanto que depois de vários minutos tive que abortar os testes para esse algoritmo, que desempenhou de maneira tremendamente ruim.

Fica claro no geral que os algoritmos informados tem vantagem e são mais rápidos que os outros. O algoritmo busca em largura consegue realizar as buscas percorrendo um caminho menor. Já o algoritmo A* apresentou o maior desempenho entre os demais algoritmos, atingindo os menores tempos no geral através do cálculo por distâncias euclidianas e mostrando eficiência mesmo em grafos de grande porte.

Referências bibliográficas

https://www.redblobgames.com/pathfinding/a-star/implementation.html

https://blog.pantuza.com/artigos/busca-em-profundidade

https://blog.pantuza.com/artigos/busca-em-largura

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html

https://www.ime.usp.br/~leliane/IAcurso2006/slides/Aula5-Astar-2006

https://algoritmosempython.com.br/cursos/algoritmos-python/algoritmos-grafos/busca-largura/

https://algoritmosempython.com.br/cursos/algoritmos-python/algoritmos-grafos/busca-profundidade/

https://www.annytab.com/best-first-search-algorithm-in-python/

www.dca.fee.unicamp.br

https://iascblog.wordpress.com/2015/11/02/buscas-em-largura-e-profundidade-aplicadas-a-inteligencia-artificial/#:~:text=A%20vantagem%20do%20algoritmo%20de,at%C3%A9%20o%20estado%2Dmeta
).