Analisis y Diseño de Algoritmos

Marco Antonio Bastida Flores

20 Junio 2023

1 Depth-first search

```
import random

nodes = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49)

Edges = [(0, 29), (0, 46), (0, 21), (0, 14), (0, 38), (0, 31), (1, 41), (1, 31),
(1, 21), (1, 17), (2, 9), (2, 26), (2, 5), (2, 25), (2, 4), (3, 18), (3, 30),
(3, 47), (4, 28), (4, 9), (4, 8), (5, 44), (5, 12), (6, 37), (6, 10), (7, 23),
(7, 22), (7, 39), (9, 19), (9, 28), (9, 27), (11, 33), (13, 25), (13, 38), (13, 29),
(14, 26), (14, 28), (14, 39), (15, 22), (15, 31), (15, 19), (15, 41), (16, 46),
(16, 26), (16, 38), (16, 27), (17, 40), (17, 29), (18, 45), (18, 42), (18, 35),
(18, 33), (18, 47), (20, 36), (20, 49), (20, 42), (22, 26), (22, 34), (23, 31),
(23, 32), (23, 40), (24, 31), (24, 44), (25, 38), (26, 31), (27, 32), (29, 48),
(29, 41), (30, 47), (30, 37), (33, 36), (33, 49), (34, 48), (35, 45), (36, 45),
(37, 49), (37, 45), (37, 47), (38, 41), (40, 48), (41, 44), (42, 49), (43, 48),
(45, 47)]

AD_List = {}
subGraph_A = []
subGraph_B = []

# Create key dictionary
for node in nodes:
    AD_List[node] = [None]

# Fill the dictionary with the edges
for edge in Edges:
    AD_List[edge[0]].insert(-1, edge[1])
    AD_List[edge[1]].insert(-1, edge[0])

# Print the Adjacent List to vizualize results.
#for key in AD_List:
    #print(key, ": ", AD_List[key], sep="")

def DFS(Graph, v, subGraph):
    # print("Edge called:", v)
    #Label v as discovered
    if v is not None:
```

```
        # print(subGraph)
        subGraph.append(v)
    else:
        return
    for sub_node in Graph[v]:
        if sub_node not in subGraph:
            DFS(Graph, sub_node, subGraph)

# Find the fisrt graph
iNode_1 = random.choice(nodes)
DFS(AD_List, iNode_1, subGraph_A)
print(subGraph_A)

# Finde the second graph
iNode_2 = random.choice(nodes)
while iNode_2 in subGraph_A:
    iNode_2 = random.choice(nodes)
DFS(AD_List, iNode_2, subGraph_B)
print(subGraph_B)
```

Se presenta arriba el codigo para el algoritmo de busqueda a profundidad, se hace uso de una lista ligada simple utilizando un diccionario de Python, que contiene como "key" el nodo y como "values" los nodos a los que apunta. Se selecciona un nodo aleatoriamente entre los elementos de los nodos para obtener el primer grafo no convexo. Con los nodos restantes se selecciona otro nodo aleatoriamente y se obtiene el segundo grafo no convexo.

1.1 Resultados

En la siguiente figura se muestran los resultados

```
= RESTART: C:\Users\chap1\OneDrive\Escritorio\Algorithms\Deep Frist Search\Code\DFS
.py
[18, 3, 30, 47, 37, 6, 10, 49, 20, 36, 33, 11, 45, 35, 42]
[41, 1, 31, 0, 29, 13, 25, 2, 9, 4, 28, 14, 26, 16, 46, 38, 27, 32, 23, 7, 22, 15,
19, 34, 48, 40, 17, 43, 39, 8, 5, 44, 24, 12, 21]
```

Figure 1: Resultados de busqueda por profundidad

El primer grafo no convexo es: [18, 3, 30, 47, 37, 6, 10, 49, 20, 36, 33, 11, 45, 35, 42]
El segundo grafo no convexo es: [41, 1, 31, 0, 29, 13, 25, 2, 9, 4, 28, 14, 26, 16, 46, 38, 27, 32, 23, 7, 22, 15, 19, 34, 48, 40, 17, 43, 39, 8, 5, 44, 24, 12, 21]

2 Dijkstra's algorithm

```python
Nodes = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)

Edges = [(0, 5, {'weight': 1}), (0, 7, {'weight': 9}), (0, 11, {'weight': 11}),
        (0, 16, {'weight': 11}), (0, 17, {'weight': 3}), (0, 18, {'weight': 9}),
        (1, 5, {'weight': 5}), (1, 7, {'weight': 1}), (1, 9, {'weight': 10}),
        (1, 15, {'weight': 1}), (1, 16, {'weight': 6}), (1, 19, {'weight': 12}),
        (2, 12, {'weight': 14}), (2, 16, {'weight': 4}), (2, 19, {'weight': 13}),
        (3, 7, {'weight': 5}), (3, 15, {'weight': 1}), (3, 16, {'weight': 10}),
        (3, 18, {'weight': 4}), (4, 7, {'weight': 3}), (4, 8, {'weight': 11}),
        (4, 11, {'weight': 12}), (4, 13, {'weight': 13}), (4, 16, {'weight': 9}),
        (4, 18, {'weight': 8}), (5, 7, {'weight': 2}), (5, 8, {'weight': 2}),
        (5, 9, {'weight': 13}), (5, 11, {'weight': 1}), (5, 14, {'weight': 12}),
        (6, 7, {'weight': 8}), (6, 10, {'weight': 6}), (6, 13, {'weight': 13}),
        (6, 15, {'weight': 5}), (6, 18, {'weight': 13}), (7, 8, {'weight': 2}),
        (7, 11, {'weight': 13}), (7, 16, {'weight': 4}), (7, 17, {'weight': 6}),
        (7, 19, {'weight': 7}), (8, 13, {'weight': 8}), (8, 14, {'weight': 10}),
        (8, 16, {'weight': 14}), (9, 16, {'weight': 9}), (10, 17, {'weight': 7}),
        (10, 19, {'weight': 5}), (11, 13, {'weight': 12}), (11, 14, {'weight': 13}),
        (11, 15, {'weight': 2}), (12, 13, {'weight': 9}), (12, 15, {'weight': 7}),
        (12, 17, {'weight': 8}), (13, 15, {'weight': 1}), (13, 18, {'weight': 9}),
        (13, 19, {'weight': 6}), (14, 18, {'weight': 9}), (15, 18, {'weight': 2}),
        (17, 18, {'weight': 14}), (17, 19, {'weight': 13})]

#Define a big enough number to distinguish between valid weights.
intmax = 999999999

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    def fill_Matrix(self):
        for edge in Edges:
            g.graph[ edge[0] ][ edge[1] ] = edge[2]["weight"]
            g.graph[ edge[1] ][ edge[0] ] = edge[2]["weight"]

    # Print the Adjacent Matrix to validate the correctness of the fillment.
    def printMatrix(self):
        for node in Nodes:
            print(self.graph[node])

    def printShortestDistances(self, distance, source):
        print("Vertex \tDistance from Source", source)
        for node in range(self.V):
            print(node, "\t", distance[node])

# Find the the minimum value in distance list that has not been visited yet
# Fill the visitedOrder list
    def minDistanceEdge(self, distance, visitedMarking, visitedOrder):
        min = intmax
```

```python
        min_index = None
        for vertix in range(self.V):
            if distance[vertix] < min and visitedMarking[vertix] == False:
                min = distance[vertix]
                min_index = vertix
        visitedOrder.append( min_index )
        # Return the vertix in which the minimum distance is found
        return min_index

# Dijkstra Algorithm
    def dijkstra_ALGA(self, source, destination):
        # List of infinite values of distance
        distance = [intmax] * self.V
        # Mark the starting point to be 0th vertix
        distance[source] = 0
        # List to mark the visited vertices
        visitedMarking = [False] * self.V
        # List of visited vertices.
        visitedOrder = []
        # Immediate source list
        immSrc = [0] * self.V

        path = []

        # Iterate over all the vertices
        for vertix in range(self.V):
            # Place the pointer at the vertix which has the current minimum distance
            pointer = self.minDistanceEdge(distance, visitedMarking, visitedOrder)
            # Mark the vertix with the minimum distance as visited
            visitedMarking[pointer] = True

            # Iterate over the distances
            for edge_weight in range(self.V):
                # If the distance is not zero
                if self.graph[pointer][edge_weight]:
                    # Calculate the overall distance from the current pointer
                    # to the edge
                    overall_distance = distance[pointer] + \
                                        self.graph[pointer][edge_weight]
                    # If the distance is less than the current distance
                    # Update the distance with the shortest
                    if visitedMarking[edge_weight] == False and \
                       distance[edge_weight] > overall_distance:
                        distance[edge_weight] = overall_distance
                        immSrc[edge_weight] =  pointer
                    #print(distance)
            #print(visitedMarking)
        # Print the results
        #self.printShortestDistances(distance, source)
        #print("Visited Order:",visitedOrder, sep="\n")
        #print("Immeadiate source:",immSrc, sep="\n")

        while( destination ):
            path.insert( 0, destination )
```

```
            destination = immSrc[destination]
        path.insert( 0, destination )
        print("Path of shortest distance from source to destination: ", path )

g = Graph(20)
g.fill_Matrix()
#g.printMatrix()
g.dijkstra_ALGA(0, 14)
```

En la siguiente figura se muesta el resultado del camino más corto desde 0 a 14.

```
= RESTART: C:\Users\chap1\OneDrive\Escritorio\Algorithms\Dijkstras\code\Dijkstra.py
Path of shortest distance from source to destination:  [0, 5, 14]
```

Figure 2: Camino más corto entre 0 y 14

Se presenta arriba el codigo para obtener el camino más corto entre el vertice 0 al vertice 14, se utiliza una matriz de adjacencia.
El camino más corto es 0 - 5 - 14 con un peso de 13.

Vertex	Distance from Source 0
0	0
1	4
2	11
3	5
4	6
5	1
6	9
7	3
8	3
9	14
10	10
11	2
12	11
13	5
14	13
15	4
16	7
17	3
18	6
19	10

Figure 3: Camino más corto entre 0 y v

View the Code on GitHub
Click to browse repository