Implementing Various Algorithms

ALGORITHM PROJECT

Ahmed Sherif Mohamed 20106620
AAST

TABLE OF CONTENTS

GREEDY ALGORITHMS

PRIM'S ALGORITHM

WHAT IS PRIM'S ALGORITHM ?

Prim's Algorithm is a greedy Algorithm to obtain the MST (minimum spanning tree) of a weighted undirected graph , MST is a tree that includes all the vertices of the graph, but only a subset of the edges that connect them with minimizing the total weight of the edges in the tree.

The algorithm starts with a single node and incrementally grows the MST by adding the minimum weight edge that connects a node in the MST to a node outside the MST. This process continues until all the nodes are included in the MST.

HERE'S HOW THE ALGORITHM WORKS :

- Choose any node to start from and add it to the MST.
- For each adjacent node that is not already in the MST, calculate the weight of the edge that connects it to the MST.
- Choose the node with the minimum edge weight and add it to the MST.
- Repeat steps 2 and 3 until all nodes are included in the MST.

PRIM'S ALGORITHM IMPLEMENTATION :

```
class Graph:
    def __init__(self,num_of_nodes):
        self.adjMatrix = []
        self.num_nodes = num_of_nodes
        for i in range(num_of_nodes):
            self.adjMatrix.append([-1 for i in range(num_of_nodes)])

    def addEdge(self,node1,node2,weight):
        if node1 == node2:
            print("Same Node")
            return False
        self.adjMatrix[node1][node2]=weight
        self.adjMatrix[node2][node1]=weight

    def getGraph(self):
        text=""
        for i in range(self.num_nodes):
            for j in range(i):
                if self.adjMatrix[i][j] >0:
                    text+=alphabet[j]+" --- " + str(self.adjMatrix[i][j])+" ---
"+alphabet[i]+"\n"
        return text
def prim(graph):
    if isinstance(graph,Graph):
        n_nodes = graph.__len__()
```

```
        visited_nodes = [False for i in range(n_nodes)]
        visited_nodes[0] = True
        edges = 0
        mst = Graph(n_nodes)

        while edges < n_nodes-1:
            node_x = 0
            node_y = 0
            min = inf
            for i in range(n_nodes):
                if visited_nodes[i]:
                    for j in range(n_nodes):
                        if graph.adjMatrix[i][j] > 0 and not visited_nodes[j]:
                            if graph.adjMatrix[i][j] < min:
                                min = graph.adjMatrix[i][j]
                                node_x, node_y = i, j

            mst.addEdge(node_x, node_y, graph.adjMatrix[node_x][node_y])
            edges += 1
            visited_nodes[node_y] = True
        return mst
    else:
        print("this is not a graph")
        return False
```

TESTING PRIM'S ALGORITHM ON THE EXAMPLE GRAPH :

```
graph = Graph(9)


graph.addEdge(0, 1, 4)
graph.addEdge(0, 2, 7)
graph.addEdge(1, 2, 11)
graph.addEdge(1, 3, 9)
graph.addEdge(1, 5, 20)
graph.addEdge(2, 5, 1)
graph.addEdge(3, 6, 6)
graph.addEdge(3, 4, 2)
graph.addEdge(4, 6, 10)
graph.addEdge(4, 8, 15)
graph.addEdge(4, 7, 5)
graph.addEdge(4, 5, 1)
graph.addEdge(5, 7, 3)
graph.addEdge(6, 8, 5)
graph.addEdge(7, 8, 12)
```

THE OUTPUT :

Prim's MST :
A --- 4 --- B
A --- 7 --- C
D --- 2 --- E
C --- 1 --- F
E --- 1 --- F
D --- 6 --- G
F --- 3 --- H
G --- 5 --- I

THE COMPLEXITY OF PRIM'S ALGORITHM

In this implementation, we used an adjacency matrix to represent the graph, so the time complexity of accessing the weight of an edge between two nodes is O(1). Therefore, the time complexity of the inner for loop is O(N).

The outer while loop runs for a maximum of N-1 times, since an MST of a graph with N nodes has N-1 edges.

Therefore, the overall time complexity of this implementation of Prim's algorithm is O(N^2)

KRUSKAL'S ALGORITHM

WHAT IS KRUSKAL'S ALGORITHM ?

Kruskal's algorithm is another algorithm used to find the minimum spanning tree (MST) of a undericted, weighted graph. Like Prim's algorithm, Kruskal's algorithm is a greedy algorithm that selects edges in increasing order of their weights. However, Kruskal's algorithm does not start with a single node but rather with a subtrees, each consisting of a 2 nodes. The algorithm then iteratively adds the minimum weight edge that connects two trees until all nodes are included in a single tree, which is the MST.

HERE'S HOW KRUSKAL'S ALGORITHM WORKS:

1. sort all edges by their weight in the ascending order
2. pick the edge with the smallest weight and try to add it to the tree. if it forms a cycle, skip that edge
3. repeat these steps until you have a connected tree that covers all nodes

KRUSKAL'S ALGORITHM IMPLEMENTATION :

```
    def getParent(self,parents,node):

        if parents[node] == node:

            return node

        else:

            return self.getParent(parents,parents[node])


    def Connect_Graph(self, parents, node_ranks ,node1,node2):
```

```
        node1_root= self.getParent(parents,node1)
        node2_root= self.getParent(parents,node2)
        if node_ranks[node1_root] > node_ranks[node2_root]:
            parents[node2_root]=node1_root
        elif node_ranks[node1_root] < node_ranks[node2_root]:
            parents[node1_root] = node2_root
        else :
            parents[node1_root] = node2_root
            node_ranks[node1_root]+=1


def kruskul(graph):
    if isinstance(graph,Graph):
        num_nodes = graph.__len__()
        parents = [i for i in range(num_nodes)]
        node_ranks = [0 for i in range(num_nodes)]
        edge_weights = sorted(graph.getEdges(),key=lambda item:item[2])
        edges=0
        i=0
        MST = Graph(num_nodes)
        while(edges < num_nodes-1):


            node1 , node2 , weight = edge_weights[i]
            i+=1
            root1=graph.getParent(parents,node1)
            root2=graph.getParent(parents,node2)
```

```
            if root1 != root2:
                graph.Connect_Graph(parents,node_ranks,root1,root2)
                MST.addEdge(node1,node2,weight)
                edges+=1

        return MST

    else:
        print("this is not a graph")
        return False
```

THE OUTPUT:

Kruskal's MST :

A --- 4 --- B
A --- 7 --- C
D --- 2 --- E
C --- 1 --- F
E --- 1 --- F
D --- 6 --- G
F --- 3 --- H
G --- 5 --- I

THE COMPLEXITY OF KRUSKAL'S ALGORITHM :

the time complexity of this implementation of Kruskal's algorithm is O(E log E), where E is the number of edges in the graph. The sorting of the edges takes O(E log E) time, and the while loop iterates over at most E edges and performs O(log N) for getParent and Connect_Graph function, where N is the number of nodes in the graph. Therefore, the overall time complexity is O(E log E + E log N) = O(E log E).

DIJKSTRA'S ALGORITHM

WHAT IS DIJKSTRA'S ALGORITHM ?

Dijkstra's algorithm is a shortest path algorithm that finds the shortest path between a source node and all other nodes in a weighted graph .

HERE'S HOW DIJKSTRA'S ALGORITHM WORKS:

1. Initialize the distance to the source node as 0, and the distance to all other nodes as infinity. Add all nodes to the unvisited set.
2. While the unvisited set is not empty, select the node with the smallest distance from the source node, and mark it as visited.
3. For each unvisited neighbor of the selected node, calculate the distance from the source node to the neighbor through the selected node. If this distance is less than the current distance to the neighbor, update the distance to the neighbor.
4. Repeat steps 2 and 3 until all nodes have been visited.

DIJKSTRA'S ALGORITHM IMPLEMENTATION:

```
def dijkstra(graph,start_node):
    if isinstance(graph,Graph):
        visited=[]
        distences={i:inf for i in range(graph.num_nodes)}
        distences[start_node]=0
        pq = PQ()
        pq.put((start_node,0))
        while not pq.empty():
            curr_node,curr_dist=pq.get()
            visited.append(curr_node)
            for node in range(graph.num_nodes):
                if graph.adjMatrix[curr_node][node]!=-1 and node not in visited:
                    old_cost = distences[node]
```

```
                    new_cost = distences[curr_node]+graph.adjMatrix[curr_node][node]
                    if new_cost < old_cost:
                        distences[node]=new_cost
                        pq.put((node,new_cost))
        return distences
```

Now let's try to run this Algorithm on the same graph we ran on the prim's and Kruskal's algorithms

```
print("Dijkstra's Shortest Path from Node "+alphabet[0]+" : \n")


dict = dijkstra(graph, 0)
for i in dict :
    print(alphabet[i]+" "+str(dict[i]))
```

THE OUTPUT:

Dijkstra's Shortest Path from Node A :

A 0
B 4
C 7
D 13
E 15
F 8
G 19
H 11
I 23

THE COMPLEXITY OF DIJKSTRA'S ALGORITHM :

The time complexity of this implementation of Dijkstra's algorithm is O((E+N) log N),This is because the algorithm visits each node at most once, and for each node, it examines all of its neighbors. The cost of selecting the node with the smallest distance is O(log N) with the priority queue, and this operation is performed at most N times. The time complexity of updating the distances is O(log N) with the priority queue, and this operation is performed at most E times.Therefore, the overall time complexity is O((E+N) log N).

HUFFMAN CODING

WHAT IS HUFFMAN CODING?

Huffman Coding is a method of compressing data to minimize its size while keeping all of its data. It was developed by David Huffman and is often beneficial for compressing data with frequently occurring characters.

The Huffman coding algorithm is a greedy algorithm that constructs a binary tree of least weight from a set of symbols in order to minimise the weighted path length from the root to each symbol.

HERE'S HOW HUFFMAN CODING ALGORITHM WORKS*:*

1. Initialize a list of nodes, each containing a symbol and its weight.

2. While the list has more than one node, do the following:

   a. Sort the list by node weight in increasing order.
   b. Combine the two smallest-weight nodes into a new node with weight equal to the sum of their weights. Make the two nodes its left and right children.
   c. Add the new node to the list.
   4. The last node in the list is the root of the Huffman tree.

HUFFMAN CODING ALGORITHM IMPLEMENTATION :

```
class Node():



    def __init__(self,symbol,weight):

        self.symbol=symbol

        self.weight=weight

        self.left=None
```

```
        self.right=None


    def __str__(self):
        return self.symbol+" "+str(self.weight)


    def isLeaf(self):
        return self.left is None and self.right is None




def huffman_tree(symbol_weights):


    if symbol_weights is None:
        return
    nodes = [Node(symbol, weight) for symbol, weight in symbol_weights.items()]
    while len(nodes) > 1:
        nodes.sort(key=lambda item: item.weight)
        node1 = nodes[0]
        node2 = nodes[1]
        new_node = Node("", node1.weight+node2.weight)
        new_node.left = node1
        new_node.right = node2
        nodes.pop(0)
        nodes.pop(0)
        nodes.append(new_node)
```

```
    return nodes[0]

def encode(node,prefix="",code={}):
    if node is None:
        return
    if node.isLeaf():
        code[node.symbol]=prefix
        return


    encode(node.left,prefix+"0",code)
    encode(node.right,prefix+"1",code)


def huffman_encode(text):
    symbol_weights = {}
    for char in text:
        symbol_weights[char] = symbol_weights.get(char, 0) + 1
    root = huffman_tree(symbol_weights)
    if root is None:
        return
    dict_code = {}
    encode(root, code=dict_code)
    encoded_text = "".join([dict_code[symbol] for symbol in text])
    return encoded_text , dict_code


def huffman_decode(encoded_text, code_dict):
    code = ""
    decoded_text = ""
```

```
    for i in encoded_text:
        code += i
        for j in code_dict:
            if code_dict[j] == code:
                decoded_text += j
                code = ""
    return decoded_text
```

test the code :

```
text = str(input())
encoded_text,dict_code = huffman_encode(text)
decoded_text=huffman_decode(encoded_text,dict_code)

print("Orginal Text : "+text)
print("Encoded Text : "+encoded_text)
print("Dictionary Code : "+str(dict_code))
print("Decoded Text : "+decoded_text)
```

THE OUTPUT :

Orginal Text : Why be greedy?
Encoded Text : 10101011011100110000100110111100000111101 1010
Dictionary Code : {'e': '00', '?': '010', 'y': '011', ' ': '100', 'W': '1010', 'h': '1011', 'b': '1100', 'g': '1101', 'r': '1110', 'd': '1111'}
Decoded Text : Why be greedy?

THE COMPLEXITY OF HUFFMAN CODING ALGORITHM :

The complexity will depend on N the number of characters in the input text and the Huffman_tree function which has the worst complexity of all the other functions, The algorithm first creates a list of nodes for each symbol and weight, which takes O(N) time. Then, the algorithm enters a loop that runs n

times, where it selects the two smallest nodes and combines them into a new node. Sorting the list of nodes takes O(N log N) time, and combining two nodes takes O(1) time. Therefore, this function takes O(N^2 log N) time.

Therefore, the overall complexity of Algorithm is O(N^2 log N) as the huffman_tree function provides an upper bound on the running time of the algorithm.

PIN PACKING PROBLEM

WHAT IS PIN PACKING PROBLEM ?

Bin packing is a optimization problem that involves packing a set of items of varying sizes into a minimum number of bins of fixed capacity. The goal is to minimize the number of bins used while ensuring that each item is allocated to a bin and the total size of the items in each bin does not exceed the size of the bin.

There are several algorithms for solving the bin packing problem, including exact algorithms based on branch-and-bound, dynamic programming, and integer linear programming, as well as greedy algorithms such as first-fit, best-fit, next-fit, and genetic algorithms. These algorithms typically aim to find a feasible solution that uses a small number of bins or a near-optimal solution that uses a minimum number of bins.

We will use first fit greedy algorithm to solve this problem, but there are 2 types of this algorithm online first fit , which the input bins are unknown and this could major problems with all online algorithms is that it is hard to pack large items, especially when they occur late in the input.

The other type is offline first fit, which the input bins is known and stored, so we can sort them in decreasing order to pack the large items first.

ONLINE FIRST FIT

```
def first_fit(bins, pack_size):
    packs = [[]]
    for bin in bins:
        pack_flag = False
        for pack in packs:
            if sum(pack) + bin <= pack_size:
                pack.append(bin)
```

```
                pack_flag = True
                break
        if not pack_flag:
            packs.append([bin])
    return packs
```

test Code

```
bins = [0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6]
packs = first_fit(bins, 1)

print("Packs Requierd : "+str(len(packs)))
for i in range(len(packs)):
    print("Pack "+str(i+1)+" : "+str(packs[i]))
```

OUTPUT :

Packs Requierd : 5
Pack 1 : [0.5, 0.5]
Pack 2 : [0.7, 0.2, 0.1]
Pack 3 : [0.4, 0.2]
Pack 4 : [0.5]
Pack 5 : [0.6]

THE COMPLEXITY OF ONLINE FIRST FIT :

The complexity of the First Fit algorithm implemented in the first_fit function is O(N*M), where N is the number of bins and M is the number of packs.

OFFLINE FIRST FIT:

```
def offline_first_fit(bins, pack_size):
    packs = [[]]
    for bin in sorted(bins, reverse=True):
        for pack in packs:
            if sum(pack) + bin <= pack_size:
                pack.append(bin)
                break
        else:
            packs.append([bin])
    return packs
```

TEST CODE

Packs Requierd : 4
Pack 1 : [0.7, 0.2, 0.1]
Pack 2 : [0.6, 0.4]
Pack 3 : [0.5, 0.5]
Pack 4 : [0.5, 0.2]

Compared to the Online First Fit algorithm implemented in the first_fit function, the Offline First Fit algorithm implemented in the offline_first_fit function can use fewer packs for the same input list of bins, resulting in a more efficient packing. However, it requires that all the bins be known in advance.

THE COMPLEXITY OF ONLINE FIRST FIT :

The time complexity of the Offline First Fit algorithm implemented in the offline_first_fit function is O(N log N), where N is the number of bins. This is because the algorithm first sorts the input list of bins in decreasing order of size, which takes O(nlog n) time, then the algorithm processes each item and allocating it to the first bin that has enough space to store it takes O(N*M) time. where M is the number of bins needed. Therefore, the overall time complexity of the algorithm is dominated by the time needed to sort the input list of bins O(N log N + N*M).

DYNAMIC PROGRAMMING

FLOYD-WARSHALL ALGORITHM

WHAT IS FLOYD-WARSHALL ALGORITHM ?

The Floyd-Warshall algorithm is a well-known algorithm for finding the shortest path between all pairs of vertices in a weighted directed graph. The algorithm works for both positive and negative edge weights, but it does not work for graphs with negative cycles.

The Floyd-Warshall algorithm is based on dynamic programming and computes the shortest path between all pairs of vertices by iteratively considering all possible intermediate vertices. The algorithm maintains a matrix A of size n x n, where n is the number of nodes in the graph, and A[i][j] represents the length of the shortest path from node i to node j.

FLOYD-WARSHALL ALGORITHM IMPLEMENTATION

```
class Graph:

    def __init__(self,num_nodes):

        self.adjMatrix = []

        self.num_nodes= num_nodes

        for i in range(num_nodes):

            self.adjMatrix.append([inf for i in range(num_nodes)])


    def setAdjMatrix(self,adjMatrix):

        self.adjMatrix=adjMatrix


    def getAdjMatrix(self):

        return self.adjMatrix
def floyd(graph):

    if isinstance(graph,Graph):

        n = graph.num_nodes
```

```
        dist_edges=graph.getAdjMatrix()
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist_edges[i][j] = min(dist_edges[i][j], dist_edges[i][k] +
dist_edges[k][j])
        printMatrix(dist_edges)
        return(dist_edges)


def printMatrix(matrix):
    txt="   "
    for i in range(len(matrix)):
       txt+=alphabet[i]+"  "
    print(txt)
    for i in range(len(matrix)):
        print(alphabet[i]+" "+str(matrix[i]))
```

NOW LET US TEST THE CODE WITH THE FOLLOWING MATRIX :

```
graph = Graph(4)
adjmatrix = [[0, 5, inf, 10],
         [inf, 0, 3, inf],
         [inf, inf, 0,   1],
         [inf, inf, inf, 0] ]

graph.setAdjMatrix(adjmatrix)
matrix = floyd(graph)
```

OUTPUT

```
  A  B  C  D
A [0, 5, 8, 9]
B [inf, 0, 3, 4]
C [inf, inf, 0, 1]
D [inf, inf, inf, 0]
```

FLOYD-WARSHALL ALGORITHM COMPLEXITY

The time complexity of the Floyd-Warshall algorithm implemented in the floyd function is O(N^3), where N is the number of nodes in the input graph.

REFERENCES

https://favtutor.com/blogs/prims-algorithm-python

https://www.javatpoint.com/implementation-of-kruskals-algorithm-in-python

https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/minimum-spanning-trees-kruskals-algorithm/

https://www.udacity.com/blog/2021/10/implementing-dijkstras-algorithm-in-python.html

https://favtutor.com/blogs/huffman-coding

https://www.scaler.com/topics/data-structures/disjoint-set/

https://www.programiz.com/dsa/floyd-warshall-algorithm

https://www.programiz.com/dsa/priority-queue