Algorithm CW 2

December 2, 2024

[ ]:
```
#Q1(a)
```

[3]:
```
import numpy as np
from collections import deque

class GraphNode:
    def __init__(self, name):
        self.name = name
        self.adjacency_list = []

    def __lt__(self, other):
        return True

    def __le__(self, other):
        return True

class GraphEdge:
    def __init__(self, from_node, to_node):
        self.from_node = from_node
        self.to_node = to_node

class Graph:
    def __init__(self):
        self.num_nodes = 0
        self.num_edges = 0
        self.node_array = []
        self.node_dictionary = {}

    def add_node(self, name):
        new_node = GraphNode(name)
        self.node_array.append(new_node)
        self.node_dictionary[name] = new_node
        self.num_nodes += 1

    def print_nodes(self):
        if self.num_nodes == 0:
            print("Empty graph")
        for i in range(self.num_nodes):
```

```python
            print(self.node_array[i].name)

    def add_edge_by_node(self, from_node, to_node):
        from_node.adjacency_list.append(GraphEdge(from_node, to_node))
        self.num_edges += 1

    def add_edge(self, from_name, to_name):
        from_node = self.node_dictionary[from_name]
        to_node = self.node_dictionary[to_name]
        self.add_edge_by_node(from_node, to_node)

    def print_graph(self):
        if self.num_nodes == 0:
            print("Empty graph")
        print("The graph structure is:")
        for i in range(self.num_nodes):
            node = self.node_array[i]
            print("{node} : {direct_descendants}".format(
                node=node.name,
                direct_descendants=[edge.to_node.name for edge in node.
 ↪adjacency_list]
            ))
```

[4]:
```python
import numpy as np

def graph_from_adjacency_matrix(names_array: list, adjacency_matrix: np.array):
    n = len(names_array)
    if adjacency_matrix.shape != (n, n):
        print("Incongruent inputs")
        return None

    output_graph = Graph()

    for name in names_array:
        output_graph.add_node(str(name))

    for i in range(len(names_array)):
        for j in range(len(names_array)):
            if adjacency_matrix[i, j] > 0:
                output_graph.add_edge(str(names_array[i]), str(names_array[j]))

    return output_graph
```

[5]:
```python
def bfs_reachable_nodes(input_graph, initial_node_name):
    initial_node = None
    for node in input_graph.node_array:
        if node.name == initial_node_name:
```

```python
            initial_node = node
            node.visited = True
        else:
            node.visited = False

    if initial_node is None:
        print(f"Start node {initial_node_name} not found.")
        return None

    list_of_reachable_nodes = [initial_node]
    node_queue = deque([initial_node])

    while node_queue:
        current_node = node_queue.popleft()
        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            if not neighbor.visited:
                neighbor.visited = True
                list_of_reachable_nodes.append(neighbor)
                node_queue.append(neighbor)

    return list_of_reachable_nodes
```

[6]:
```python
np.random.seed(2024)
num_tests = 10
num_nodes = 10
names_v = [chr(ord('A') + i) for i in range(num_nodes)]

for i in range(num_tests):
    adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.8, 0.
 ↪2])
    G = graph_from_adjacency_matrix(names_v, adj_mat)
    reachable_nodes = [node.name for node in bfs_reachable_nodes(G, "A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.
 ↪join(reachable_nodes)}")
```

```
There are 1 reachable nodes: A
There are 9 reachable nodes: A, B, C, D, E, H, F, G, J
There are 8 reachable nodes: A, E, F, G, I, C, B, H
There are 2 reachable nodes: A, E
There are 9 reachable nodes: A, E, J, H, D, F, B, I, C
There are 7 reachable nodes: A, E, H, I, G, J, F
There are 1 reachable nodes: A
There are 7 reachable nodes: A, F, H, J, B, G, I
There are 1 reachable nodes: A
There are 10 reachable nodes: A, B, C, F, G, H, E, J, I, D
```

[ ]:
```
#Q1(b) finding a shortest path in an unweighted graph.
```

[ ]:
```

```

[ ]:
```
#Q2(a)
```

[7]:
```
def dfs_reachable_nodes(input_graph, initial_node_name):
    initial_node = None
    for node in input_graph.node_array:
        if node.name == initial_node_name:
            initial_node = node
            node.visited = True
        else:
            node.visited = False

    if initial_node is None:
        print(f"Start node {initial_node_name} not found.")
        return None

    list_of_reachable_nodes = []

    def dfs(node):
        list_of_reachable_nodes.append(node)
        for edge in node.adjacency_list:
            neighbor = edge.to_node
            if not neighbor.visited:
                neighbor.visited = True
                dfs(neighbor)

    dfs(initial_node)

    return list_of_reachable_nodes
```

[8]:
```
np.random.seed(2024)
num_tests = 10
num_nodes = 10
names_v = [chr(ord('A') + i) for i in range(num_nodes)]

for i in range(num_tests):
    adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.8, 0.
↪2])
    G = graph_from_adjacency_matrix(names_v, adj_mat)
    reachable_nodes = [node.name for node in dfs_reachable_nodes(G, "A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.
↪join(reachable_nodes)}")
```

```
There are 1 reachable nodes: A
There are 9 reachable nodes: A, B, C, D, F, G, H, J, E
```

```
There are 8 reachable nodes: A, E, G, B, C, F, H, I
There are 2 reachable nodes: A, E
There are 9 reachable nodes: A, E, H, B, F, I, C, J, D
There are 7 reachable nodes: A, E, G, F, J, H, I
There are 1 reachable nodes: A
There are 7 reachable nodes: A, F, H, B, J, G, I
There are 1 reachable nodes: A
There are 10 reachable nodes: A, B, C, G, F, H, E, I, D, J
```

[ ]:
```
#Q2(b)
```

[9]:
```
def topological_sort(input_graph):
    in_degree = {node.name: 0 for node in input_graph.node_array}
    for node in input_graph.node_array:
        for edge in node.adjacency_list:
            in_degree[edge.to_node.name] += 1

    zero_in_degree_queue = deque([node for node in input_graph.node_array if␣
 ↪in_degree[node.name] == 0])
    topological_order = []

    while zero_in_degree_queue:
        current_node = zero_in_degree_queue.popleft()
        topological_order.append(current_node)

        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            in_degree[neighbor.name] -= 1
            if in_degree[neighbor.name] == 0:
                zero_in_degree_queue.append(neighbor)

    if len(topological_order) != input_graph.num_nodes:
        print("The graph contains a cycle and is not a DAG.")
        return None

    return topological_order
```

[10]:
```
np.random.seed(2024)
for j in range(6, 30, 4):
    names_v = [chr(ord('A') + i) for i in range(j)]
    np.random.shuffle(names_v)
    adj_mat = np.triu(np.ones((j, j)), 1)
    G = graph_from_adjacency_matrix(names_v, adj_mat)
    np.random.shuffle(G.node_array)
    top_order = topological_sort(G)
    if top_order:
        print(*(node.name for node in top_order), sep=',')
```

```
```

```
E,B,D,F,C,A
G,H,J,I,B,E,F,A,C,D
J,K,I,C,A,N,B,E,M,D,F,H,G,L
D,C,M,E,L,F,Q,J,K,B,P,A,R,O,N,I,H,G
H,S,T,O,N,V,C,I,E,K,D,R,A,M,J,U,B,G,Q,F,P,L
L,M,A,I,D,Q,V,B,W,S,G,N,C,Y,R,J,T,H,K,E,X,F,O,Z,P,U
```

[ ]:
```
#Q2(c)
```

[11]:
```
def topological_sort_DAG_check(input_graph):
    in_degree = {node.name: 0 for node in input_graph.node_array}
    for node in input_graph.node_array:
        for edge in node.adjacency_list:
            in_degree[edge.to_node.name] += 1

    zero_in_degree_queue = deque([node for node in input_graph.node_array if␣
↪in_degree[node.name] == 0])

    topological_order = []

    while zero_in_degree_queue:
        current_node = zero_in_degree_queue.popleft()
        topological_order.append(current_node)

        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            in_degree[neighbor.name] -= 1
            if in_degree[neighbor.name] == 0:
                zero_in_degree_queue.append(neighbor)

    if len(topological_order) != input_graph.num_nodes:
        return False

    return topological_order
```

[12]:
```
np.random.seed(2024)
for j in range(9, 30):
    names_v = [chr(ord('A') + i) for i in range(j)]
    np.random.shuffle(names_v)
    adj_mat = np.triu(np.ones((j, j)), 1)
    if j % 3 == 0:
        adj_mat[-1, 0] = 1  # Create a cycle for testing
    G = graph_from_adjacency_matrix(names_v, adj_mat)
```

```
    np.random.shuffle(G.node_array)
    top_order = topological_sort_DAG_check(G)
    if top_order:
        print(f"DAG: {', '.join([node.name for node in top_order])}")
    else:
        print("Cycle found!")
```

```
Cycle found!
DAG: J, A, I, F, E, D, G, C, H, B
DAG: J, K, I, C, A, H, B, E, G, D, F
Cycle found!
DAG: J, I, D, F, A, B, E, H, G, K, M, L, C
DAG: F, I, C, K, E, L, H, M, N, A, J, B, D, G
Cycle found!
DAG: K, P, O, D, I, C, L, A, F, E, B, H, N, G, J, M
DAG: M, C, I, G, P, B, N, E, F, O, L, H, K, Q, J, A, D
Cycle found!
DAG: Q, N, O, B, H, C, I, P, D, M, G, E, A, L, K, R, F, J, S
DAG: G, N, D, C, A, B, E, I, K, S, O, P, T, M, H, J, L, R, F, Q
Cycle found!
DAG: C, D, E, S, G, M, L, V, J, A, N, U, T, O, F, K, H, Q, B, R, P, I
DAG: J, L, G, Q, D, S, K, R, W, U, V, N, B, O, I, A, F, P, T, C, M, H, E
Cycle found!
DAG: O, K, S, Y, N, R, B, W, X, I, M, G, P, L, A, T, C, E, V, J, D, Q, U, H, F
DAG: G, F, N, O, H, D, V, U, A, B, K, P, Z, C, I, W, S, R, X, Q, J, M, E, T, L,
Y
Cycle found!
DAG: R, A, B, [, T, Q, E, N, O, C, K, Y, U, I, M, G, P, S, D, J, Z, V, L, W, H,
X, \, F
DAG: V, F, D, A, L, K, I, P, H, W, S, N, X, ], Z, E, B, Y, \, [, R, U, M, G, Q,
C, T, J, O
```

[ ]:

```
```

[ ]:

```
#Q3(a)
```

[13]:

```
def compute_max_island_area(grid_map: np.array) -> int:
    n, m = grid_map.shape
    visited = [[False for _ in range(m)] for _ in range(n)]

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def dfs(x, y):
        visited[x][y] = True
        area = 1

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
```

```
            if 0 <= nx < n and 0 <= ny < m and grid_map[nx][ny] == 1 and not␣
↪visited[nx][ny]:
                area += dfs(nx, ny)
        return area

    max_area = 0

    for i in range(n):
        for j in range(m):
            if grid_map[i][j] == 1 and not visited[i][j]:
                max_area = max(max_area, dfs(i, j))

    return max_area
```

[14]:

```
from scipy import signal

def random_grid_map(grid_width, land_propensity=0.5, interaction_strength=3):
    grid_map = np.random.choice([0, 1], size=(grid_width, grid_width), p=[1 -␣
↪land_propensity, land_propensity])
    grid_map = (0.5 + signal.convolve2d(grid_map, np.
↪ones((interaction_strength, interaction_strength)) /
                (interaction_strength ** 2)).astype(int)
    grid_map[0, :] = 0
    grid_map[-1, :] = 0
    grid_map[:, 0] = 0
    grid_map[:, -1] = 0
    return grid_map

np.random.seed(2024)
grid_map = random_grid_map(6)
print("The grid map looks as follows:")
print(grid_map)

max_area = compute_max_island_area(grid_map)
print(f"The largest island has area {max_area}.")
```

```
The grid map looks as follows:
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 1 1 1 0 0 0 0]
 [0 1 1 1 0 0 0 0]
 [0 1 1 1 1 0 0 0]
 [0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
The largest island has area 12.
```

[15]:
```python
np.random.seed(2024)
num_lakes_list = []

for i in range(15):
    grid_map = random_grid_map(100, land_propensity=0.45,␣
 ↪interaction_strength=5)
    num_lakes_list.append(compute_max_island_area(grid_map))

print(num_lakes_list)
```

```
[524, 170, 275, 274, 570, 308, 305, 268, 326, 289, 391, 329, 317, 423, 287]
```

[ ]:
```python

```

[ ]:
```python
#Q4(a) Dijkstra algorithm and Sheduling problem
```

[ ]:
```python
#Q4(b)
```

[16]:
```python
import heapq
import numpy as np

def dijkstras_min_euclidean_norm_path(input_graph, initial_node):
    for node in input_graph.node_array:
        node.min_euclidean_norm_path_from_start = np.inf
    initial_node.min_euclidean_norm_path_from_start = 0

    priority_queue = [(0, initial_node)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > current_node.min_euclidean_norm_path_from_start:
            continue

        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            edge_weight = edge.edge_length
            new_distance = np.sqrt(current_distance**2 + edge_weight**2)

            if new_distance < neighbor.min_euclidean_norm_path_from_start:
                neighbor.min_euclidean_norm_path_from_start = new_distance
                heapq.heappush(priority_queue, (new_distance, neighbor))
```

[17]:
```python
np.random.seed(2024)

num_nodes = 20
adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.7, 0.3])
names_v = [chr(ord('A') + i) for i in range(num_nodes)]
```

```python
G = graph_from_adjacency_matrix(names_v, adj_mat)

def assign_random_edge_lengths(graph, edge_length_max, random_seed=0):
    np.random.seed(random_seed)
    for node in graph.node_array:
        for edge in node.adjacency_list:
            edge.edge_length = np.random.randint(1, edge_length_max + 1)

assign_random_edge_lengths(G, 10)

def print_dijkstras_min_euclidean_norm_path_algorithm_output(graph,␣
 ↪initial_node_name):
    initial_node = graph.node_dictionary[initial_node_name]
    dijkstras_min_euclidean_norm_path(graph, initial_node)

    for node in graph.node_array:
        print(f"The minimum norm path from node {initial_node_name} to node␣
 ↪{node.name} is {node.min_euclidean_norm_path_from_start:.2f}")

print_dijkstras_min_euclidean_norm_path_algorithm_output(G, "A")
```

```
The minimum norm path from node A to node A is 0.00
The minimum norm path from node A to node B is 4.12
The minimum norm path from node A to node C is 5.10
The minimum norm path from node A to node D is 6.86
The minimum norm path from node A to node E is 5.48
The minimum norm path from node A to node F is 5.10
The minimum norm path from node A to node G is 5.74
The minimum norm path from node A to node H is 5.83
The minimum norm path from node A to node I is 4.47
The minimum norm path from node A to node J is 5.48
The minimum norm path from node A to node K is 4.12
The minimum norm path from node A to node L is 1.00
The minimum norm path from node A to node M is 5.74
The minimum norm path from node A to node N is 4.00
The minimum norm path from node A to node O is 5.10
The minimum norm path from node A to node P is 5.20
The minimum norm path from node A to node Q is 5.57
The minimum norm path from node A to node R is 5.00
The minimum norm path from node A to node S is 5.20
The minimum norm path from node A to node T is 4.00
```

[ ]:
```python
'''
Q4(c) All nodes' min_euclidean_norm_path_from_start values are initialized to␣
 ↪infinity, with the starting node
```

```
set to 0. This indicates that the starting node has a distance of 0, while␣
↪other nodes have not yet had any paths
discovered. A min-heap structure is used to always process the node with the␣
↪shortest Euclidean path calculated so
far. For each edge from node u to a connected node v, a new path is calculated.
↪If this path is shorter than the
distance currently stored for v, it updates the distance and inserts v into the␣
↪queue for further exploration.
Each node is processed only once, ensuring no infinite loops occur in a graph␣
↪with a finite number of nodes and
edges. Since Dijkstra's algorithm follows a greedy approach, a node's shortest␣
↪distance is finalized once it is
processed. The algorithm consistently applies the minimum Euclidean norm across␣
↪all edges. Through its greedy
approach, Dijkstra's algorithm calculates the shortest path for all nodes, and␣
↪this method is equally effective
when applied to Euclidean distances.
'''
```

[ ]:

[ ]:
```
#Q5(b)
```

[1]:
```
import numpy as np
import heapq

def min_graph_cost(edge_cost_matrix: np.array) -> float:
    n = edge_cost_matrix.shape[0]
    visited = [False] * n
    min_heap = []
    total_cost = 0

    visited[0] = True
    for j in range(1, n):
        heapq.heappush(min_heap, (edge_cost_matrix[0, j], 0, j))

    while len(min_heap) > 0:
        cost, u, v = heapq.heappop(min_heap)
        if visited[v]:
            continue

        visited[v] = True
        total_cost += cost

        for next_node in range(n):
            if not visited[next_node]:
```

```python
                heapq.heappush(min_heap, (edge_cost_matrix[v, next_node], v,␣
↪next_node))

    return total_cost
```

[2]:
```python
np.random.seed(2024) # set random seed
for n in range(3,31,3):
    edge_cost_matrix=np.random.randint(low=1,high=25, size=(n,n))
    edge_cost_matrix=np.triu(edge_cost_matrix,1)+np.triu(edge_cost_matrix,1).T
    min_cost=min_graph_cost(edge_cost_matrix)
    print(f"The minimum cost for this {n} by {n} edge cost matrix was␣
↪{min_cost}.")
```

```
The minimum cost for this 3 by 3 edge cost matrix was 2.
The minimum cost for this 6 by 6 edge cost matrix was 29.
The minimum cost for this 9 by 9 edge cost matrix was 34.
The minimum cost for this 12 by 12 edge cost matrix was 28.
The minimum cost for this 15 by 15 edge cost matrix was 31.
The minimum cost for this 18 by 18 edge cost matrix was 36.
The minimum cost for this 21 by 21 edge cost matrix was 45.
The minimum cost for this 24 by 24 edge cost matrix was 43.
The minimum cost for this 27 by 27 edge cost matrix was 40.
The minimum cost for this 30 by 30 edge cost matrix was 53.
```

[ ]:
```python
'''
Q5(c) The algorithm starts with an arbitrary node and initializes a priority␣
↪queue (min-heap) to store edges by
their costs. At the beginning, only edges connected to the starting node are␣
↪considered, ensuring that the first
edge added is the cheapest possible connection. The min-heap ensures that at␣
↪each step, the edge with the smallest
cost is processed first. This guarantees that the algorithm adds the␣
↪minimum-cost edge connecting the current MST
to a new node. Once a node is visited, it is marked as "visited" to prevent␣
↪processing it again. At every step, the
edge chosen minimizes the total cost. This is because the algorithm always␣
↪selects the smallest edge
connecting the current MST to an unvisited node. The total cost accumulated␣
↪during the execution of the algorithm
represents the sum of the weights of the edges, which is the minimum possible␣
↪cost to connect all nodes.
Thus, the function guarantees that the result is the minimum possible cost to␣
↪connect all nodes in the graph.
'''
```

CW 2.

Q5 (a) A graph with n nodes can have at most

$$\binom{n}{2} = \frac{n(n-1)}{2}$$ edges.

For each possible edge in E_{max}, we have two choices ; either include the edge in the graph G' or exclude it.

$\therefore$ The tortal number of possible subsets of edges is $2^{|E_{max}|} = 2^{\frac{n(n-1)}{2}}$

Because each subset of E_{max} corresponds to a unique graph G', the total number of possible graphs is

$2^{\frac{n(n-1)}{2}}$ //