Assignment 3

Av: Erlend Kulander Kvitrud

Python code is available at https://github.com/ErlendKK/DAT600-Algorithmic-Theory/tree/main/Assignment3

Problem 1

Problem 1 a)

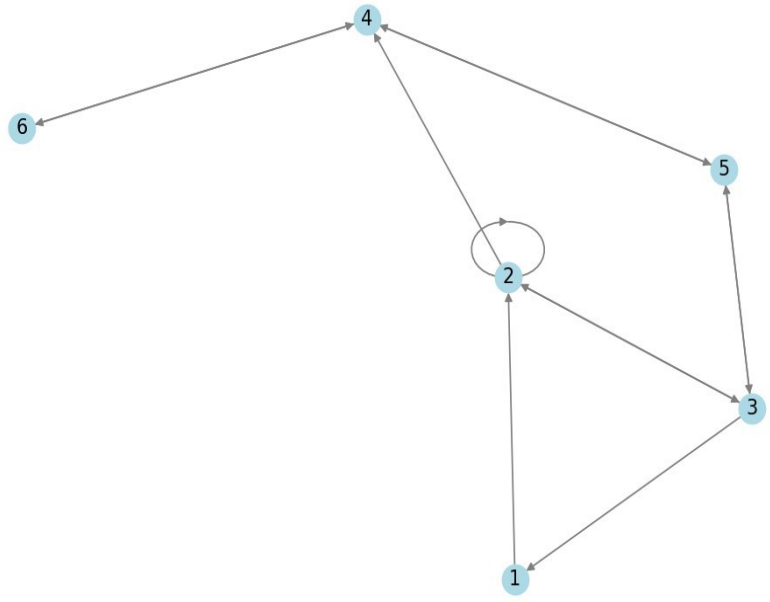
i) Here is my algorithm

```python
def create_adj_list(adj_matrix):
    adj_list = []

    for row in adj_matrix:
        new_list = []
        adj_list.append(new_list)

        # The index of any non-zero element equals the ID of a bordering node - 1
        for idx, num in enumerate(row):
            print(num)
            if num == 1:
                new_list.append(idx + 1)

    return adj_list
```

Here is the result

1: [2]
2: [2, 3, 4]
3: [1, 2, 5]
4: [5, 6]
5: [3, 4]
6: [4]

ii) The graph is drawn in mathplotlib using networkx. The code is available in the github repo.



Iii) This problem was solved manually by going over the graph vortex by vortex:

A: [B]
B: [C, D]
C: [E, F]
D [E, F]
E: [F, G, J]
F: [B, G, J]
G: []
H [I]
I: []
J: [I]

This solution was verified with the algorithm in probelm 1d.

Problem 1b
Here is my implementations of BFS and DFS in python. Both algorithms assume the graph to be sorted alphabetically. Although declaring 'visited as a set would be more efficient, I store it is as list so as to be able to return the list of visited vortices in their tranversed order.

Breadth-First Search (BFS)

```
def breadth_first_search(graph, s):
    queue = [s]
    visited = []
    times = {}

    while len(queue) > 0:
        current = queue.pop(0)
        start_time = time.time()
        if current in visited:
            end_time = time.time()
            times[current] = (start_time, end_time )
            continue

        visited.append(current)

        for node in graph[current]:
            if node not in visited and node not in queue:
                queue.append(node)

        end_time = time.time()
        times[current] = (start_time, end_time )

    return visited, times
```

The result was: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'J', 'H', 'I']

It was a bit unclear whether we should time each vortex or the entire function. I opted for the former. Here is a modified veresion including times. Times are logged as a dict in which the keys are vortices and the values are tuples of start and end times for each vortex.

Depth-First Search (DFS)
Depth-First Search is traditionally implemented recursively, but I prefere an iterative implementation for the following reasons:

- Memory: Although the memory complexities of both implementations are O(V), the actual amount of RAM required by the iterative implementation will be lower (no need to add new frames to the call stack for each edge explored).
- Runtime: Although both implementations have time-complexities of O(V+E), the iterative solution

spares the overhead of additoinal function calls
- Maintaiability: easier to modify (e.g. add edge cases), debug and otherwise maintain

My implementation of DFS is very similar to my implementation of BFS, with two modification:

1) The queue is replaced by a stack. This means that vortices are added to the start of the 'visited' list (using .insert(0, node)) instead of the end (using .append(node)).
2) When iterating over the list of vortices connected to the vortex stored as 'current', I iterate backwards to ensure that each vortex is added to the stack in the correct order.

```python
def depth_first_search(graph, s):
    stack = [s]
    visited = []
    times = {}

    while len(stack) > 0:
        current = stack.pop()
        start_time = time.time()
        if current in visited:
            end_time = time.time()
            times[current] = (start_time, end_time )
            continue

        visited.append(current)

        # Need to iterate backwards here to ensure the correct order in the stack
        length = len(graph[current])
        for i in range(length-1, -1, -1):
            node = graph[current][i]
            if node not in visited:
                stack.append(node)

        end_time = time.time()
        times[current] = (start_time, end_time )

    return visited, times
```

The result was: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'I']

Problem 1c

The graph is already directed => it will become DAG (Directed Acyclic Graph) if it becomes acyclic. From figure 1 it is easy to see that all cycles are located on the left-hand side of the graph, and all involve F -> B. For instance, there are three different paths cycling back to node B:

B -> C -> F -> B
B -> C -> E -> F -> B
B -> D -> F -> B

Breaking any single one of these paths, by removing any edge along the path other than F -> B, would leave the other path intact. Removing both would isolate A-B from the rest of the graph. The graph thus cannot become acyclic as long as the edge F -> B remains. => The only way to make the graph acyclic is to remove the edge F -> B.

```python
def topological_sort(graph):
    ordered_graph = []

    while len(graph) > 0:
        incomming_edges = {key : 0 for key in graph.keys()}
        list_of_edge_lists = [value for value in graph.values()]

        for list in list_of_edge_lists:
            for i in range(len(list)):
                node = list[i]
                incomming_edges[node] += 1

        for node in incomming_edges.keys():
            if incomming_edges[node] == 0:
                ordered_graph.append(node)
                graph = {key:value for key, value in graph.items() if key != node }
                break

    return ordered_graph
```

Problem 1d

I will solve this problem with a simplified implementation of Kahn's Algorithm which assumes the input graph to be a DAG, and thus does not check for cycles (if this assumption does not hold, I could always start the function by calling dagify(graph) which is defined in problem 1e.

The algorithm counts the number of dependencies for each node, then iteratively moves nodes without any dependencies from the input graph to the ordered graph.

Below is my general algorithm for dagifying directed graphs.

```python
def acyclify(graph, node, visited, recursion_stack):
    visited.add(node)
    recursion_stack.add(node)

    for neighbour in graph[node]:
        if neighbour not in visited:
            if acyclify(graph, neighbour, visited, recursion_stack):
                return True

        elif neighbour in recursion_stack:
            if neighbour in graph[node]:
                graph[node].remove(neighbour)

            return True

    recursion_stack.remove(node)
    return False

def dagify(graph):
    visited = set()
    recursion_stack = set()

    graph = {node: neighbors[:] for node, neighbors in graph.items()}

    for node in list(graph.keys()):
        if node not in visited:
            acyclify(graph, node, visited, recursion_stack)

    return graph

for key, value in dagify(graph).items():
    print(key, value)
```

Here is the results:
A ['B']
B ['C', 'D']
C ['E', 'F']
D ['F']
E ['F', 'G', 'J']
F ['G', 'H', 'J']
G []
H ['I']
I []
J ['I']

The only missing edge is F -> B which is the one I concluded had to be removed for the graph to become a DAG => at least in this case the algorithm works.

Tests including two additional edges (one from I → C and another from C → A) are included in the repo.

Problem 2

Problem 2a)
this is an instance of the minimum-spanning-tree (MST) problem. The text book describes two greedy algorithms for solving the MST: Kruskal's algorithm; and Prim's algorithm. Kruskals algorithm is often more efficient for sparse grafs (number of edges is << the maximum number of edges, which for an undirected graph is V(V-1)/2, where V is the number of vertices). For this graph, the maximum number of edges would be 8(8-1)/2 = 28 edges. => the graph is sparse => I choce Kruskals algorithm.

```
vertices = len(graph)

def kruskal(graph):
    parent = dict() # tracks the parent of each vertex
    rank = dict()   # track the depth of trees in the disjoint set
    mst = []        # solution to the problem

    # Helper function for determining which subset a particular element is in
    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    # Helper-function for joining two subsets.
    def union(root1, root2):
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        elif rank[root1] < rank[root2]:
            parent[root1] = root2
        else:
            parent[root1] = root2
            rank[root2] += 1

    for edge in graph:
        parent[edge[0]] = edge[0]
        parent[edge[1]] = edge[1]
        rank[edge[0]] = 0
        rank[edge[1]] = 0

    sorted_edges = sorted(graph, key=lambda item: item[2])

    for edge in sorted_edges:
        root1 = find(edge[0])
        root2 = find(edge[1])
        if root1 != root2:
            mst.append(edge)
            union(root1, root2)

    total_weight = sum(edge[2] for edge in mst)
    return mst, total_weight
```

Spanning Tree:

('A', 'D', 1)
('D', 'C', 2)
('D', 'E', 2)
('D', 'B', 4)
('D', 'F', 4)
('C', 'G', 6)
('F', 'H', 7)

Total weight: 26

Problem 2b)
My solution to this problem is allmost identical to the solution to the previous problem with the exception that I included a counter to track the number of D-edges included in A, increments this counter each time an edge involving a D-node is added, and stops adding such noedes to A once the threshold is met.

```python
def kruskal_with_max_three_D_edges(graph):
    parent = dict()
    rank = dict()
    mst = []
    number_of_D_edges = 0

    # Helper function for determining which subset a particular element is in
    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    # Helper-function for joining two subsets.
    def union(root1, root2):
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        elif rank[root1] < rank[root2]:
            parent[root1] = root2
        else:
            parent[root1] = root2
            rank[root2] += 1

    for edge in graph:
        parent[edge[0]] = edge[0]
        parent[edge[1]] = edge[1]
        rank[edge[0]] = 0
        rank[edge[1]] = 0

    sorted_edges = sorted(graph, key=lambda item: item[2])

    for edge in sorted_edges:
        root1 = find(edge[0])
        root2 = find(edge[1])
        if root1 != root2:

            # Increment the D-count if allowed; otherwise skip this node
            if 'D' in edge:
                if number_of_D_edges >= 3:
                    continue
                else:
                    number_of_D_edges += 1

            mst.append(edge)
            union(root1, root2)

    total_weight = sum(edge[2] for edge in mst)
    return mst, total_weight
```

The result of this algorithm is:
('A', 'D', 1)
('D', 'C', 2)
('D', 'E', 2)
('B', 'A', 5)
('C', 'G', 6)
('F', 'H', 7)
('E', 'H', 8)

Total weight: 31

From this it doesn't seem possible to comply with the weight constraint for this case.

Suboptimal solution: Kruskal's algorithm assumes that you can always take the lowest-weighted edge

available that doesn't form a cycle. By imposing restrictions on the number of edges a given node may have, this assumption is not valid. The algorithm may consequently skip edges that would have been part of the optimal MST, thus having to choce edges with larger weights later on => increased total weight of the spanning tree relative to the globaly optimal solution.

A simple example is one in which all the edges connected to 'D' has the same weight (eg. 4 in the example below). Which of the five potential edges connected to D that the algorithm includes in its MST would then be totally arbetrary and depend on their ordering in the input list.

```
graph = [
    ('A', 'D', 4),
    ('B', 'D', 4),
    ('C', 'D', 4),
    ('E', 'D', 4),
    ('F', 'D', 4),
    ('B', 'A', 5),
    ('C', 'G', 6),
    ('F', 'G', 9),
    ('F', 'H', 7),
    ('E', 'H', 8),
    ('E', 'B', 8)
]
```

Problem 2c
This can be achieved by swiching the weights of ('B', 'A', 5) and ('F', 'H', 7). The following graph would have a total weight of 24. As far as I can tell this is the lowest total weight that can be acchieved by adjusting the position of a single edge.

=> the graph can be made to comply with any constraint >= 24.

Problem 3

Problem 3a)
The most straighforward solution is to use a BFS/ DFS algorithm that loops over all vertices ('candidate_champions' in the implementation shown below). If the 'visited' list of a given candidate includes all vertices in the input graph, the candidate is a true champion and is added to the list of champions. After looping thorough all candidats, the list of true champions is returned. This function has an outer loop spanning all input vertices, each of which also loops over each input vertices => exponential time complexity.

```python
def find_champions(graph):
    vertices = [key for key in graph.keys()]
    champions = []

    for i in range(len(vertices)):
        candidate_champ = vertices[i]
        queue = [candidate_champ]
        visited = []

        while len(queue) > 0:
            current = queue.pop(0)
            if current in visited:
                continue

            visited.append(current)
            print(visited)

            for node in graph[current]:
                if node not in visited and node not in queue:
                    queue.append(node)

        if all(vertex in visited for vertex in vertices):
            champions.append(candidate_champ)

    return champions,
```

Problem 3b)
Finding groups in which each node has defeated the others, either directly or indirectly, enatails looking for loops. Below is my functions for doing so. My first version of the algorithm, which did not include merge_groups, would stop seeking larger groups once some groups was found. In this case, it thus stoped looking once {A,B} was found, and thus didn't find {A,B,D}. Consequently it returned {A,B}, {E,F,G}, {C}, {D}.

To find the solution {A,B,D}, I had to add the helper function «merge_groups» which incorporates any singletons into an existing group, if possible. This is probably not the most efficient way of solving this problem, and is also not a completely general solution (if I inerpreted the intent of the problem correctly; that the goal was to return the largest possible groups), yet it gives the correct solution for this problem, and genereally given the restraint that no cyclic group includes one or more cylic sub-groups.

```python
def modified_depth_first_search(node, visited, path):
    if node in path:
        # Found a cycle, extract it and return
        cycle_start_index = path.index(node)
        return set(path[cycle_start_index:])
    if node in visited:
        return None

    visited.add(node)
    path.append(node)

    for neighbor in graph.get(node, []):
        cycle = modified_depth_first_search(neighbor, visited, path)
        if cycle:
            return cycle
    path.pop()
    return None

def merge_groups(graph, cyclic_groups):
    singletons = [''.join(group) for group in cyclic_groups if len(group) == 1]
    groups = [group for group in cyclic_groups if len(group) > 1]

    for singleton in singletons:
        for group in groups:
            for node in group:
                if singleton in graph[node] and any(node in graph[singleton] for node in group):
                    group.add(singleton)
                    singletons.remove(singleton)
                    break

    cyclic_groups = groups
    for singleton in singletons:
        cyclic_groups.append(set([singleton]))

    return cyclic_groups


def find_cyclic_groups(graph):
    visited = set()
    cyclic_groups = []
    all_nodes = set(graph.keys())
    cycle_nodes = set()

    for node in graph:
        cycle = modified_depth_first_search(node, visited, [])
        if cycle:
            if not any(cycle <= c for c in cyclic_groups):
                cyclic_groups = [c for c in cyclic_groups if not c <= cycle]
                cyclic_groups.append(cycle)
            cycle_nodes.update(cycle)

    non_cyclic_nodes = all_nodes - cycle_nodes
    for node in non_cyclic_nodes:
        cyclic_groups.append({node})

    return merge_groups(graph, cyclic_groups)
```

Complexity:

the algorithm is devided into 3 functions. The total time complexity can be found by assessing each of these seperately.

modified depth-first search: Like an ordinary DFS this function iterates over each node, and for each node it explores all its neighbors. Its time complexity is thus O(V+E).

merge groups: It consists of three nested loops that iterates over singletons (S), then over groups (G), and then over nodes in groups (N). This makes its time complexity O(S*G*N). In the worst case, where all nodes form singleton- groups, S == G == V and N == 1. Complexity thus reduces to O(V2).

find_cyclic_groups: It first iterates over the vertices (V) and calls modified_depth_first_search V times => O(V x (V + E) = O(V2 + EV). It then calls merge_groups once (O(V2)). Finally, it iterates over cyclic_groups which in the worst case (all groups consist of 2 nodes) is O(V/2) which reduces to O(V).

Total complexity thus reduces to O(V2 + EV) for a dense graph or O(V2) for a sparse graph. In the worst case, in which E = V2, it reduces to O(V3).

Problem 4

Problem 4a)
Here is a simple example which the algorithm gets wrong:

```
edges = [
    ('A', 'B', 5),
    ('A', 'C', 3),
    ('B', 'D', -100),
    ('C', 'D', 2)
]
```

the correct answer is

```
{
  'A': 0,
  'B': 5
  'C': 3,
  'D': -95
}
```

while Dijkstra's_algorithm returns

```
{
  'A': 0,
  'B': 5
  'C': 3,
  'D': 5,
}
```

Problem 4b)
Below is my implementation of a solution that fixes this problem (find_shortest_path). It applies Dijkstra's_algorithm if all weights are positive and the graph is sparse (operationalized as E < V * log(V)); otherwise it applies the Bellman Ford algorithm.

```python
def find_vertices(edges):
    vertices = set()
    for edge in edges:
        vertices.update([edge[0], edge[1]])
    return vertices


def dijkstra_shortest_path(edges, start):
    vertices = find_vertices(edges)
    unvisited = vertices.copy()
    distances = {vertex: float('inf') for vertex in vertices}
    distances[start] = 0

    while unvisited:
        current_node = min(unvisited, key=lambda vertex: distances[vertex])
        unvisited.remove(current_node)

        edges_to_check = [edge for edge in edges if current_node == edge[0]]

        for edge in edges_to_check:
            neighbor, weight = edge[1], edge[2]

            if neighbor in unvisited:
                new_distance = distances[current_node] + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance

    return distances

def bellman_ford_shortest_path(edges,start):
    vertices = find_vertices(edges)
    distances = {vertex: float('inf') for vertex in vertices}
    distances[start] = 0

    for _ in range(len(vertices) - 1):
        for u, v, w in edges:
            if distances[u] != float('inf') and distances[u] + w < distances[v]:
                distances[v] = distances[u] + w

    for u, v, w in edges:
        if distances[u] != float('inf') and distances[u] + w < distances[v]:
            print("Graph contains a negative weight cycle")
            return None

    return distances


def find_shortest_path(edges, start):
    from math import log

    vertices = find_vertices(edges)
    sparse = len(edges) < len(vertices) * log(len(vertices))
    negative_weights = [edge[2] for edge in edges if edge[2] < 0]

    # Use Dijkstra's algorithm for sparce graphs if possible
    if len(negative_weights) == 0 and sparse:
        return dijkstra_shortest_path(edges, start)

    # Otherwise, use Bellman-Ford's algorithm
    return bellman_ford_shortest_path(edges, start)
```

Problem 5:

Problem 5a)
The antiparallel edge issue can be solved by replacing one of the two parallell edges with a new node placed between v1 and v3 (v13 in the edge-lists below).

```
edges_old = [
   ('s', 'v1', 14),
   ('s', 'v2', 25)
   ('v1', 'v3', 3),
   ('v1', 'v4', 21),
   ('v3', 'v1', 6), #removed node
   ('v3', 'v5', 15),
   ('v4', 'v3, 10),
   ('v4', 't', 20),
   ('v5', 'v4', 5),
   ('v5', 't', 10),
]

edges_new = [
   ('s', 'v1', 14),
   ('s', 'v2', 25)
   ('v1', 'v3', 3),
   ('v1', 'v4', 21),
   ('v3', 'v13', 6), #added node
   ('v13', 'v1', 6), #added node
   ('v3', 'v1', 6),
   ('v3', 'v5', 15),
   ('v4', 'v3, 10),
   ('v4', 't', 20),
   ('v5', 'v4', 5),
   ('v5', 't', 10),
]
```

Problem 5b)
All flows are initialized to 0. The residual graph is initialized as the original graph. I will use depth-first-search, always proceeding to the next lowest-indexed, available neighbour.

The first augmenting path is
S -> V1 -> V3 -> V5 -> V4 -> t

The maximum flow along this path is min(14, 3, 15, 5, 10) = 3

```
residual graph = [
   ('s', 'v1', 11),
   ('s', 'v2', 25)
   ('v1', 'v3', 0),
   ('v1', 'v4', 21),
   ('v3', 'v13', 6),
   ('v13', 'v1', 6),
   ('v3', 'v1', 6),
   ('v3', 'v5', 12),
   ('v4', 'v3, 10),
```

```
    ('v4', 't', 20),
    ('v5', 'v4', 2),
    ('v5', 't', 7),
]
```

The next augmenting path is:
S -> V1 -> V4 -> V3 -> V5 -> t;
The maximum flow along this path is min(11, 21, 10, 12, 7) = 7

```
residual graph = [
    ('s', 'v1', 4),
    ('s', 'v2', 25)
    ('v1', 'v3', 0),
    ('v1', 'v4', 14),
    ('v3', 'v13', 6),
    ('v13', 'v1', 6),
    ('v3', 'v1', 6),
    ('v3', 'v5', 5),
    ('v4', 'v3, 3),
    ('v4', 't', 20),
    ('v5', 'v4', 5),
    ('v5', 't', 0),
]
```

The next augmenting path is:
S -> V1 -> V4 -> t;
The maximum flow along this path is min(4, 14, 20) = 4

```
residual graph = [
    ('s', 'v1', 0),
    ('s', 'v2', 25)
    ('v1', 'v3', 0),
    ('v1', 'v4', 10),
    ('v3', 'v13', 6),
    ('v13', 'v1', 6),
    ('v3', 'v1', 6),
    ('v3', 'v5', 5),
    ('v4', 'v3, 3),
    ('v4', 't', 16),
    ('v5', 'v4', 5),
    ('v5', 't', 0),
]
```

etc.

Problem 5c)

If I had completed this analysis, I could have used an algorithm that used DFS/ BFS on the final residual graph to identify all reachable vertices. The set of reachable vertices (those that still have available capacity) and the set of non-reachable vertices would then have formed the two sides of a minimum cut. The edges that cross the minimum cut (n the original graph, not the residual graph) would have been the bottlenecks.

Problem 5d)
The time complexity of the algorithm depends on the choice of search algorithm and characteristics of the graph.

For both DFS and BFS, the best case is theta(E),which occurs if only a single path is needed to reach maximum flow.

DFS worst case is O(E * F) (where F = maximum total flow) which occurs when each augmenting path only adds 1 unit of flow => F iterations are required to reach maximum flow.

BFS worst case is O(V * E2) which occurs when an edge is used in O(V) augmenting paths, and the BFS (O(E)) must be repeated for E edges.

Improvements: the traditional implementation of the algorithm might be improved by making it select DFS or BFS based on characteristics of the graph, such as denisty. It might further be improved by applying capacity scaling (prioritizing paths with higher capacities in the early stages of the algorithm) if the variance in capacities vary wildly.