Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2023
Report for Assignment 3

Weizhi Ao

8th December 2023

Part 1
Problem 1: Controlling the Maximum Flow

A Specification and Program

A.1 Implement of $sensitive(G, s, t)$ Function

As the code with comments is a little long to place here, I stick that at end of document.1

The `sensitive` function is designed to identify a sensitive edge in a flow network. This network is represented by a graph G with a specified source s and sink t. A sensitive edge is characterized as one where a reduction in its capacity leads to a decrease in the network's maximum flow. The function commences by constructing a residual network R from G. In R, each edge from u to v is assigned a capacity equivalent to the unused capacity of the corresponding edge in G, and each reverse edge from v to u is given a capacity equal to the flow in G. Utilizing Depth-First Search (DFS) initiated from source s, the function identifies all vertices reachable in the residual network, accumulating them in the set `ReachableVertices`. The final step involves iterating over every edge in G. For edges where flow matches capacity (fully utilized), the function examines if they cross from the reachable vertex set to the other vertices. An edge that meets this criterion is marked as sensitive and returned. If no such edge is detected, the function yields $(\text{None}, \text{None})$, signaling the absence of sensitive edges in the network.

B worst-case time complexity

Let V be the number of vertices and E be the number of edges in the graph G.

B.1 Creating the Residual Network (G_f):

- The algorithm iterates through all edges of G.
- For each edge (u, v), it adds two edges to G_f.
- If adding an edge takes constant time, this part has a time complexity of O(E).

B.2 Depth-First Search (DFS):

- The DFS traverses each vertex and edge of G_f at most once.
- The time complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges in G_f.
- Since G_f is constructed from G and has at most twice the number of edges, the complexity remains O(V + E).

B.3 Identifying Sensitive Edge:

- The function iterates over all edges in G.
- For each edge, it performs constant-time checks and possibly returns the edge.
- This iteration contributes O(E) to the time complexity.

Combining these parts, the overall worst-case time complexity of the *sensitive* function is O(V + E).

C Analysis with sensitive edge

Not every edge at capacity (where flow equals its capacity) in a flow network is necessarily sensitive. A sensitive edge is one whose capacity reduction would decrease the network's overall maximum flow. Key reasons why not all edges at capacity are sensitive include:

1. **Alternative Paths:** The existence of alternative paths in the network can compensate for the reduced capacity of an edge at capacity, meaning the edge is not sensitive.
2. **Position in the Network:** An edge's significance in the network's overall structure matters. Edges not part of any minimum cut do not affect maximum flow when their capacity is reduced.
3. **Flow Distribution:** The current flow distribution is one possible configuration. Other configurations achieving the same maximum flow might not utilize the same edges at full capacity.

Thus, while an edge being at capacity is necessary for being sensitive, it is not sufficient. The edge's role and the existence of alternative flow paths are crucial in determining sensitivity.

D Real-world example

1. **(a) Description of a Sensitive Edge:**

   In the context of a traffic network where edges are roads and nodes are intersections, a sensitive edge would correspond to a road segment where the current traffic flow is equal to its maximum capacity. A reduction in this road's capacity (due to, for instance, a lane closure or construction) would immediately impact the overall traffic flow, potentially causing congestion or rerouting of traffic. This edge is critical in maintaining the optimal flow of vehicles from the source (vehicle's position) to the sink (vehicle's destination).

2. **(b) Importance of Finding Sensitive Edges:**

   Identifying sensitive edges in a traffic network is crucial for several reasons:

   (a) **Traffic Management:** Knowing which roads are sensitive can help in traffic planning and management. It allows city planners and traffic controllers to predict and mitigate congestion before making road changes or during unexpected incidents.

   (b) **Infrastructure Improvement:** Identifying sensitive edges can guide infrastructure development. Prioritizing upgrades or maintenance on these roads can have a significant impact on improving overall traffic flow and reducing congestion.

   (c) **Emergency Planning:** In emergency scenarios, understanding which roads are critical for maintaining traffic flow helps in effective route planning for emergency vehicles and in designing evacuation routes.

In summary, sensitive edges in a traffic network are those road segments that are crucial for maintaining the optimal flow of traffic, and identifying them is key for efficient traffic management, infrastructure planning, and emergency response strategies.

Part 2

Problem 2: The Party Seating Problem

A Formulate the two-table party seating problem as a graph problem.

1. *Given:* A set of guests G and a symmetric set $known[g]$ for each guest $g \in G$, indicating the other guests known by g.

2. *Objective:* Seat all guests at two tables such that no guest knows any other guest at the same table.

3. *Formulation:*

   - Represent each guest as a vertex in an undirected graph.
   - Draw an edge between vertices g_1 and g_2 if g_1 knows g_2 (and vice versa).
   - The problem reduces to coloring the graph with two colors (representing two tables) such that no two adjacent vertices (guests who know each other) share the same color.

   Constraints:

   - Every vertex must be assigned one of two colors.
   - Adjacent vertices cannot have the same color.

   Solution:

   - A valid solution exists if and only if the graph is bipartite.
   - If the graph can be colored with two colors under these constraints, the seating arrangement is possible. Otherwise, it is not.

B Design and implement an algorithm

2

Function Signature:

```
def party(known: List[Set[int]]) -> Tuple[bool, Set[int], Set[int]]:
```

This defines the function `party` which takes a list of sets, `known`, as input. Each set in the list represents the guests known by a particular guest (indexed by the list). The function returns a tuple consisting of a boolean and two sets of integers.

Nested Function - is_bipartite:

```
def is_bipartite(v, color):
```

This is a helper function used for depth-first search (DFS) in the graph. It takes a vertex `v` and a color list `color` as input. The function iterates over each neighbor `u` of the vertex `v`. There are two main checks here:

- If the neighbor `u` is already colored with the same color as `v`, then the graph is not bipartite (return `False`).
- If the neighbor `u` is not colored (`color[u] == -1`), color it with an alternate color (`1 - color[v]`) and recursively call `is_bipartite` for `u`.

Initialize Color List:

```
color = [-1] * n
```

This creates a list of size `n` (number of guests), initially setting all elements to `-1`, indicating that no vertices have been colored yet.

Main Loop:

The function iterates through each vertex (guest) in the graph.

- If a vertex is not colored (`color[i] == -1`), the function colors it (`color[i] = 0`) and calls `is_bipartite` to recursively color connected vertices.
- If `is_bipartite` returns `False` at any point, the function immediately returns `False` with two empty sets, indicating that no valid seating arrangement is possible.

Creating Seating Arrangements:

If the graph is bipartite (no conflicts found), the function then creates two sets, `table1` and `table2`.

- `table1` includes all vertices with color `0`, and `table2` includes all vertices with color `1`.

Return Value:

The function returns `True` (indicating a valid arrangement is possible) along with the two sets `table1` and `table2`, representing the seating arrangement for each table.

Example Call:

```
print(party([{1, 2}, {0}, {0}]))
```

In this example, guest `0` knows guests `1` and `2`. The function checks whether it's possible to seat them at two tables such that no two guests who know each other sit at the same table. The expected output for this example is (`True`, {`0`}, {`1`, `2`}), indicating a valid arrangement is possible with guest `0` at one table and guests `1` and `2` at the other.

C Time complexity

The time complexity of the `party` algorithm can be analyzed by considering the graph representation and the depth-first search (DFS) procedure used for graph coloring.

1. **Graph Representation:** The graph is implicitly represented by the `known` list of sets, where each set contains indices of guests known by a particular guest. The total number of vertices in the graph is $|V| = n$, where n is the number of guests, equivalent to the length of the `known` list. The edges in the graph are represented by these sets, with symmetry in the acquaintance relationships.

2. **Edges and ℓ:** The total number of edges, denoted as $|E|$, is related to ℓ, the sum of lengths of the lists in `known`. Each acquaintance relation is counted twice, once in each guest's list, hence the actual number of edges is $\ell/2$. For complexity analysis, the sum of lengths of these lists is considered, equating to ℓ.

3. **DFS Procedure:** The DFS procedure colors the graph. For each vertex, we explore its neighbors. DFS traversal ensures each vertex and edge is visited at most once, leading to a complexity proportional to ℓ, the sum of lengths of the adjacency lists.

4. **Overall Time Complexity:** Initializing the `color` list takes $O(n)$ time. The DFS procedure takes $O(\ell)$ time, as it traverses each edge exactly once. Therefore, the overall time complexity of the algorithm is $O(n + \ell)$. Here, n accounts for initialization and potential vertex visits, while ℓ covers edge traversal. Since n is generally smaller or comparable to ℓ, the complexity can also be expressed as $O(\ell)$, aligning with the given complexity of $O(|\text{known}| + \ell)$.

D Advanced problem

The modified party seating problem can be formulated as a maximum-flow problem. We construct a network flow graph where the flow from a source node to a sink node represents the assignment of guests to tables under the given constraints.

Graph Nodes:

- **Source Node (S)**: Represents the starting point of the flow.
- **Group Nodes**: Represent each of the p groups. Each group node corresponds to a group in the `Group` array.
- **Table Nodes**: Represent each of the q tables. Each table node corresponds to a table in the `Table` array.
- **Sink Node (T)**: Represents the endpoint of the flow.

Graph Edges and Capacities:

- **Edges from Source to Group Nodes**: Connect the source node to each group node. The capacity of each edge is equal to the size of the group, signifying that only that many members from the group can be seated.
- **Edges from Group Nodes to Table Nodes**: Connect each group node to all table nodes. The capacity of these edges is set to 1, indicating that at most one member from a group can be seated at a table.

- **Edges from Table Nodes to Sink Node**: Connect each table node to the sink node. The capacity of each of these edges is equal to the size of the table, representing the maximum number of guests that can be seated at that table.

Objective:

- The objective is to find the maximum flow in the graph from the source node to the sink node. A feasible seating arrangement exists if and only if the maximum flow value equals the total number of guests (sum of the sizes of all groups).

Flow Conservation:

- At each group node and table node, the flow conservation rule must hold: the sum of the incoming flow must equal the sum of the outgoing flow.

In this formulation, finding a seating arrangement where no more than one member from each group is seated at the same table is equivalent to determining if the maximum flow in the network equals the total number of guests. If this condition is satisfied, the flow from group nodes to table nodes indicates the seating arrangement for each guest.

```python
def sensitive(G: Graph, s: str, t: str) -> Tuple[str, str]:
    '''
    Finds a sensitive edge in a flow network G with source s and sink t.
    A sensitive edge is one whose reduction in capacity would reduce the overall / max flow of the network.

    Pre: G is a flow network at maximum flow, with s as its source and t as its / sink.
    Post: Returns a sensitive edge if one exists, otherwise returns (None, None).
    Ex: sensitive(g1, 'a', 'f') = ('b', 'd')
    '''

    # Function to create the residual network from the original graph G
    def create_residual_network(G: Graph) -> Graph:
        R = Graph(is_directed=True)
        for u, v in G.edges:
            flow, capacity = G.flow(u, v), G.capacity(u, v)
            # Add forward edge with residual capacity
            R.add_edge(u, v, capacity - flow)
            # Add reverse edge with capacity equal to the flow
            R.add_edge(v, u, flow)
        return R

    # Set to keep track of vertices reachable from source s in the residual network
    ReachableVertices = set()

    # Create the residual network from G
    R = create_residual_network(G)

    # Depth-First Search to find all vertices reachable from s in the residual / network
    def DFS(u: str) -> None:
        ReachableVertices.add(u)
        for v in R.neighbors(u):
            # If vertex v is not visited and there is residual capacity, visit it
            if v not in ReachableVertices and R.weight(u, v) > 0:
                DFS(v)
    DFS(s)

    # Iterate over all edges in G to find a sensitive edge
    for u, v in G.edges:
        # Skip edges that are not fully utilized
        if G.capacity(u, v) == 0 or G.flow(u, v) < G.capacity(u, v):
            continue

        # Check if the edge crosses from reachable to unreachable vertices
        if (u in ReachableVertices and v not in ReachableVertices) or (u not in / ReachableVertices and v in ReachableVertices):
            return (u, v) # Return the sensitive edge

    # If no sensitive edge is found, return (None, None)
    return (None, None)
```

Listing 1: Implement of $sensitive(G, s, t)$ Function

```python
def party(known: List[Set[int]]) -> Tuple[bool, Set[int], Set[int]]:
    '''
    Determines if a valid seating arrangement exists for a party
    where no two acquainted guests sit at the same table.

    Pre: known is a list of sets, where known[i] contains the guests known by guest i.
    Post: Returns a tuple with a boolean indicating if a valid arrangement exists,
         and two sets representing the seating arrangement at two tables.
    Ex: party([{1, 2}, {0}, {0}]) = True, {0}, {1, 2}
    '''

    def is_bipartite(v, color):
        for u in known[v]:
            # If the neighbor is colored with the same color, it's not bipartite
            if color[u] == color[v]:
                return False

            # If the neighbor is not colored, color it with alternate color and check further
            if color[u] == -1:
                color[u] = 1 - color[v]
                if not is_bipartite(u, color):
                    return False

        return True

    n = len(known)
    color = [-1] * n # -1 means uncolored, 0 and 1 are the two colors

    for i in range(n):
        if color[i] == -1: # If the vertex is not colored, color it and check for bipartite
            color[i] = 0
            if not is_bipartite(i, color):
                return False, set(), set()

    # If graph is bipartite, divide vertices based on colors
    table1 = {i for i in range(n) if color[i] == 0}
    table2 = {i for i in range(n) if color[i] == 1}

    return True, table1, table2
```

Listing 2: Case2 dfs implement