Assignment 11

Problem 1. Is it a bottleneck? (10 pts)

Let $G = (V, E)$ be a flow network with source s and t sink. We say that an edge e is a bottleneck if it crosses every minimum-capacity cut separating s from t. Give an efficient algorithm to determine if a given edge e is a bottleneck in G and explain the complexity.

Answer:

To determine if a given edge e is a bottleneck in the flow network G(V, E) with source S and sink T, you can use the following algorithm:

1. Calculate the maximum flow in the network G without including the edge e. You can use any efficient flow algorithm like the Ford-Fulkerson algorithm, Edmonds-Karp algorithm, or Dinic's algorithm for this purpose. This step will find the maximum amount of flow that can be sent from S to T without using the edge e.
2. Create a residual graph G_res(V, E_res) based on the current flow in G. The residual graph represents the remaining capacity for each edge after the flow is calculated. For each edge (u, v) in G, add two edges to G_res: (u, v) with capacity c(u, v) - f(u, v) (remaining capacity), and (v, u) with capacity f(u, v) (backwards edge representing flow that can be canceled).
3. Run a graph traversal algorithm, like BFS (Breadth-First Search), to find all the vertices reachable from S in G_res. This step will identify the set of vertices that are still connected to the source S in the residual graph.
4. Check if the sink T is reachable from S in the residual graph G_res. If T is not reachable, then the edge e is a bottleneck because removing it disconnects S from T, and it is required to maintain the flow from S to T.
5. If T is reachable from S in G_res, then the edge e is not a bottleneck because there exists an alternative path from S to T that does not use e, and we can still achieve the same flow from S to T without e.

Algorithm Complexity:

1. Calculating the maximum flow in the network G requires running a flow algorithm. The complexity of flow algorithms like Ford-Fulkerson, Edmonds-Karp, or Dinic's is

typically O(E * V^2) for general graphs. However, if the flow capacities are integers, the complexity can be reduced to O(E * V * C) using the scaling technique, where C is the maximum capacity in the graph.

2. Constructing the residual graph G_res takes O(E) time, as we add two edges for each original edge.
3. The graph traversal (BFS) in the residual graph G_res takes O(V + E_res) time, which is equivalent to O(V + E) as E_res <= 2E (since each edge can contribute two residual edges in the worst case).

Overall, the algorithm's complexity is dominated by the flow algorithm used in step 1, giving a final complexity of $O(E * V^2)$ or $O(E * V * C)$ if the capacities are integers. This algorithm efficiently determines if a given edge is a bottleneck in the flow network.

Problem 2. Word construction (10 pts)

Given a word of length n and n five-sided dice with a character on each side. Find out if this word can be constructed by the set of given dice.

Example 1

Input: word = "hello" dice = [[a, l, c, d, e], [a, b, c, d, e], [a, b, c, h, e], [a, b, c, d, o], [a, b, c, l, e]]

Output: true

Explanation: dice[2] + dice[1] + dice[0] + dice[4] + dice[3]

Example 2

Input: word = "hello" dice = [[a, b, c, d, e], [a, b, c, d, e], [a, b, c, d, e], [a, b, c, d, e], [a, b, c, d, e]]

Output: false

Example 3 Input: word = "aaaa" dice = [[a, a, a, a, a], [b, b, b, b, b], [a, b, c, d, e], [a, b, c, d, e]]

Output: false

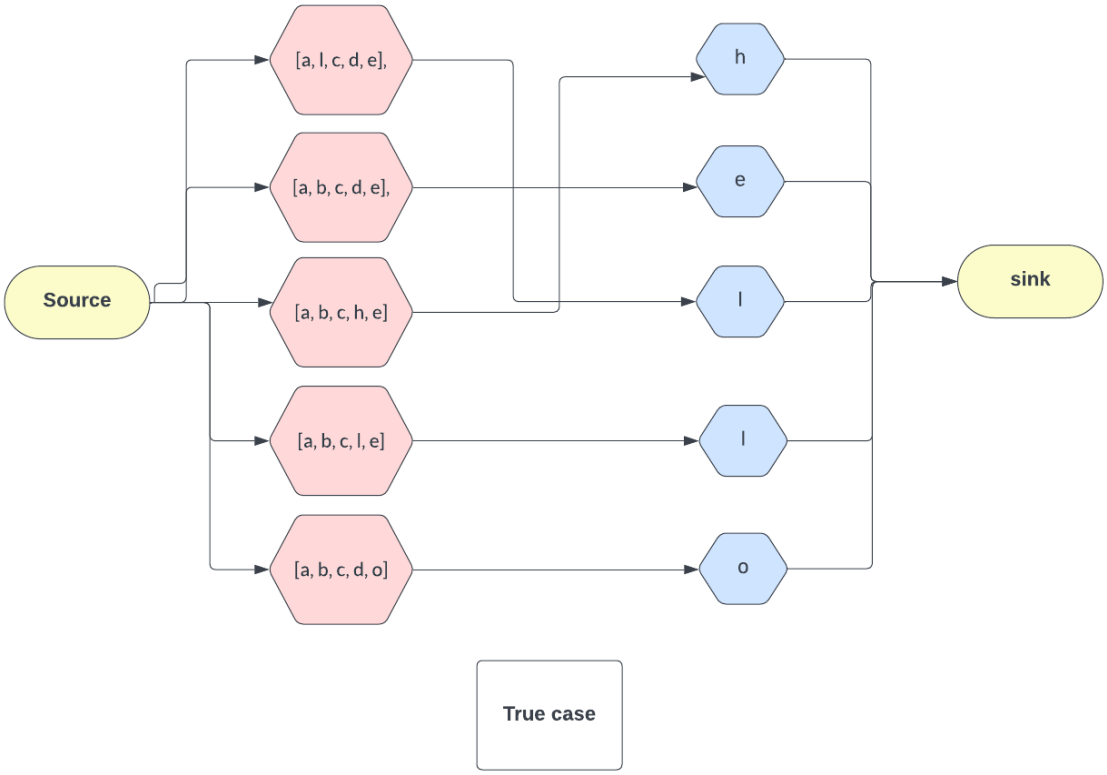
(a) Frame this problem as a network flow problem; Give a graph showing the nodes with sources and sinks and give a rationale for your formulation.

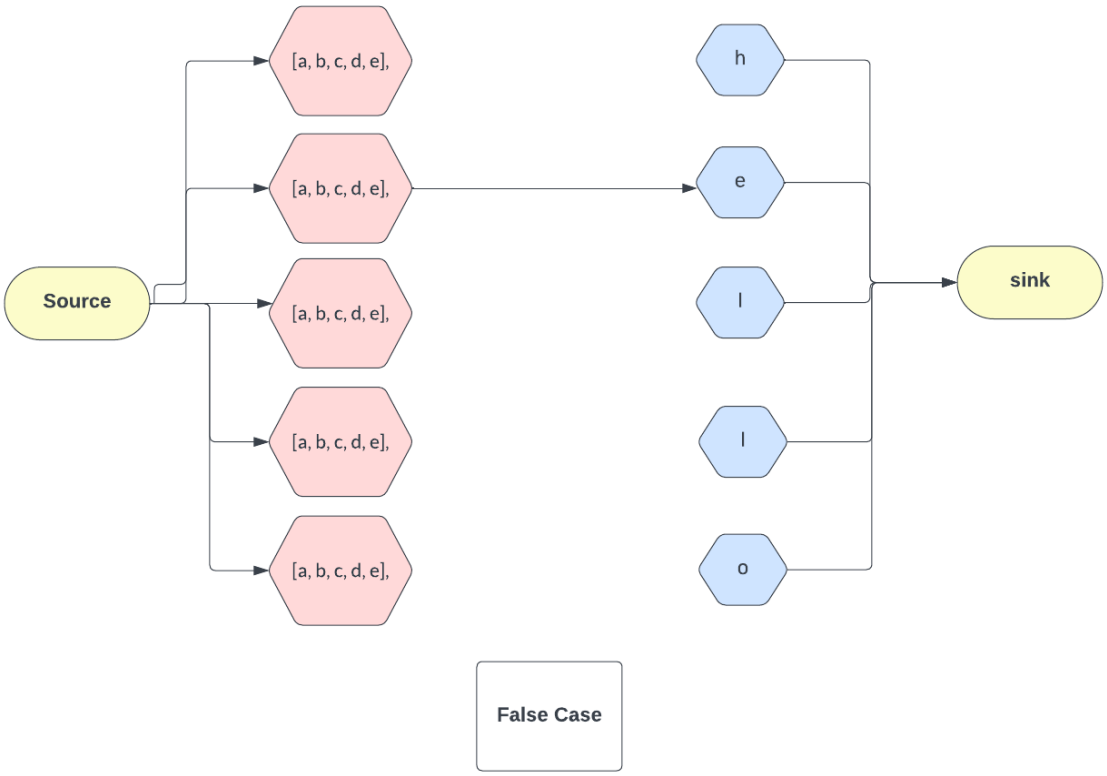
(b) To implement this, what algorithm(s) would you use? What is the running time of the algorithm? Why does it work (You can use exchange arguments to provide a proof)

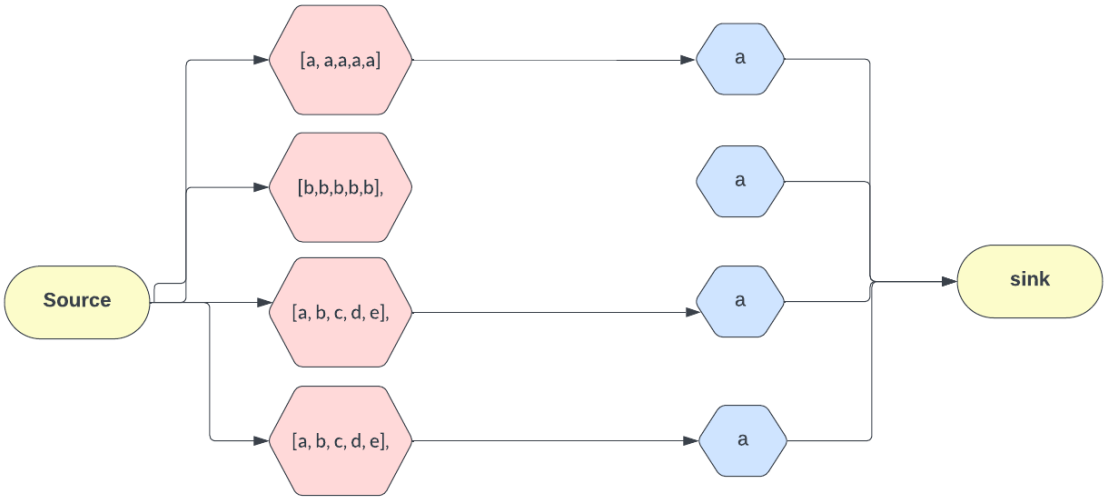
Answer:

(a) Framing the problem as a network flow problem:

There is a 1-capacity edge from Source to each individual character in the dice.There is an edge from each character on the word to T with a capacity of one. There is an edge with capacity 1 linking the word character to the dice character for each pair of characters (word character and dice character) that match if each and every letter has matched with one of the dice non repeating we give it as true below are the examples




Source
[a, b, c, d, e],
[a, b, c, d, e],
[a, b, c, d, e],
[a, b, c, d, e],
[a, b, c, d, e],
h
e
l
l
o
sink
False Case



False Case

To frame this problem as a network flow problem, we can model the given word construction as a bipartite matching problem. We can create a flow network G(V, E) with a source S and a sink T. The vertices in the network will be divided into two sets: the set of characters in the word and the set of characters on the dice. Each edge in the network represents the possibility of matching a character from the word to a character on the dice, and the capacity of the edge is set to 1 to indicate that each dice can only be used once.

Nodes in the network:

- The source node S, representing the starting point of the flow.
- The sink node T, representing the destination of the flow.
- The vertices from the word, representing the characters in the word.
- The vertices from the dice, representing the characters on the dice.

Edges in the network:

- For each character in the word, there is an edge from S to that character with capacity 1.
- For each character on the dice, there is an edge from that character to T with capacity 1.
- For each pair of characters (word character and dice character) that match, there is an edge connecting the word character to the dice character with capacity 1.

Rationale for the formulation:
The objective is to find a matching that covers all the characters in the word using the characters on the dice. This can be achieved by finding a maximum flow in the flow network G from the source S to the sink T. If the maximum flow is equal to the length of the word, it means that all characters in the word can be matched with the characters on the dice, and the word can be constructed.

(b) Algorithm and Proof of Correctness:

To implement this, we can use any maximum flow algorithm like the Ford-Fulkerson algorithm or the Edmonds-Karp algorithm. The running time of the maximum flow algorithm will depend on the specific implementation, but it will generally be efficient.

Algorithm Steps:

1. Create the flow network G based on the word and dice.
2. Run a maximum flow algorithm on G to find the maximum flow from S to T.
3. Check if the maximum flow is equal to the length of the word.
4. If the maximum flow is equal to the length of the word, return True; otherwise, return False.

Proof of Correctness:

The maximum flow algorithm ensures that the flow in the network G corresponds to a matching between the characters in the word and the characters on the dice. Since the capacity of each edge is 1, each dice character can be used at most once to match a word character. Therefore, if the maximum flow is equal to the length of the word, it means that all characters in the word can be matched with the characters on the dice, and the word can be constructed.

Running Time of the Algorithm:

The running time of the maximum flow algorithm will depend on the specific implementation used. In practice, the Ford-Fulkerson algorithm with the Edmonds-Karp implementation is often used due to its efficiency, which has a time complexity of $O(E * V^2)$, where E is the number of edges and V is the number of vertices in the flow network G. Since the number of vertices and edges in G is proportional to the size of the word and the number of dice, the overall time complexity of the algorithm will be reasonable for practical purposes.