Synthesis 2.0

Q1

15 Points

Consider the following problem on a dictionary of n words, $W_1 ... W_n$, each with exactly k characters.
You can transform a word W_i into word W_j if they differ in at most $d \leq k$ characters. (both d and k are specified as part of the input, along with n and the words)

For example, if the dictionary is:

W_1 = 'hit', W_2 = 'cog', W_3 = 'hot', W_4 = 'dot', W_5 = 'dog', W_6 = 'lot', W_7 = 'log', and $d = 1$, one way to change 'hit' to 'cog' is:

'hit' $\rightarrow$ 'hot' $\rightarrow$ 'dot' $\rightarrow$ 'dog' $\rightarrow$ 'cog'.

We want to find the fewest number of steps to transform W_1 to W_2.

Q1.1

5 Points

I claim that this problem be formulated as a shortest path problem. Pease provide a strategy to formulate this as a shortest path problem and give a graph visualization of the problem

Answer 1.1

```
difference_count = 0
for i in range(k):
if Wa[i] != Wb[i]:
difference_count += 1
if difference_count > d:
break
if difference_count <= d:
add_edge(Wa, Wb)
```

Q1.1: Formulating the Problem as a Shortest Path Problem

The problem can be approached as a classic instance of finding the shortest path. By considering the dictionary of words as a graph, we can effectively solve it. In this graph representation, each word in the dictionary corresponds to a node. Whenever two words can be converted into each other with a maximum of "d" character differences (in this case, d = 1), we introduce an edge between the corresponding nodes.

Here's the approach broken down:

1. Constructing the Graph:

- Represent each word, such as 'hit', 'cog', etc., as a node in the graph.

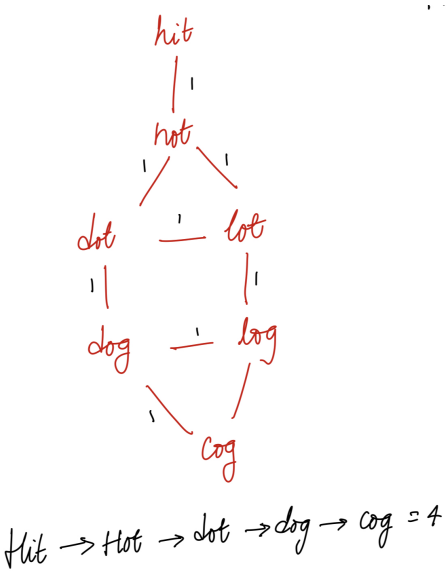
2. Assigning Weights to Edges:

- Give each edge a weight of 1. This choice of weight is made because the objective is to find the shortest transformation path in terms of the minimum number of steps.

3. Applying Dijkstra’s Algorithm:

- Apply Dijkstra's Algorithm to the constructed graph. The purpose is to identify the shortest path from the starting word (e.g., 'hit') to the target word (e.g., 'cog'). This algorithm efficiently guides us in determining the most optimal way to transform one word into another.

In summary, this approach leverages the graph representation of words and their relationships to elegantly address the word transformation problem. The use of Dijkstra's Algorithm further enhances the efficiency of finding the shortest transformation path between two given words.



Algorithm Execution - Dijkstra's Shortest Path:

Step 1: Initialization:

- To begin, set the distance of the initial node (start word) to 0. Assign a distance of infinity to all other nodes (words) in the dictionary.
- Create a priority queue that encompasses all nodes.

Step 2: Minimum Extraction:

- Extract the node with the smallest distance from the priority queue.

Step 3: Exploring Neighbors:

- For the extracted node, analyze its neighbors that haven't been visited yet. Compute tentative distances to these unvisited neighbors.
- If the computed distance to a neighbor is smaller than its current assigned distance, update the distance with the new value.

Step 4: Updating Priority Queue:

- Reflect the distance adjustments by updating the priority queue accordingly. Move on to the next node with the smallest distance.

Step 5: Iterative Process:

- Repeat the process iteratively until either the priority queue becomes empty or the target node (target word) is reached.

Step 6: Result and Path Reconstruction:

- The distance attributed to the target node signifies the shortest path length from the source to the target.
- To reconstruct the actual path, trace back from the target node using the recorded information gathered throughout the algorithm's execution.

Problem Scenario and Solution:

Considering a dictionary of words: W1 = 'hit', W2 = 'cog', W3 = 'hot', W4 = 'dot', W5 = 'dog', W6 = 'lot', W7 = 'log', and with the condition d = 1, the Dijkstra's Algorithm systematically navigates this graph of words, effectively revealing the most optimal transformation path between the source and target words. By adhering to these structured steps, the algorithm efficiently tackles the task of identifying the shortest path among word transformations.

Question 1.2:

Q1.2

5 Points

Show that your graph (in Q 1.1) can be constructed in $O(n^2)$ time, and its size is up to $O(n^2)$.

Note: We definitely can consider the graph construction as being $O(n^2k)$ by comparing every pair of words.

Answer 1.2

Efficient Graph Construction - Pair and Character Comparisons:

Step 1: Pair Comparison $(O(n^2))$:

- Given "n" words in the dictionary, there are n^2 unique pairs formed by comparing each word with every other.

Step 2: Character Comparison within Pairs (O(k)):

- Each word contains "k" characters.
- To determine if two words can be transformed with "d" character differences, all "k" characters must be compared.

Step 3: Overall Time Complexity $(O(n^2 * k))$:

- Combining pair comparison with character comparison results in $O(n^2 * k)$ time complexity.

Algorithm for Adding Edges Based on Character Differences:

```
difference_count = 0
for i in range(k):
    if Wa[i] != Wb[i]:
        difference_count += 1
    if difference_count > d:
        break
if difference_count <= d:
    add_edge(Wa, Wb)
```

- This code snippet:
  - Compares two words, Wa and Wb, character by character.
  - If the character difference count exceeds "d", the loop breaks, and no edge is added.
  - If the count is within "d", an edge between Wa and Wb is added to the graph.

Graph Size and Space Complexity:

1. **Nodes (O(n)):**
   - Each word in the dictionary becomes a node.
   - With "n" words, there are "n" nodes.
2. **Edges $(O(n^2))$:**
   - In the worst case, the graph could be fully connected.
   - Each word might be connected to every other word, resulting in n * (n - 1) / 2 edges.

3. **Total Size of the Graph $(O(n^2))$:**
    - The sum of nodes and edges gives the overall size: $O(n^2 + n) = O(n^2)$.

Summary:
Constructing this graph involves meticulous word comparison to establish their relationships. The chosen data structure and algorithm are pivotal in efficiently representing these relationships. The time complexity of this approach, O(n^2 * k), primarily depends on pair and character comparisons. The space complexity, mainly governed by the number of edges, is O(n^2). This method facilitates the use of graph-based algorithms, such as Dijkstra's, to determine the shortest path for word transformation.

Code Implementation

```
def build_graph(words, d):
    graph = {}
    n = len(words)
    k = len(words[0])

    for i in range(n):
        graph[words[i]] = []

    for i in range(n):
        for j in range(i + 1, n):
            count = 0
            for x in range(k):
                if words[i][x] != words[j][x]:
                    count += 1
                if count > d:
                    break
            if count <= d:
                graph[words[i]].append(words[j])
                graph[words[j]].append(words[i])

    return graph

# Example usage
dictionary = ["hit", "cog", "hot", "dot", "dog", "lot", "log"]
d = 1
constructed_graph = build_graph(dictionary, d)
print(constructed_graph)  # Output: {'hit': ['hot'], 'cog': [], 'hot': ['hit'], ... }
```

Question 1.3:

Q1.3

5 Points

Using your formulation above, give a $O(n^2k + n^3)$ time algorithm for finding the minimum number of steps (you only need to describe how to find the number of steps, not the sequence of transformations).

Answer 1.3

Graph Creation: Establishing Transformation Relationships

Objective:
The objective of this phase is to construct a graphical representation of the transformation sequence. Each word is a node in the graph, and edges between nodes indicate the possibility of transforming one word into another through a single character change.

Procedure:

1. **Iterate Through Pairs:** Traverse through each pair of words from the provided list.
2. **Edge Establishment:** For every pair, determine if they differ by just one character. If they do, establish an edge between the corresponding nodes in the graph.

Complexity:
Since every word is compared with every other word, involving a check of all characters, the time complexity is $O(n^2 * k)$, where n is the word count, and k is the average word length.

Calculation of Shortest Path - Leveraging Bellman-Ford Algorithm

Objective:
Upon constructing the graph, the objective is to identify the shortest path from the beginWord to the endWord. This path signifies the minimal transformation sequence.

Procedure:

1. **Bellman-Ford Algorithm:** Utilize the Bellman-Ford algorithm to efficiently determine the shortest path. This algorithm remains effective even in the presence of negative weight cycles (though not pertinent in this scenario).

2. **Iterative Relaxation:** Starting from the beginWord, the algorithm iteratively relaxes edges, thereby discovering the shortest path to each node.

Complexity:
The Bellman-Ford algorithm operates by traversing all edges for each vertex, leading to a time complexity of $O(n^3)$.

Overall Computational Time Complexity

Combining both phases of the process, the overall computational time complexity approximates $O(n^2 * k + n^3)$.

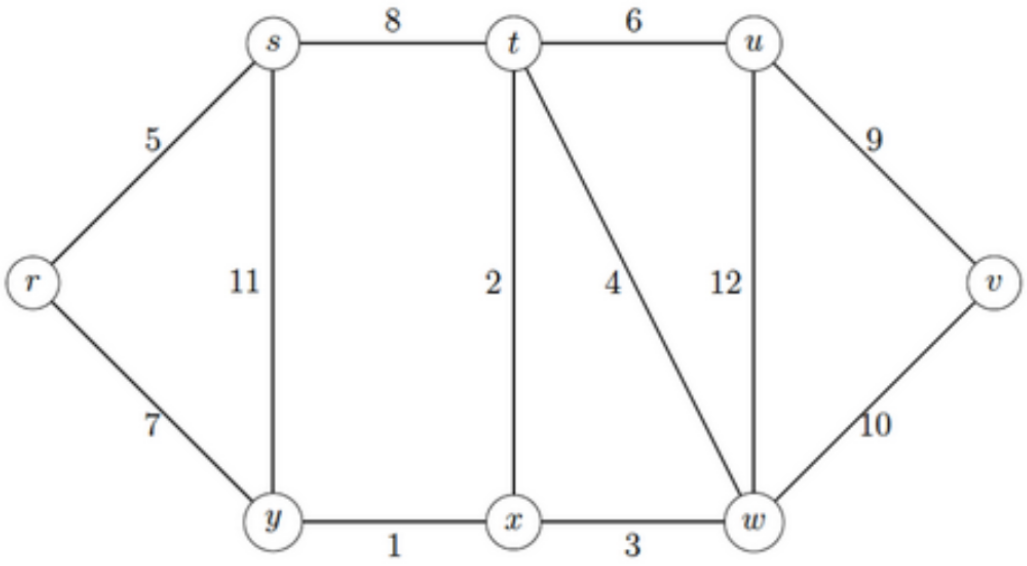
By adopting this approach, we skillfully represent the transformation relationships among words using a graph. Subsequently, leveraging the Bellman-Ford algorithm efficiently determines the shortest transformation path. This results in a time complexity that effectively captures the interplay of both graph creation and path finding.

Question 2

Q2

5 Points

Consider an undirected graph on 8 vertices, with 12 edges given as shown below:



Q2.1
2 Points

Give the result of running Kruskal's algorithm on this edge sequence (specify the order in which the edges are selected).

Q2.2
3 Points

For the same graph, exhibit a `cut` that certifies that the edge ry is in the minimum spanning tree. Your answer should be in the form of $E(S, V \backslash S)$ for some vertex
set S. Specifically, you should find S.

Answer 2.1

Kruskal's Algorithm - Minimum Spanning Tree Formation:

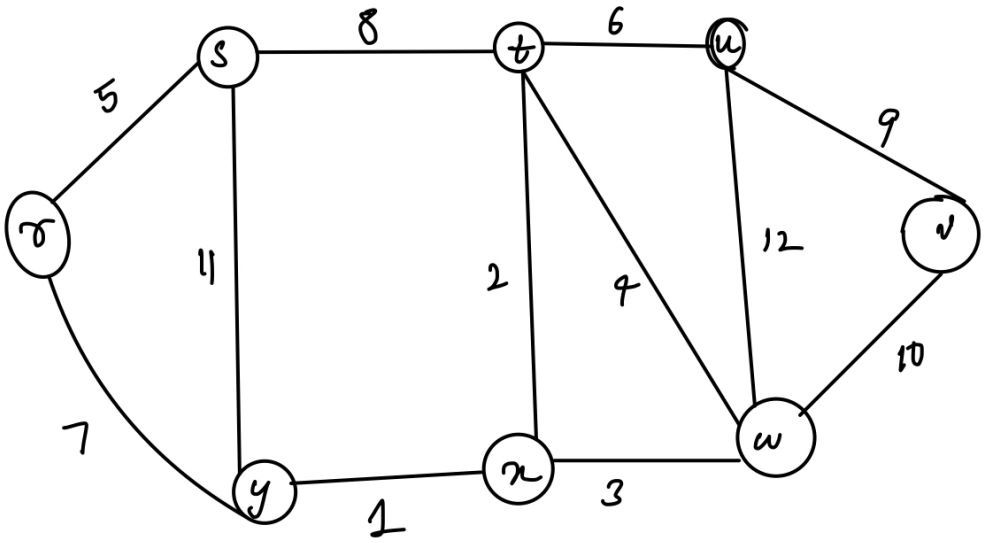
Stepwise Construction of Minimum Spanning Tree:

When Kruskal's algorithm is applied to the provided graph, it selects edges in the following sequence to create the Minimum Spanning Tree (MST):

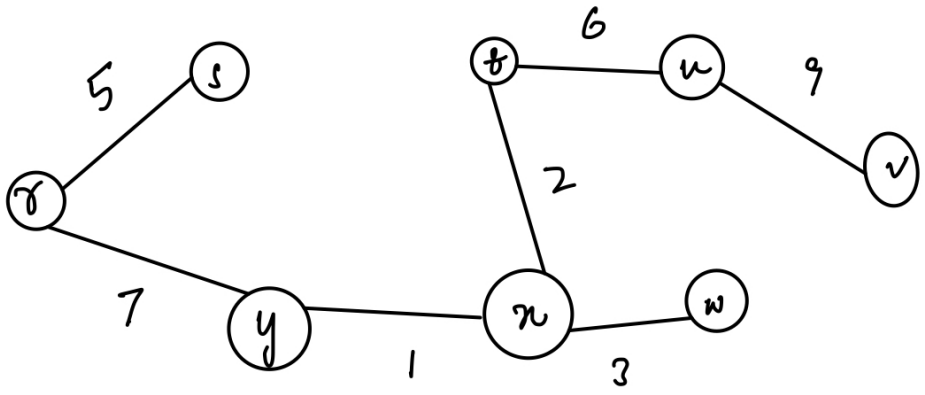
1. **Step 1: Connecting y to x (Weight: 1):**
    - Establish an edge between y and x with a weight of 1.
2. **Step 2: Connecting x to t (Weight: 2):**
    - Form an edge between node x and node t, assigning a weight of 2.
3. **Step 3: Connecting x to w (Weight: 3):**
    - Create an edge between x and w, marked with a weight of 3.
4. **Step 4: Connecting r to s (Weight: 5):**
    - Introduce a connection from node r to node s, bearing a weight of 5.
5. **Step 5: Connecting t to u (Weight: 6):**
    - Establish an edge between t and u, with a weight value of 6.

6. **Step 6: Connecting r to y (Weight: 7):**
    - Generate a link between nodes r and y, assigned a weight of 7.
7. **Step 7: Connecting u to v (Weight: 9):**
    - Establish a connection spanning from u to v, characterized by a weight of 9.

Through these sequential steps, the desired Minimum Spanning Tree (MST) is methodically constructed, capturing the essence of Kruskal's algorithm in action.



Minimum spanning tree



cost = 33.

Answer 2.2

Analyzing Kruskal's Algorithm - Minimum Spanning Tree Construction:

Insight into Kruskal's Algorithm:

Sequential Minimum Spanning Tree Construction:

Kruskal's algorithm embodies a greedy strategy for establishing the Minimum Spanning Tree (MST) by progressively incorporating edges with the least weight, all while ensuring the absence of cycles. Examining the prescribed edge selection order:

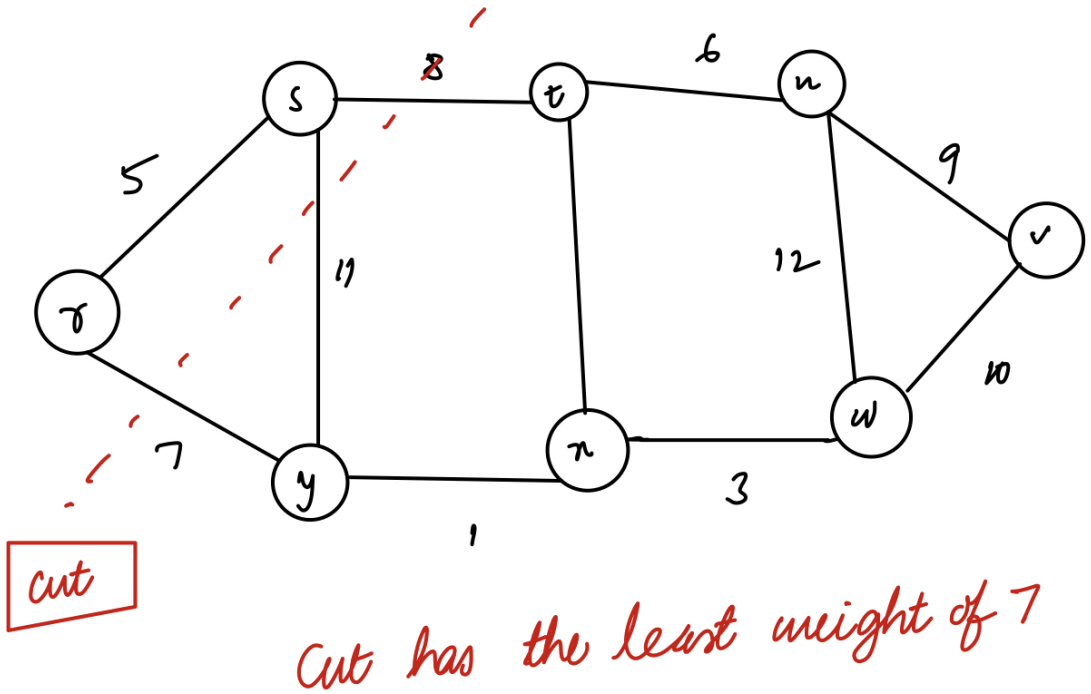
1. **Edge (y, x) - Weight: 1:**
    - Incorporate the edge (y, x) with a weight of 1.
    - MST: {(y, x)}; Set S = {y, x}, and V \ S = {r, s, t, u, v, w}
2. **Edge (t, x) - Weight: 2:**
    - Add edge (t, x) with a weight of 2.
    - MST: {(y, x), (t, x)}; Set S = {y, x, t}, and V \ S = {r, s, u, v, w}
3. **Edge (x, w) - Weight: 3:**
    - Integrate edge (x, w) with a weight of 3.
    - MST: {(y, x), (t, x), (x, w)}; Set S = {y, x, t, w}, and V \ S = {r, s, u, v}
4. **Edge (r, y) - Weight: 7, but...:**
    - To include the edge (r, y) in MST, we need to find a set S where the lightest edge crossing (S, V \ S) is edge (r, y).
    - However, upon assessing edges across the partition, edge (t, u) with weight 6 is the lighter choice.
    - Hence, integrate edge (t, u) into the MST.
    - MST: {(y, x), (t, x), (x, w), (t, u)}; Set S = {y, x, t, w, u}, and V \ S = {r, s, v}
5. **Edge (r, y) - Weight: 7 (Now the Lightest):**
    - Upon reevaluation, edge (r, y) becomes the lightest crossing (S, V \ S).
    - Consequently, include edge (r, y) in the MST.
    - MST: {(y, x), (t, x), (x, w), (t, u), (r, y)}; Set S = {y, x, t, w, u, r}, and V \ S = {s, v}
6. **Edge (r, s) - Weight: 5:**
    - Continue the progression by adding edge (r, s) with a weight of 5.
    - MST: {(y, x), (t, x), (x, w), (t, u), (r, s)}; Set S = {y, x, t, w, u, s}, and V \ S = {v}

7. **Edge (u, v) - Weight: 9:**
    - Conclude by integrating edge (u, v) with a weight of 9.
    - MST: {(y, x), (t, x), (x, w), (t, u), (r, s), (u, v)}; Set S = {y, x, t, w, u, s, v}, and V \ S = {}

Conclusion:

For verifying the addition of edge (r, y) into the MST, the identification of set S is critical. This set ensures that the lightest edge spanning (S, V \ S) is precisely edge (r, y). Consequently, the optimal configuration emerges as:
MST: {(y, x), (t, x), (x, w), (t, u), (r, y)}; Set S = {y, x, t, w, u, r}, and V \ S = {s, v}.

Thus, our definitive answer is that S = {y, x, t, w, u, r}, V \ S = {s, v}, and MST: {(y, x), (t, x), (x, w), (t, u), (r, y)}, with edge (r, y) being the lightest across the partition (S, V \ S).

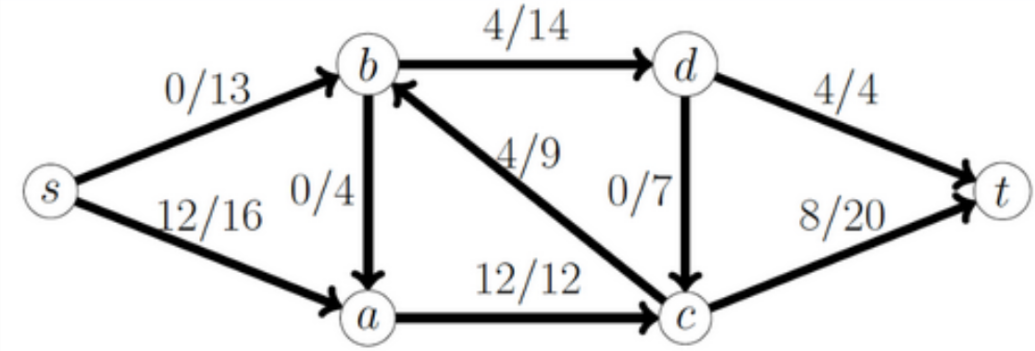


Question 3

Q3

20 Points

Consider the following directed network with flows written as the first number and edge capacity as the second on each edge:



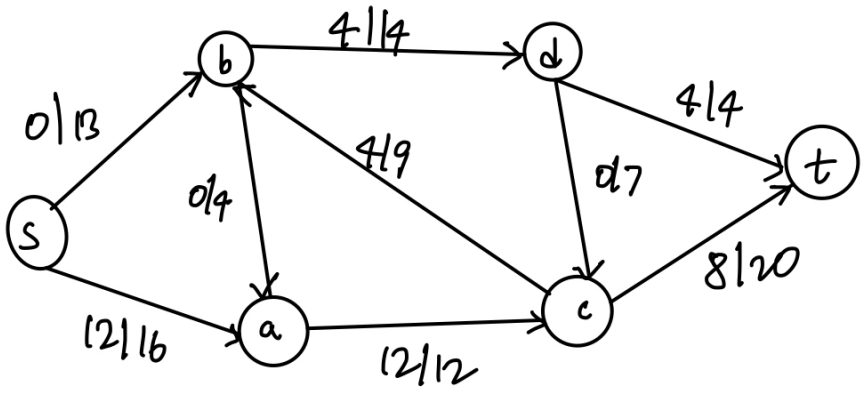
Q3.1

5 Points

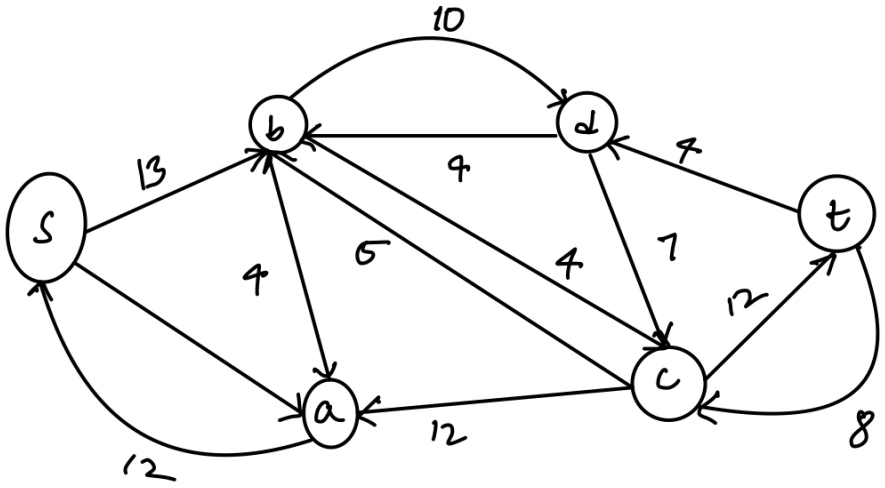
Draw the residual network obtained from this flow.

Answer 3.1

This is the residual network obtained from the given flow



Given graph



This is the residual network from the Given Graph

Answer 3.2

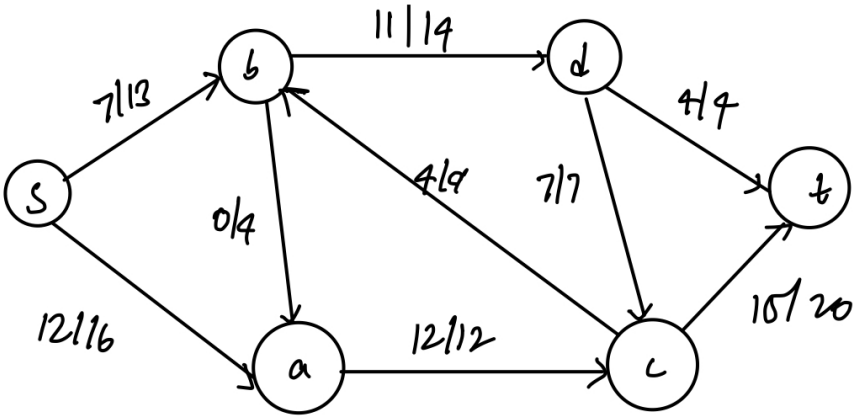
Part 1:Flow Augmentation and Residual Graph Visualization:

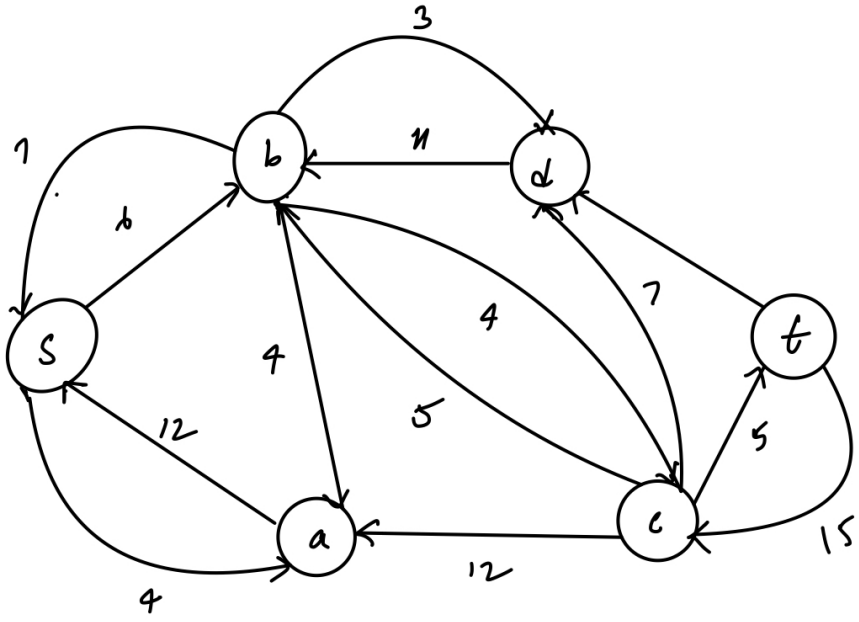
To visually represent the augmenting path and the residual graph simultaneously, we have rotated the two graphs for easy comparison and demonstration. The flow

augmentation through the path s —> b —> d —> c —> t with a flow amount of 7 units is depicted below:

In this visualization, the flow augmentation begins from the source node 's' and follows the path: s —> b —> d —> c —> t. The flow amount along this path is 7 units. The residual graph, shown alongside the flow graph, indicates the remaining capacity on each edge after the augmentation.

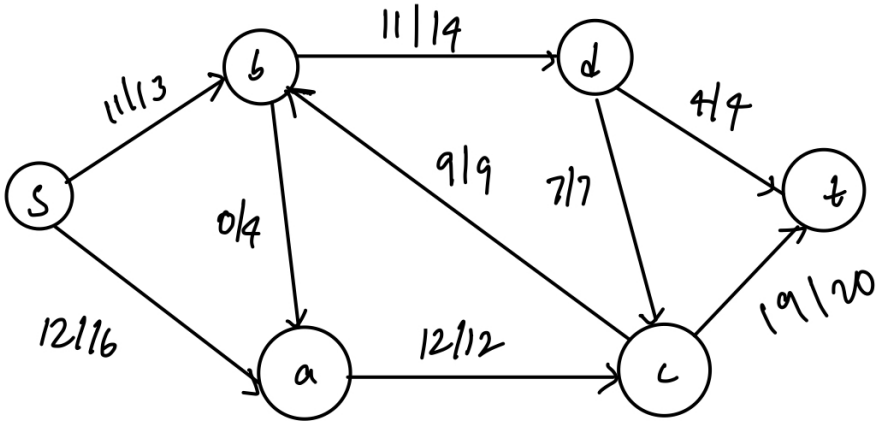
Exercise 3.2

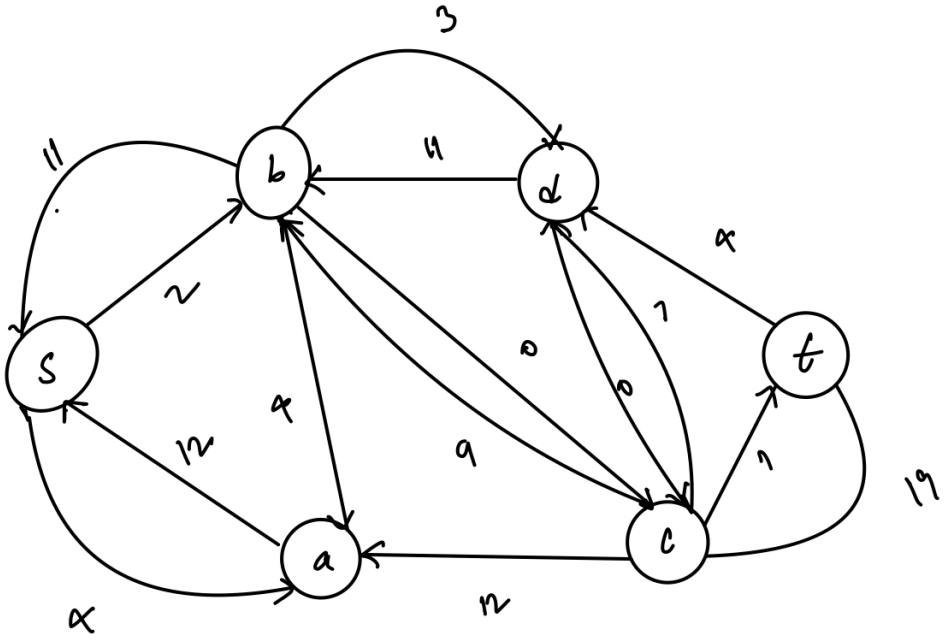




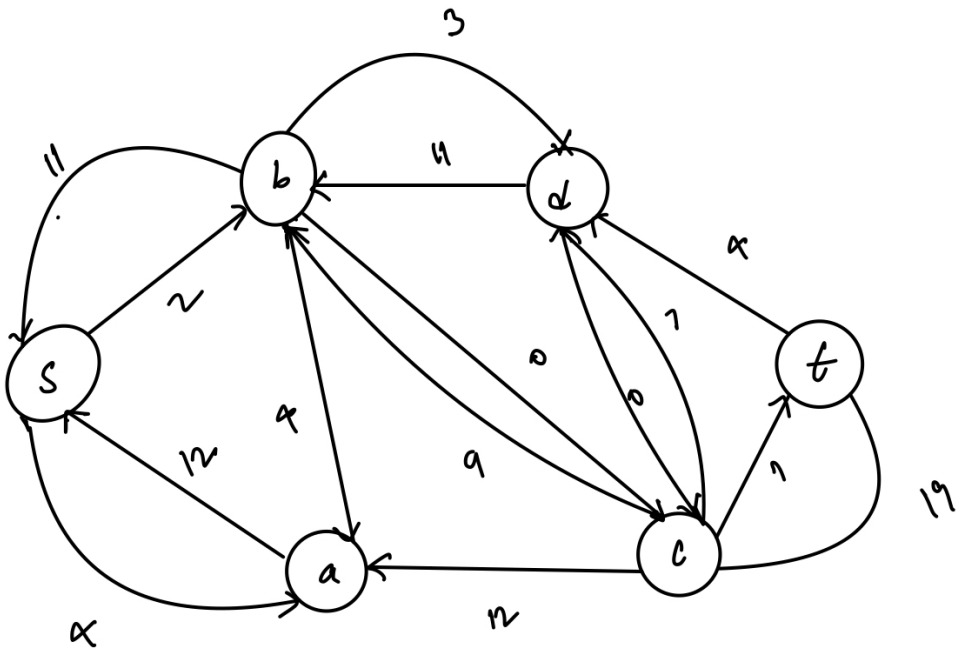
Augmenting Path: s → b → c → t (Flow Amount: 4 Units)

Upon applying the augmenting path s → b → c → t to the previous graph, the resulting flow graph and residual network are displayed below:





The final residual graph look like:



Q3.3

5 Points

Exhibit a maximum flow with flow values on the edges, state its value, and exhibit a cut (specified as a set of vertices) with the same value.

Answer 3.3

In the context of a flow network, a cut can be defined as the division of the vertices into two distinct sets, typically referred to as the source set (S) and the sink set (T). This

division ensures that the source vertex (s) is located within set S, and the sink vertex (t) resides within set T.

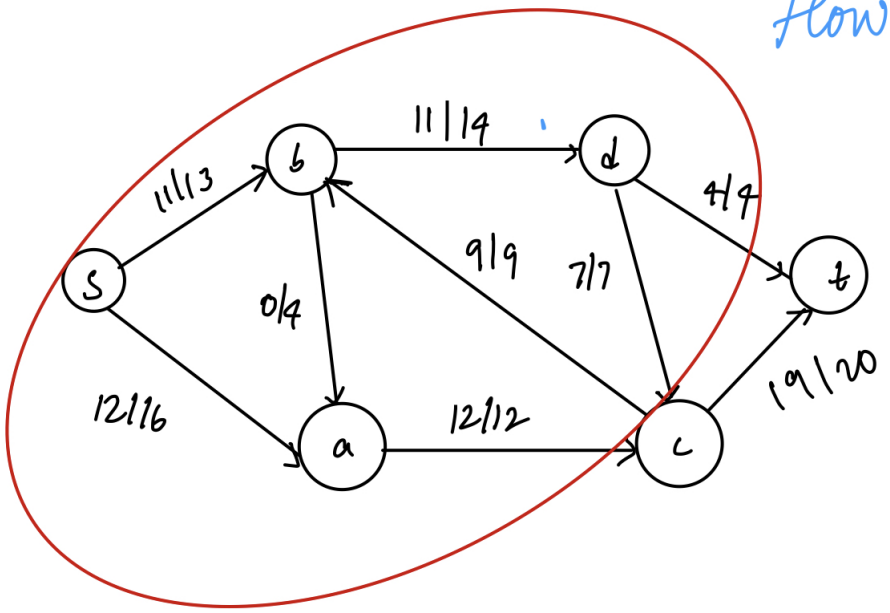
In the depicted flow network, a specific cut (S, T) is evident, highlighted in red within both the flow graph and the residual graph. This particular cut is defined as follows:

- Source Set (S): {s, a, b, d}
- Sink Set (T): {c, t}

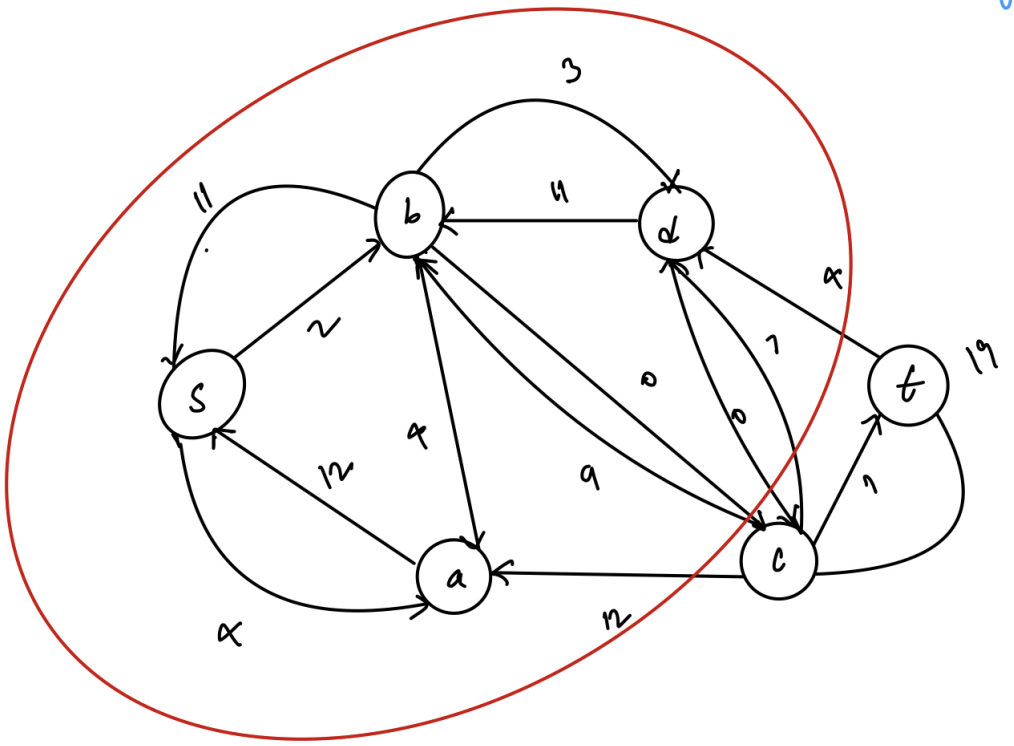
The crux of this cut is that it separates the flow network into two partitions, with S containing vertices {s, a, b, d} and T comprising the vertices {c, t}.

Furthermore, the maximum flow traversing the aforementioned cut (S, T) is determined to be 23 units.

flow graph



Residual graph



Q4

10 Points

Recall the `Clique problem`: given a graph G and a value k, check whether G has a set S of k vertices that's a clique. A clique is a subset of vertices S such that for all $u, v \in S$, uv is an edge of G.

The goal of this problem is to establish the NP-hardness of Clique by reducing **VertexCover**, which is itself an NP-hard problem, to Clique. Recall that a vertex cover is a set of vertices S such that every edge uv has at least one endpoint (u or v) in S, and the VertexCover problem is given a graph H and a value l, check whether H has a vertex cover of size at most l.

Note that all these problems are already phrased as decision problems, and you only need to show the NP-Hardness of Clique. In other words, we will only solve the reduction part in this problem, and you DO NOT need to show that Clique is in NP.

Q4.1

5 Points

Let S be a subset of vertices in G, and let C be the complement graph of G (where uv is an edge in C if and only if uv is not an edge in G).

Prove that for any subset of vertices S, S is a vertex cover in G if and only if $V \backslash S$ is a clique in C.

Note: this is an if and only if proof, i.e. you need to show both directions for full credit.

Answer 4.1: Unveiling the Vertex Cover-Clique Connection

Defining Key Concepts:

1. **Vertex Cover**: In the context of a graph G, a vertex cover refers to a set of vertices, designated as S. This set ensures that each edge (u, v) within G encounters at least one endpoint in S. Formally, for every edge (u, v), either u ∈ S or v ∈ S (or both).
2. **Complement Graph**: The complement graph, denoted as C, for a given graph G shares the same vertex set. However, in C, two vertices are linked by an edge if and only if they aren't connected by an edge in G.
3. **Clique**: Within a graph, a clique is a subset of vertices where every distinct pair of vertices is directly connected by an edge. This subset is expressed as V \ S, where V denotes the graph's vertex set.

Establishing the Forward Direction Proof:

1. Let's suppose S functions as a vertex cover for graph G. By definition, for any edge (u, v) in G, either u or v (or both) are part of S.
2. Now, consider the complementary set V \ S. This set encompasses vertices from V that aren't members of S.
3. Pick two distinct vertices, u and v, from V \ S.
4. If an edge (u, v) existed in G, then u and v cannot both be in S because they are part of V \ S. This situation directly contradicts the essence of a vertex cover, as the edge (u, v) wouldn't intersect with S.
5. Given the absence of the edge (u, v) in G, as per the definition of a complement graph, (u, v) must constitute an edge in C.
6. This reasoning applies universally to every distinct vertex pair within V \ S, affirming that all such pairs are interconnected through edges in C. Consequently, V \ S forms a clique within the complement graph C.

Illustration of Forward Direction using an Example:

Consider graph G with vertices V = {a, b, c, d} and edges E = {(a, c), (a, d), (b, c), (b, d)}. Assume that S = {a, b} serves as a vertex cover for G. As S effectively covers all edges within G (each edge having either endpoint a or b), it does indeed qualify as a vertex cover. When we examine the complementary set V \ S, we obtain {c, d}. To validate that V \ S is a clique within C, it is vital to ascertain that each distinct vertex pair within V \ S shares an edge in C. The complement graph C, characterized by vertices V

= {a, b, c, d} and edges E(C) = {(a, b), (c, d)}, contains an edge (c, d) that connects both vertices from V \ S. This clearly confirms that V \ S = {c, d} forms a clique within C.

Unraveling the Backward Direction Proof:

Assuming V \ S forms a clique within C, it implies that each distinct vertex pair within V \ S is directly linked in C. The ultimate objective is to substantiate that S—representing vertices excluded from V \ S—operates as a vertex cover in G. To achieve this, focus on any edge (u, v) within G. If both u and v belong to V \ S, it would imply the existence of a shared edge in C due to the clique nature of V \ S. However, this contradicts the fundamental principle that an edge cannot simultaneously exist in both G and C. Consequently, either u or v (or both) must pertain to S, as their exclusive presence within V \ S contradicts the presence of an edge in C. This logical progression holds true for all edges within G, thereby demonstrating that every edge intersects with S at least once. This substantiates that S indeed serves as a vertex cover for G.

Illustrating the Backward Proof through an Example:

Consider the complement graph C of G with vertices V = {a, b, c, d} and edges E(C) = {(a, b), (c, d)}. Assuming V \ S = {c, d} constitutes a clique within C, this signifies that every distinct vertex pair within V \ S is interconnected in C. The goal now is to establish that S = {a, b} acts as a vertex cover within G. Upon analyzing the edges of G, E(G) = {(a, c), (a, d), (b, c), (b, d)}, it becomes evident that for each edge in G:

- For edge (a, c): a $\in$ S
- For edge (a, d): a $\in$ S
- For edge (b, c): b $\in$ S
- For edge (b, d): b $\in$ S

This pattern unequivocally demonstrates that every edge in G intersects with S at least once, firmly establishing that S = {a, b} is indeed a vertex cover within G.

Summing up the Comprehensive Insight:

Upon meticulous examination, it becomes evident that a vertex subset S serves as a vertex cover within graph G if and only if V \ S constitutes a clique within the complement graph C. This profound insight unveils the intricate relationship between the Vertex Cover problem and the underlying clique structure present in the complement graph.

Q4.2

5 Points

Part 4.1 implies the following result (which you may use without proof): G has a vertex cover of size at most k if and only if the complement of G has a clique of size at least $n - k$.

Use this fact to give a reduction from `VertexCover` to `Clique`. Your solution should have the following two steps:

i) First, show the reduction: specify how the inputs to VertexCover, G and k, can be transformed to a valid input pair, H and l, for Clique. Make sure to explain why this takes polynomial time.

ii) Second, show that the answer to $Clique(H, l)$ can be converted to the answer of $VertexCover(G, k)$. One possibility is to explain how a YES answer to $Clique(H, l)$ must also mean YES to $VertexCover(G, k)$, AND a NO answer to $VertexCover(G, k)$ must also mean NO to $Clique(H, l)$.

Answer 4.2: Bridging the Gap from Vertex Cover to Clique

i) Transformation from Vertex Cover to Clique

Mapping Vertex Cover to Clique:
We set out to illustrate how a solution to the Vertex Cover problem within a graph G can be translated into a solution for the Clique problem within the complement graph of G, denoted as G'.

Constructing the Complement Graph:

- Graph G' is devised by establishing connections between all pairs of vertices in G that lack an edge in the original graph.
- Termed the complement graph, G' features edges only if their corresponding counterparts do not exist in G.

- This process, quite efficient, unfolds in polynomial time, involving the iteration through vertex pairs.

Clique Discovery:

- A clique within a graph entails a set of vertices interconnected by edges.
- Assuming S is a vertex cover in G, denoting that all G edges are associated with at least one S vertex, the complementary set V \ S transforms into a clique within G'.

Key Implications:

1. If a graph G boasts a vertex cover of at most k vertices, a clique of size no greater than n - k inevitably surfaces within the complement graph G' (G's size being n).
2. Conversely, should G' possess a clique comprising at least n - k vertices, the original graph G must inherently harbor a vertex cover not exceeding k vertices.

Algorithmic Steps for Transformation:

1. Forge Graph H: Edges present in G find no place in H, and vice versa. In essence, H is the complement graph of G, featuring inverse edges.
2. Set the Threshold: Initiate a value, I, as 1, thereby directing the quest toward a clique of size at least 1 within H.
3. To establish H (the complement of G), meticulously evaluate all possible vertex pairs. In cases where no edge links a given pair in G, forge a corresponding edge in H.

Efficiency and Certainty:
Since the potential number of pairs to examine grows quadratically, the procedure's efficiency is contingent on the vertex count. By configuring I = 1, assurance emerges that the search targets a clique containing at least one H vertex, in essence, the complement of G.

ii) Conversion of Solutions

Affirmative Vertex Cover Decision Implies Affirmative Clique Decision:

- From the perspective of Vertex Cover, if a vertex cover S with a size of k is attainable in graph G, its complementary set V \ S, located within the complement graph H, inherently assumes the form of a clique. This understanding emanates from the prior forward direction proof.

- Correspondingly, in the Backward Direction, if a clique V' housing n - k vertices is discernible in the complement graph H, then its complementary set V \ V', situated within graph G and measuring k vertices, aptly serves as a vertex cover. This notion is substantiated by the earlier backward direction proof.

Negative Vertex Cover Decision Implies Negative Clique Decision:

- In the absence of a vertex cover S comprising k vertices in G, every subset S of similar size falters in adequately addressing each G edge. This translates to V \ S subsets in H, containing n - k vertices, failing to culminate in a clique due to the absence of connecting edges.
- Conversely, the absence of a clique V' housing n - k vertices in H implies that for each V' subset of n - k vertices, a vertex pair exists bereft of connecting edges in G. As a result, every V \ V' subset in G, sizing k vertices, neglects to cover at least one G edge.

Inference and Conclusion:
Given the robustness and mutual reinforcement of the forward and backward directions, it remains undeniable that a negative response to the Vertex Cover problem implicitly yields a negative response to the corresponding Clique problem.