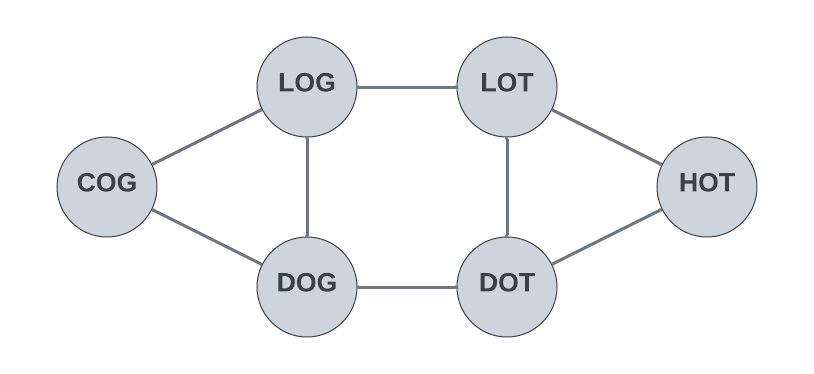
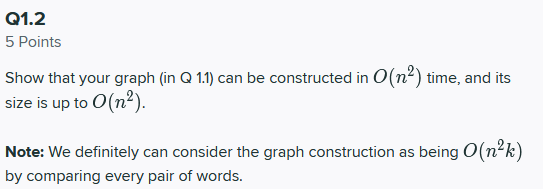


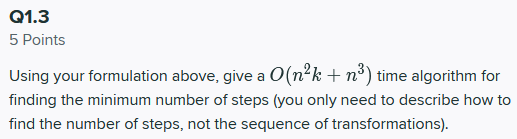
Answer:

* Our objective here is to consider n words, denoted as W1, W2, … Wn where each word has exactly k characters describing it
* We are told we can transform a word into another in which they differ at most d=> k
* A strategy we can take here is a graph strategy in which we present each of the words we are given as nodes.
* Next, we can connect these nodes together between each of the words as edges, based on the changes they require to transform from one word into another.
* A word that can transform into another given one change, will be separated by one edge.
* One can interpret this as a graph in which a given word shares an edge with another word if they have two letters in common, and only one letter needs to be changed.
* We can see a depiction of this as a shortest path problem since we need to reach from one word to another in the shortest distance possible
* The diagram below illustrates this as a shortest path graph:

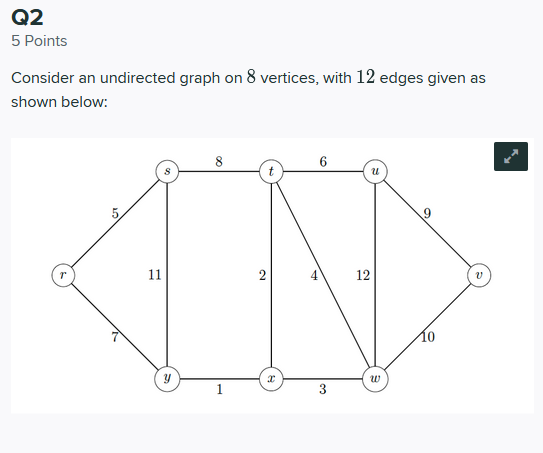


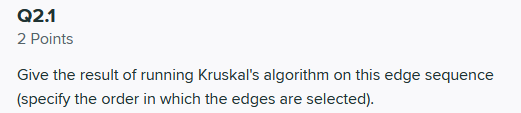


* We can construct the graph above with a time complexity of O(n2)
* To accomplish this, we can view the words W1, W2, … Wi as nodes within in a graph network as shown in the question above. To move from one node to another, we mutate a letter, so we can see this as a directed graph.
* To start, we must create n nodes, one for each of the words provided in the list above.
* Next, we need to iterate over the words in the form of a loop, which will be done n times.
  + For each we need to check a conversion can be made from one word to another.
* Given the nested nature of these two loops, each going through the results n times, we will see a time complexity of O(n2)
* We can represent the number of edges in this graph as nC2
* Asymptotically, this is **O(n2)**

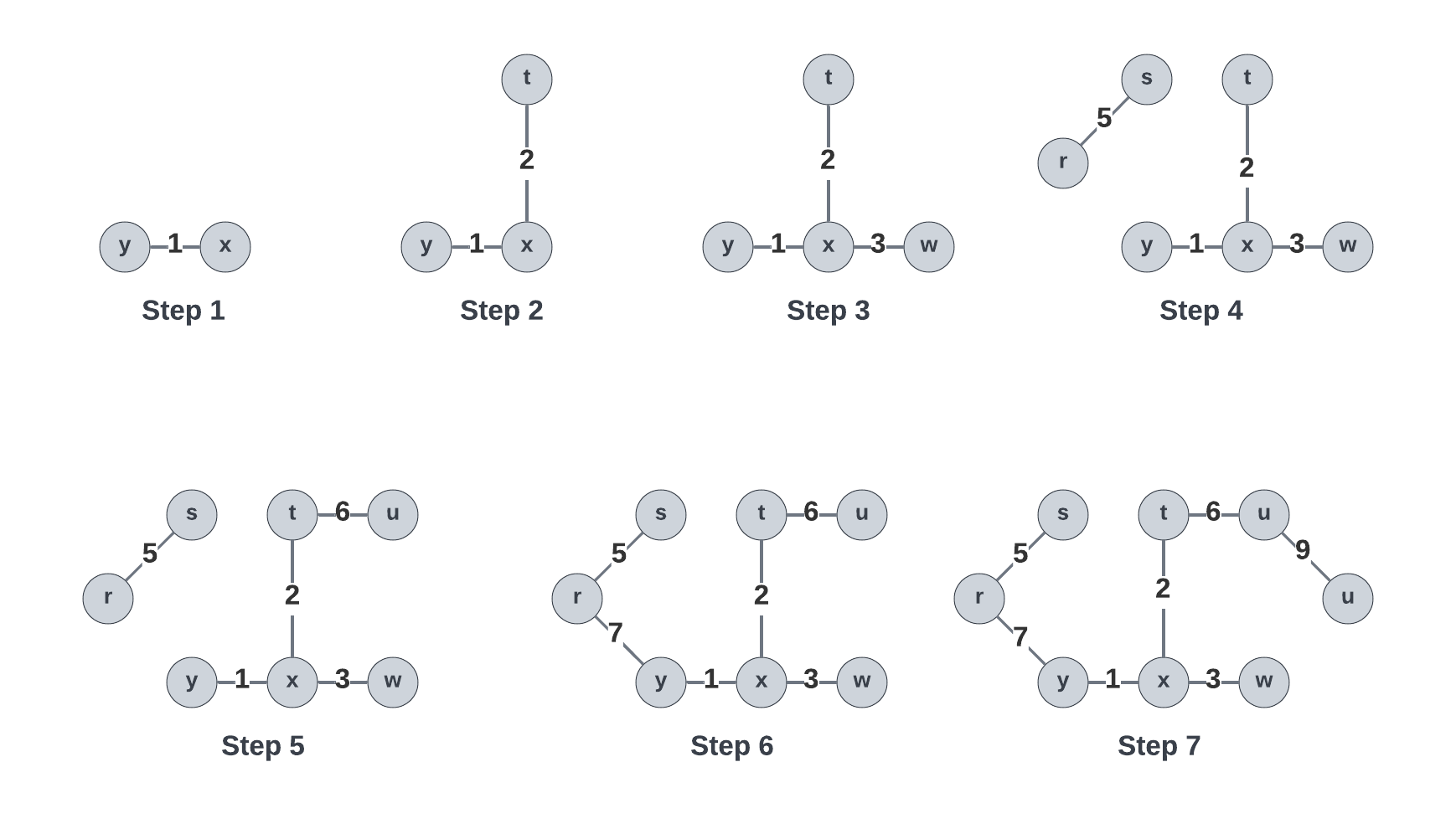


* Our objective here is to give an algorithm with a time of O(n2k+n3) to find the minimum number of steps
* We can likely use the Bellman-Form algorithm to accomplish this, which has its own time complexity of O(n3)
* We select this algorithm because it operates by underestimating the length of the path from that beginning vertex to all others, and it iteratively discovers new paths that are shorter.
* The concept here of shortest distance can help with the use case of words above.
* Algorithm:
  + The first step here is to create a list of the graph’s edges, generally presented in the form of an adjacency list or what not.
  + Next, The number of iterations are calculated. This is generally done in the form of V-1 given that the algorithm is formed around the concept of shortest-distance.
  + Next, a loop is entered in which we iterate such that for each edge u-v in the graph we compute the path lengths
    - If the distance v is greater than distance u plus the edge weight uv, then the distance v is set to the distance u plus that edge weight.
  + Finally, that distance is returned.
* Form the previous question we saw that the construction of the graph can take O(n2k), coming from the vertices and edges in the Graph itself, and contrasting words.
* Therefore, given the use of the algorithm, we will see a runtime of O(n2k+n3)

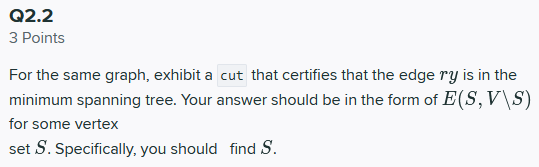




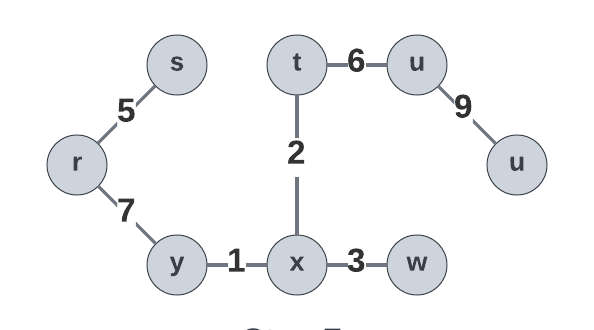
* The idea here is to apply Kruskal’s algorithm on this graph, and specify the order in which the edges are selected.
* The idea behind Kruskal is generally for minimum spanning trees (MST) of a given graph network
* The method operates by sorting edges based on weight from low to high, and the continuously connects them, but stops if a connection will create a cycle.
* We can see the order in which the edges are selected depicted below:



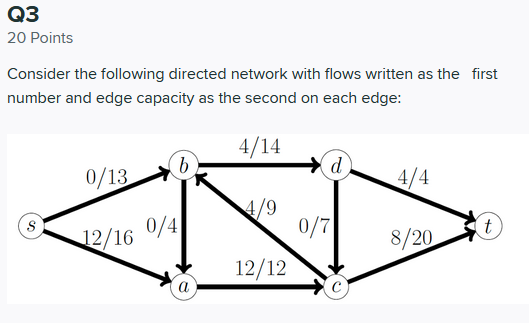
* Final Order: xy, xt, xw, rs, tu, ry, uv

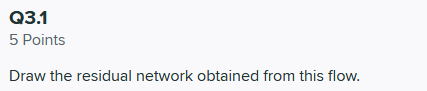


* We can define a cut as a partition of the vertices for a given graph
* In this question, our objective is to exhibit a cut that certifies that the edge ry is in the MST
* To begin, we can review the graph which we developed from Kruskal’s algorithm we prepared in the earlier question:

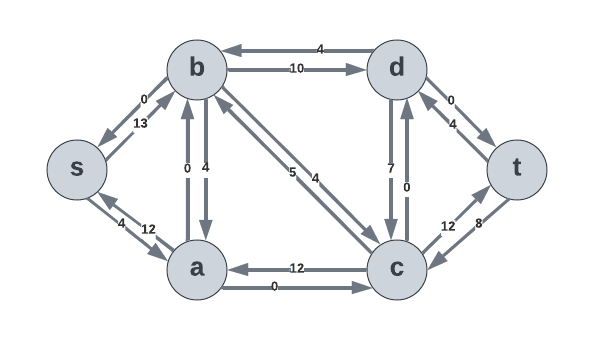


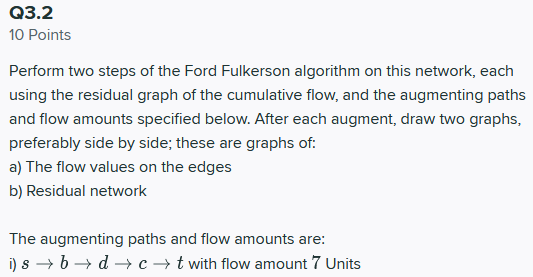
* For the cut of edge x-y,we see the minimal edge of the graph network outline above.
* If we removed this edge, we will not have a minimum spanning tree
* Using the form E(S, V\S), we arrive at E(X, Y\X)
* However, the sy edge is an edge cut that was not taken, therefore one can argue that the edge ry is the minimum spanning tree.



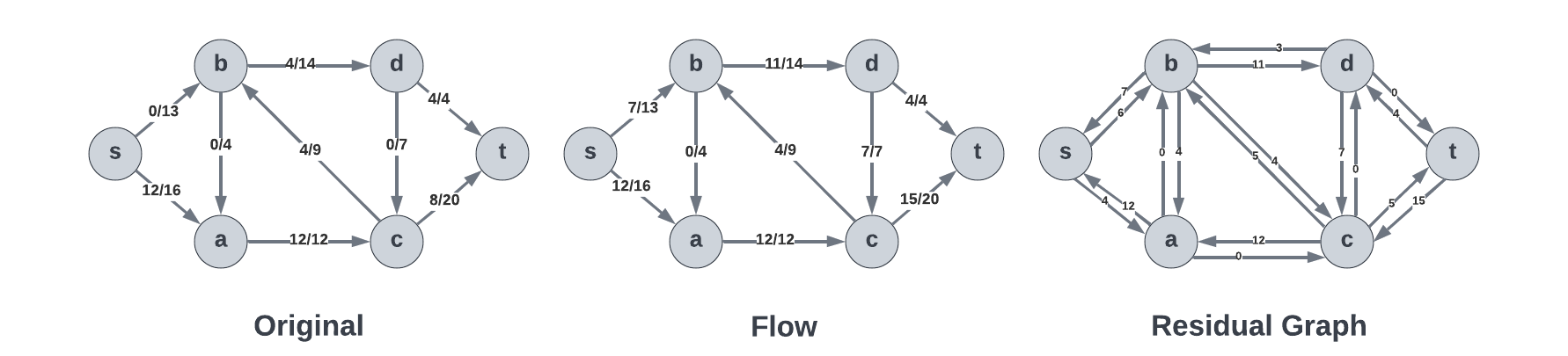


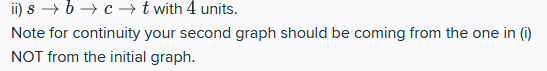
* We can define the residual graph R of a network G as a graph that has the same of vertices as the original network, but with a forward and backwards edge
* We can use residual graphs to help solve maximum flow problems for a given graph G
* For each edge e of a given graph, we note that e = (u, v) ∈ G:
  + Forward edge = (u, v) with capacity c-f
  + Backwards edge = (v, u) with capacity f
* Given that definition, we arrive at the following residual network:



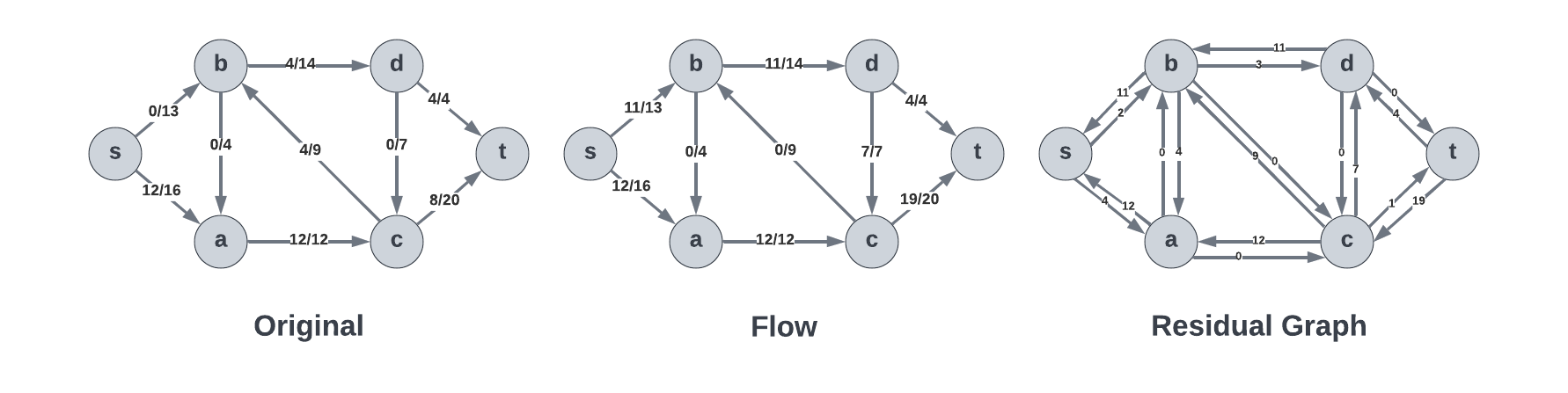


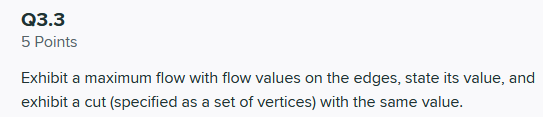
* We can recall that Ford-Fulkerson is a greedy algorithm to calculates the maximum flow in a flow network.
* The algorithm operates by:
  + Start with the flow initially at 0
  + While there exists an augmenting path s -> t
    - Add this path flow to the flow
  + Return the final flow



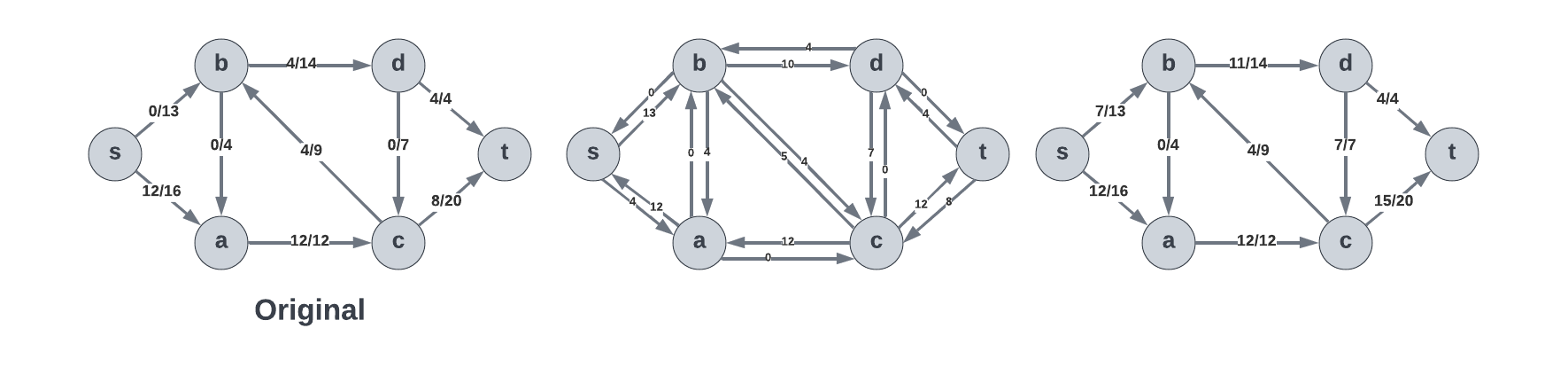


* Per the requirement above, we will continue from the graph outlined in (i)
* Continuing with the same idea and process as outlined above, we arrive with the following figure representing the original graph, the flow, and residual graph:

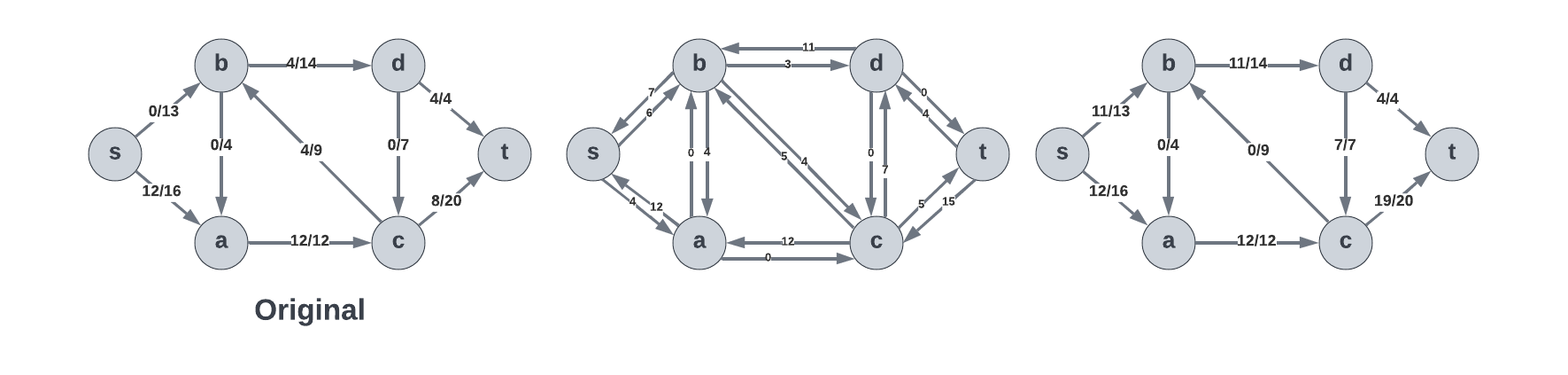




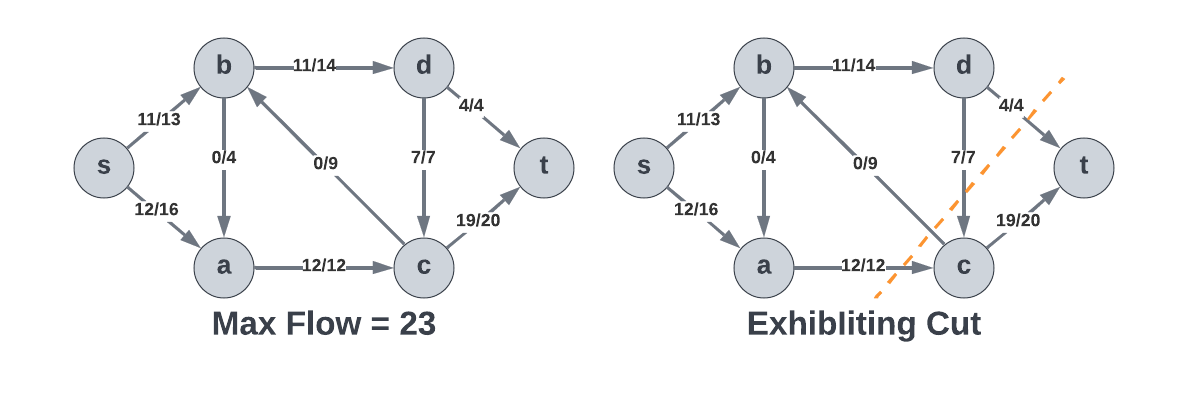
* Objective here is to show the process of obtaining the max flow, and specify the flow values on the edges, and exhibit a cut
* In order to do so, we will need to start off by drawing the residual network
* We then find the augmenting path from the RN and show flow from the path
  + We augment from s -> t
  + Process: s -> b -> d -> c -> t
  + Results in flow = 7

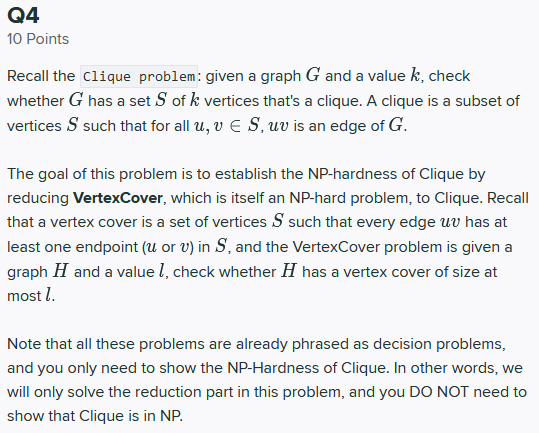


* Similarly, we can augment with a similar manner
  + We augment s -> t
  + Process: s -> b -> c -> t
  + Results in flow = 4

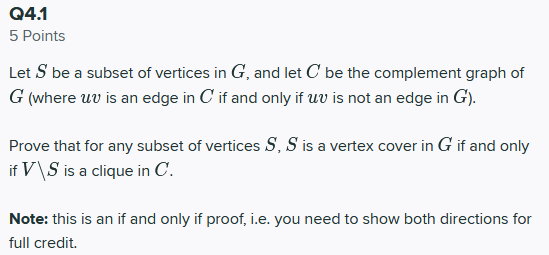


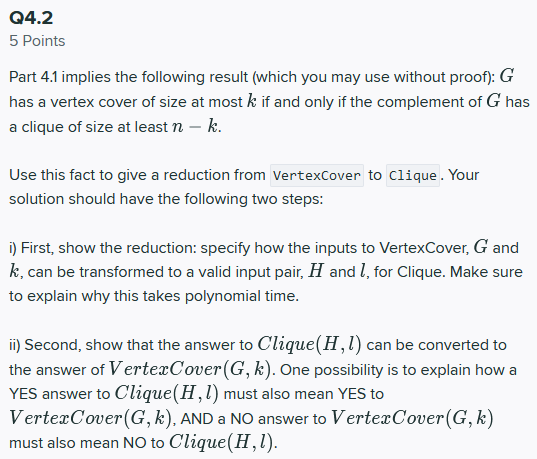
* From this, we see the inability to augment the path
* We can then show the maximum flow diagram below
* Maximum flow = 23





This question took quite a bit of time to figure out, so I will present my best answer in a way such that I explain my reasoning as I present my results for 4.1, 4.21, and 4.23 since they discuss the same idea.





* **(4.1 and 4.2)**
* We will let S be a subset of vertices in G, and we will let C be the complement graph of G
* We will assume uv is an edge in C iff uv is not an edge in G
* Our objective is to prove that for any subset of vertices S, S is a vertex cover in G iff V\S is a clique in C
* We will define a clique a as a collection of vertices in an undirected graph, so that every two vertices in a given clique are seen as nearby which implies that the subgraph is complete
* We can see that by definition that S is a vertex cover in the graph G if:
* For every edge in graph G, denoted by uv, has at least either u or v in S
* Since C is the complement graph of the graph G, we see that since edge uv is in C, uv is not an edge in G
* Therefore, every edge denoted by uv in C, there is at least u or v that is not in S
* Since S is a vertex cover in graph G, every edge denoted by uv is in-fact in C, and so at least one of the nodes u or v will be in V\S
* We look this over within the confines of a proof, in which a claim is made such that Graph G has a VertexCover of a given size k, iff (if and only if) Graph H (based on the original description) has a clique of size n-k.
  + Proof.
    - Given graph G and its size of k, we can say that if G = (V, E) has a VertexCover of a size of k, and a set S as its associated VertexCover set, then we can say that all the edges in that graph G will be incident on the given set S.
    - Given that relation, we can state that there will be no edges in the vertices in that set V\S.
    - That said, the complement graph, denoted by H, and all the vertices in V\S will be connected.
    - Since they are connected, H will have a clique of size n-k (as specified above)
    - However, if S is not a Vertex Cover for graph G, then there must be an edge uv in which neigher v or u are in S or H. Thus, the set V\S can in no way be clique because of u and v in V\S not being connected to each other.
    - Therefore, G can have a VertexCover with a given size k IFF (if and only if) its complement H has a clique of n-k.
* **Therefore, we can state that V\S is a clique in C, as described in both directions.**
* **(4.2)**
* Our objective here is to show the reduction
* We are asked to specify how the inputs to VertexCover, G and k, can be transformed to a value input pair denoted by H and l for Clique.
* Our objective here is to also explain why this takes polynomial time
* Using this, we are then asked to show how Clique(H, l) can be converted to the answer of VertexCover(G, k)
* It is suggested that a possible path is to show how a YES to Clique(H,l) is also a YES to VertexCover(G, k)
* In addition, a NO to VertexCover(G, k) is also a NO to Clique(H, l).
* We can begin with the reduction. Let us assume we are given a Graph G and some input H and I
* In order to convert this to input values for Clique the first objective is to create a graph, G’, and some integer value in which the G’ is the complement graph of the original graph G.
* We must note that the edges that exist in G and its complement G’ are reversed, as described in part 1 of this problem.
* The reduction described above can be done in Polynomial time, relative to the size of the given input value. This is of course dependent on the VertexCover(G, k).
* This is because the VertexCover(G, k) is NP-Hard. That said, Clique here will be NP-Hard as well.
* **Similar to the proof above:**
  + Since S is a vertex cover in graph G, every edge denoted by uv is in-fact in C, and so at least one of the nodes u or v will be in V\S
  + We look this over within the confines of a proof, in which a claim is made such that Graph G has a VertexCover of a given size k, iff (if and only if) Graph H (based on the original description) has a clique of size n-k.
    - Proof.
      * Given graph G and its size of k, we can say that if G = (V, E) has a VertexCover of a size of k, and a set S as its associated VertexCover set, then we can say that all the edges in that graph G will be incident on the given set S.
      * Given that relation, we can state that there will be no edges in the vertices in that set V\S.
      * That said, the complement graph, denoted by H, and all the vertices in V\S will be connected.
      * Since they are connected, H will have a clique of size n-k (as specified above)
      * However, if S is not a Vertex Cover for graph G, then there must be an edge uv in which neigher v or u are in S or H. Thus, the set V\S can in no way be clique because of u and v in V\S not being connected to each other.
      * Therefore, G can have a VertexCover with a given size k IFF (if and only if) its complement H has a clique of n-k.

**References**:

[1] CS5800 Course Modules and Notes

[2] Introduction to Algorithms, Cormen, Third Edition. (CLRS)

[3] LucidChart Drawing Tool (<https://lucid.app/lucidchart>)

[4] CS5800 Princeton Reference: https://www.cs.princeton.edu/~wayne/cs423/lectures/reductions-poly-4up.pdf