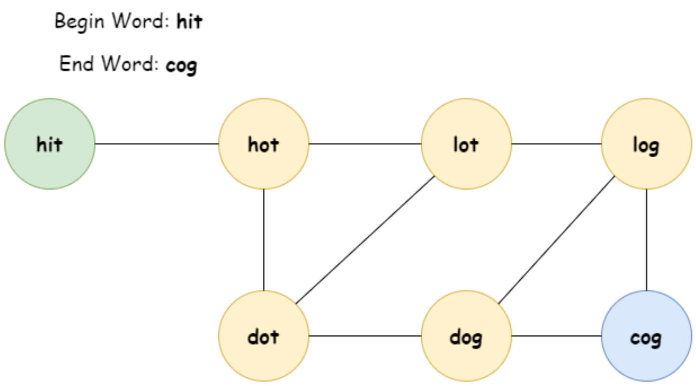
Q1.1

1. We have an edge from one word to another if the two words are only different by a single letter. Any path from one word to another is a solution to the word ladder puzzle.
2. The graph is an undirected graph and that the edges are unweighted.



Approaches:

- Represent the relationships between the words as a graph.
- Use BFS to find an efficient path from the starting word to the ending word.

Q1.2

As a starting point, we can create a vertex in the graph for every word in the list.

- To figure out how to connect the words, we compare each word in the list with every other.
- When we compare, we are looking to see how many letters are different. If the two words in question are different by only one letter, we can create an edge between them in the graph.
- Comparing one word to every other word on the list is an $O(n^2)$ algorithm.

```
def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile,'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
    return g
```

Q1.3
We use BFS to find the minimum steps of transformations needed to turn the starting word into the ending word.

- We use queue as the data structure to decide which vertex to explore next, and to maintain a record of the depth to which we have traversed at any point.
- We initialize a visited set to retain a record of which vertices have been visited already.
- We begin to systematically grow the paths one at a time, starting from the path at the front of the queue, in each case taking one more step from the vertex last explored.
- Once we have popped from our queue a path to continue exploring, and retrieved the last the vertex visited from that path, we retrieve its neighbors from our graph, remove those vertices that we know have already been visited, then for each of the remaining (unvisited) neighbors do two things:
    - Add the vertex to visited
    - Add a path consisting of the path so far plus the vertex

Q2.1
order of the edge selected: yx -> xt -> xw -> rs -> tu -> ry -> uv

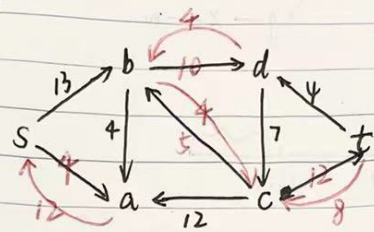
Q2.2

A cut that divided the graph G into 2 subgraphs S1: {r, s} , S2: {t, v, w, x, y, z} , Cut Vetices V: {r, y}
The cut set of G is {st, sy, ry}
According to the cut property, the minimum weighted edge from the cut set should be present in the minimum spanning tree of G.
Therefore, ry = 7 is the minimum from the cut set and ry is in the minimum spanning tree.

We have E (S1, V, S2)

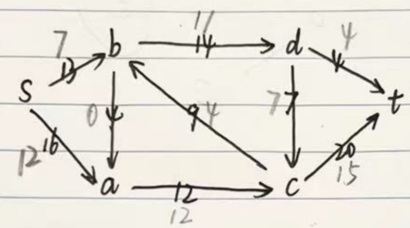
Q3

Q 3.1



Q 3.2

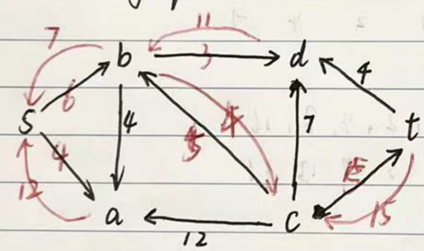
(i)



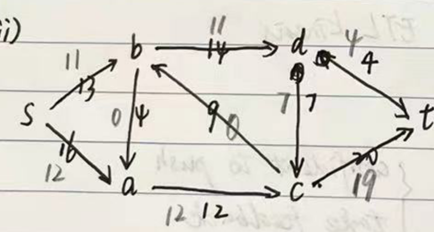
augmenting path: $s \to b \to d \to c \to t$

flow values marked in gray

residual graph:



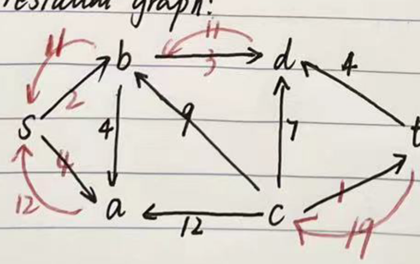
(ii)

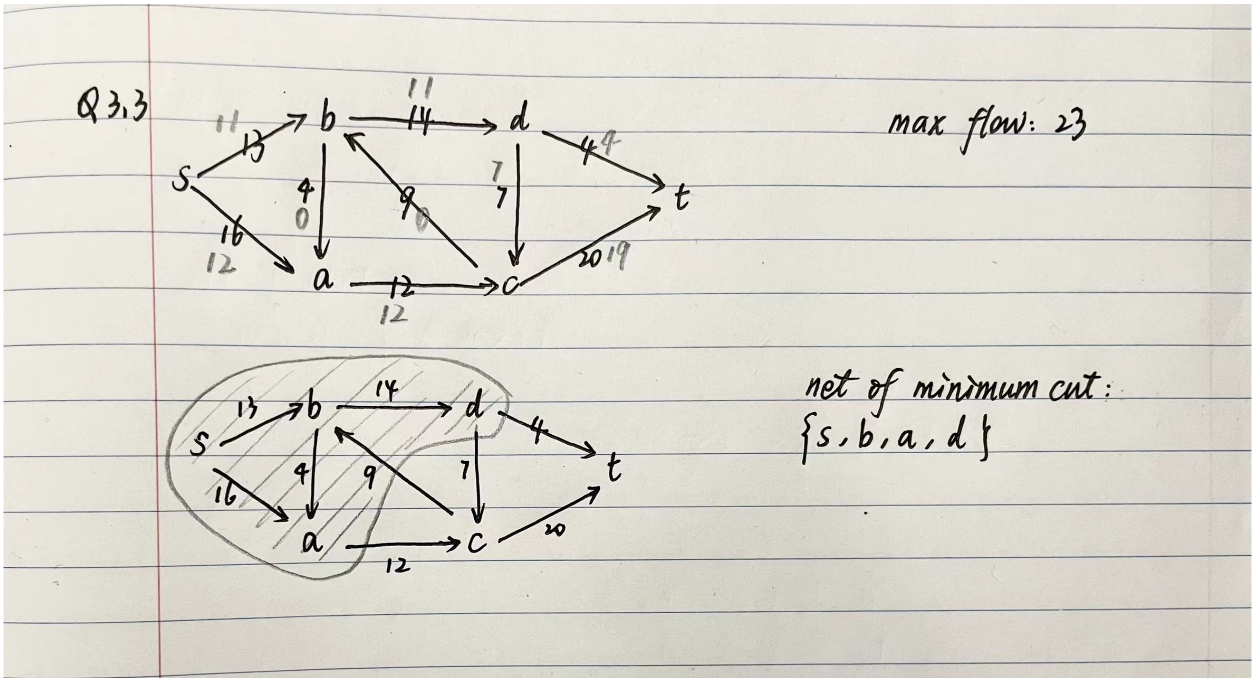


augmenting path: $s \to b \to c \to t$

flow values marked in gray on graph

residual graph:





Q4.1

- ➔ If S is a vertex cover in G

From the assumption statement, if S is a vertex cover in G, it is a set of vertices that includes at least one endpoint of every edge of the graph. Then, any edges will be covered with vertices in S. And there will be no edges amongst vertices in (V\S). Thus, all the vertices in (V\S) will be completely connected.
Since uv is an edge in C if and only if is not an edge in G, all the vertices in the (V\S) will be in C.
Therefore, (V\S) is a clique in C.

- ➔ If V\S is a clique in C

From the assumptio statement, If V\S is a clique in C, then all vertices in V\S will be completely connected. i.e for all u, v ∈ (V\S), uv is an edge of G.
Then, all any other edges must have at least one endpoint in S.
Therefore, S is a vertex cover in G.

Q4.2

(i) Show the reduction

Given a graph G and integer k as input to vertex cover problem where we have to decide if G has VertexCover of size k or not.

To convert it to input of clique, we will create graph H and integer l where H is the complement graph of graph G (edge present in G is not in H and vice-versa) and l = n where n is number of vertices in G. Thus, we have Clique (H, l)

To generate the complement graph, we only need a single scan over all pairs of vertices in the original graph, and generate an edge if there is not edge between any pair. This operation can be done in polynomial time.

(ii) Show that the answer can be converted to VertexCover(G, k)

➔ YES answer to Clique(H, l) must also mean YES to VertexCover(G,k)

If G = (V, E) has VertexCover of size k with some set S as VertexCover set. Then all edges in G will be incident on set S. And there will be no edges amongst vertices in (V\S).

This means that in the complement graph H, all the vertices in (V-S) will be completely connected, which infers that H will have clique of size (V\S) = n – k.

➔ NO answer to Clique(H, l) must also mean NO to VertexCover(G,k)

And if S in not a VertexCover in G, then there exist some edges or such that neither v nor u is in s. Hence, in graph H, set (V\S) cannot be clique because vertices v and u in (V\S) will not be connected.

Therefore, G will have VertexCover of cover size at most k if and only if H will have clique of size at least n-k.