Synthesis2 - Lun Jyun Jhu

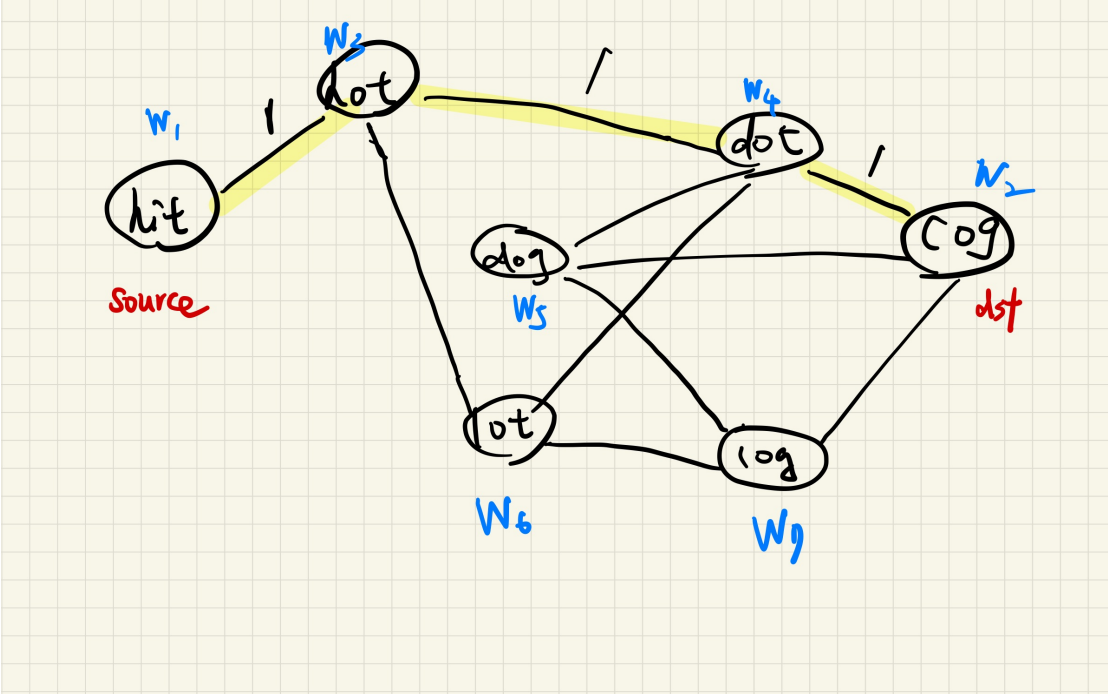
Lun Jyun Jhu

Q1

1.1

We can create a graph with one node for each word in the dictionary. We will then add an edge between two nodes if the corresponding words differ in at most *d* characters. The goal is then to find the shortest path from the node corresponding to *W*1 to the node corresponding to *Wn*. There are a few different algorithms that can be used to solve the shortest path problem. One common algorithm is Dijkstra's algorithm

Here is a graph visualization of the problem for the example given in the question:



As you can see, there are several different paths from the node corresponding to "hit" to the node corresponding to "cog" in this graph. The shortest path is the one that takes 3 steps: "hit" -> "hot" -> "dot" -> "cog".

1.2

The graph can be constructed in $O(n^2)$ time by first sorting the dictionary of words. This can be done in $O(n \log n)$ time. Once the words are sorted, we can then iterate through the words and add an edge between any two words that differ in at most d characters. This can be done in $O(n)$ time per word, for a total of $O(n^2)$ time.

The size of the graph is up to $O(n^2)$ because there are up to n^2 possible pairs of words that differ in at most d characters. For each such pair of words, we will add an edge between them. Therefore, the total number of edges in the graph is at most $O(n^2)$.

1.3

1. Create a graph with n nodes, one for each word in the dictionary.
2. For each pair of nodes i and j, check if the corresponding words Wi and Wj can be transformed into each other according to the given condition. This step takes O(n^2 * k) time.
3. Create an adjacency matrix A for the graph, where A(i, j) = 1 if there is an edge between nodes i and j, and 0 otherwise. This takes O(n^2) time.
4. Compute the shortest path matrix D using Floyd-Warshall algorithm. The Floyd-Warshall algorithm takes O(n^3) time.
5. The length of the shortest path between node 1 (representing the first word in the dictionary) and node n (representing the last word in the dictionary) is D(1, n).
6. Return the length of the shortest path found.

Q2

2.1

E(y, x) = 1

E(x, t) = 2

E(x, w)= 3

E(r, s)= 5

E(t, u) = 6

E(r, y) = 7

E(u, v) = 9

MST = 33

2.2

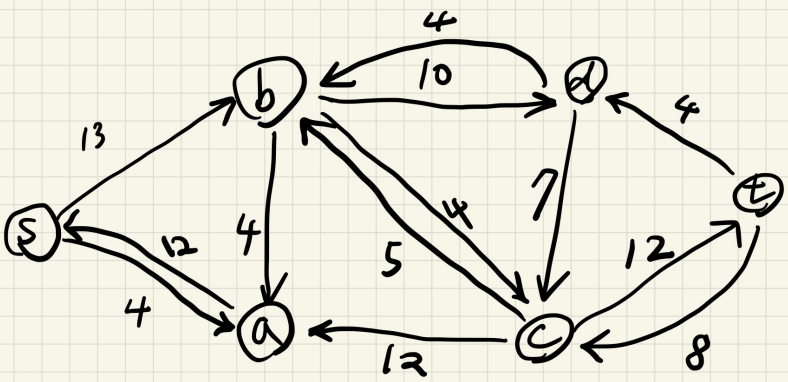
To exhibit a cut that certifies the edge E(r, y) is in the minimum cost spanning tree, we need to find a cut (S, V\S) such that E(r, y) is the minimum cost edge crossing the cut.

1. Start with the set S = {r}.
2. Add to S any vertex that can be reached from r by following edges that are strictly smaller in weight than E(r, y). In this case, the only such vertex is s, so we add s to S.
3. The cut (S, V\S) consists of the vertices {r, s} on one side and the remaining vertices {t, u, v, w, x, y} on the other side.
4. The edge E(r, y) is the minimum cost edge crossing the cut (S, V\S), with a weight of 7. All other edges crossing the cut have higher weights.

Therefore, the cut (S, V\S) certifies that E(r, y) is in the minimum cost spanning tree. The vertex set S is {r, s}.

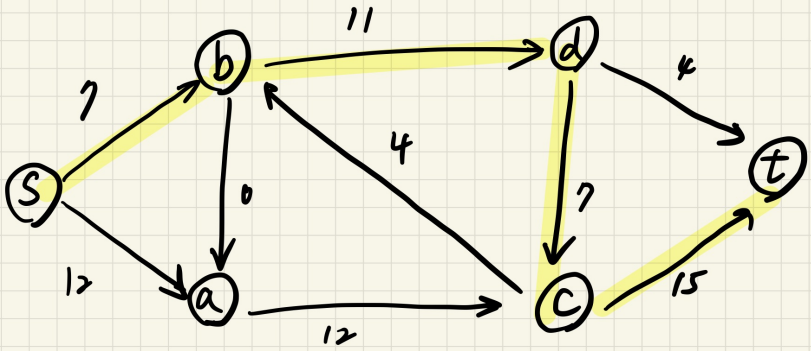
Q3

3.1

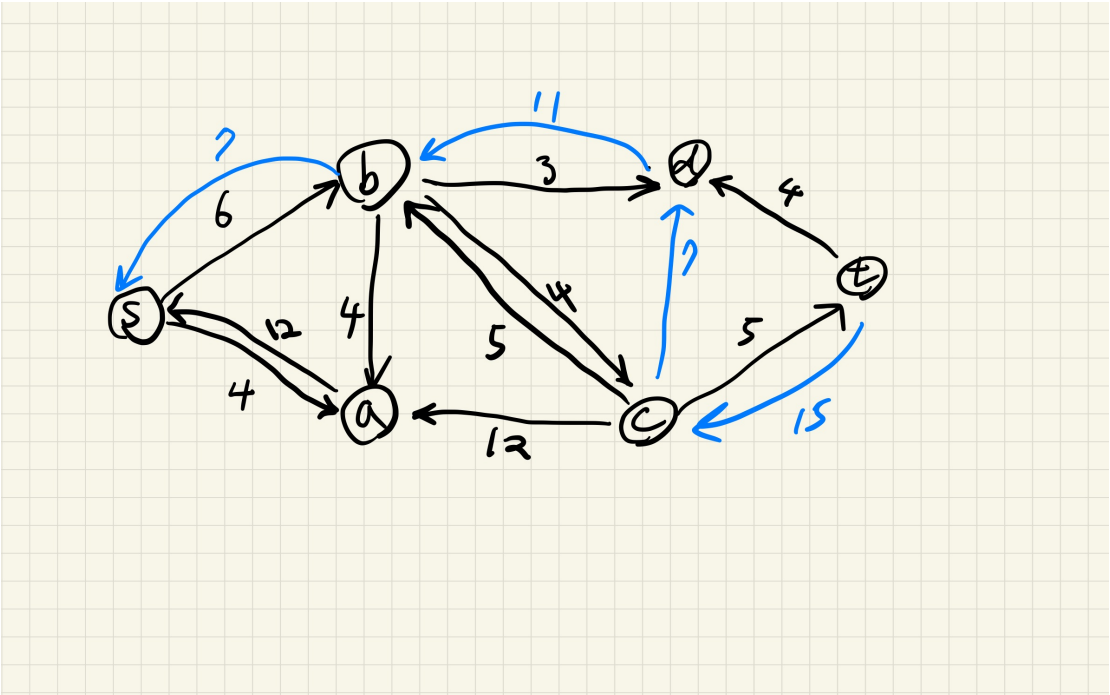


s → b → d → c → t

flow graph



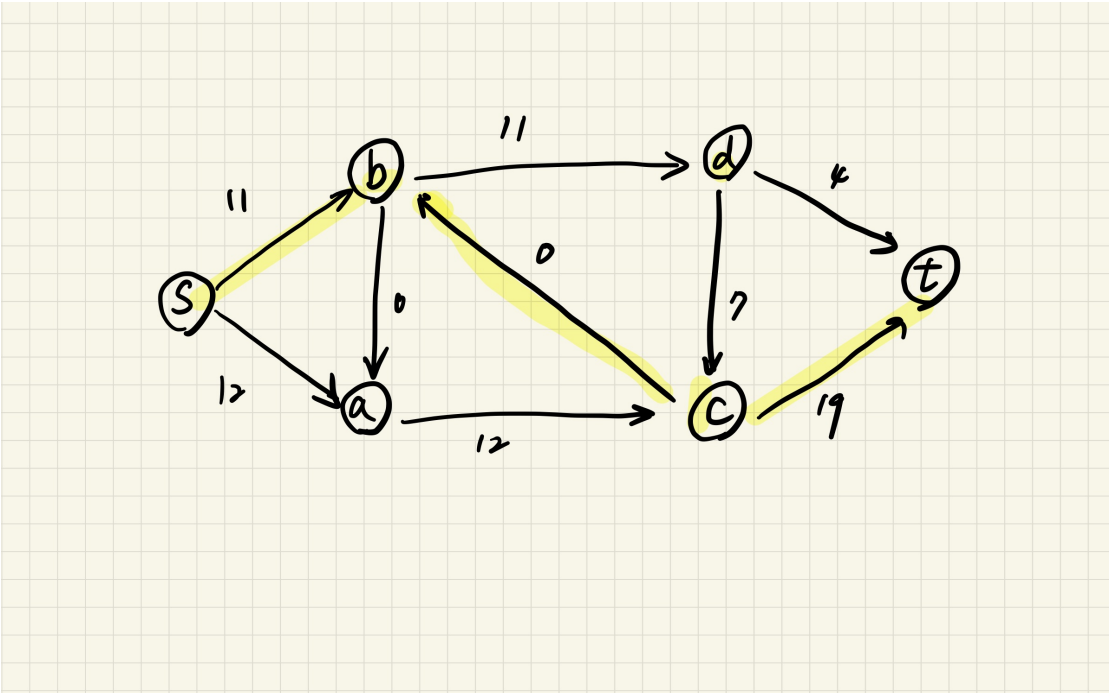
residual graph



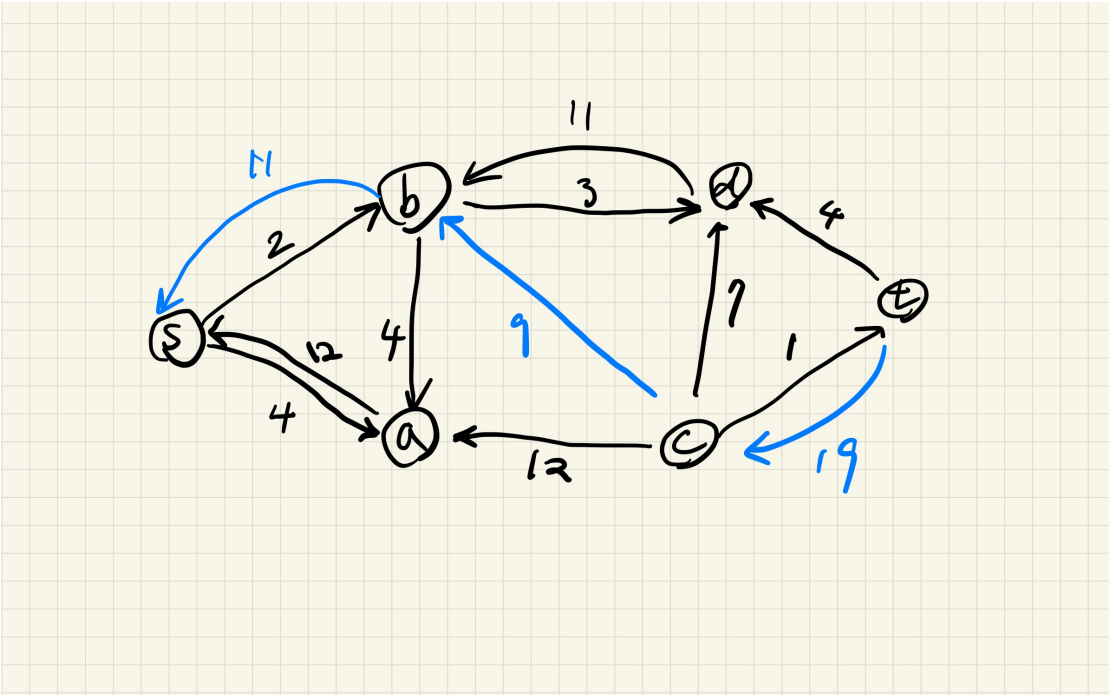
Q3.2

s → b → c → t

flow graph

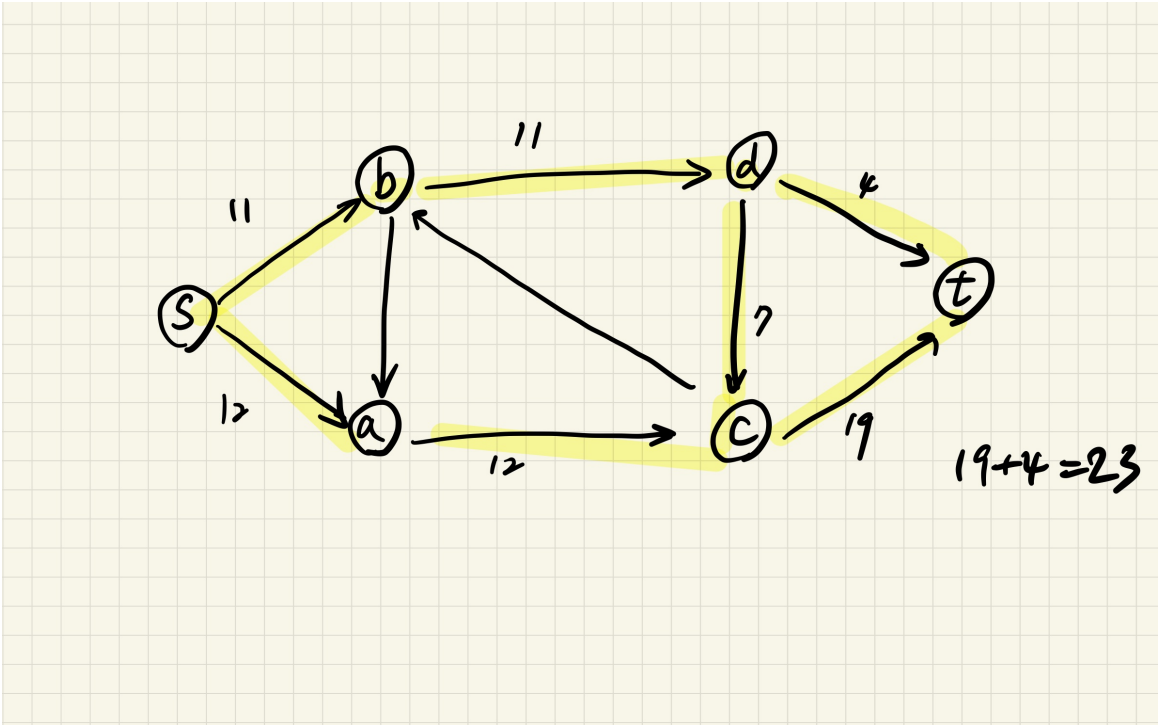


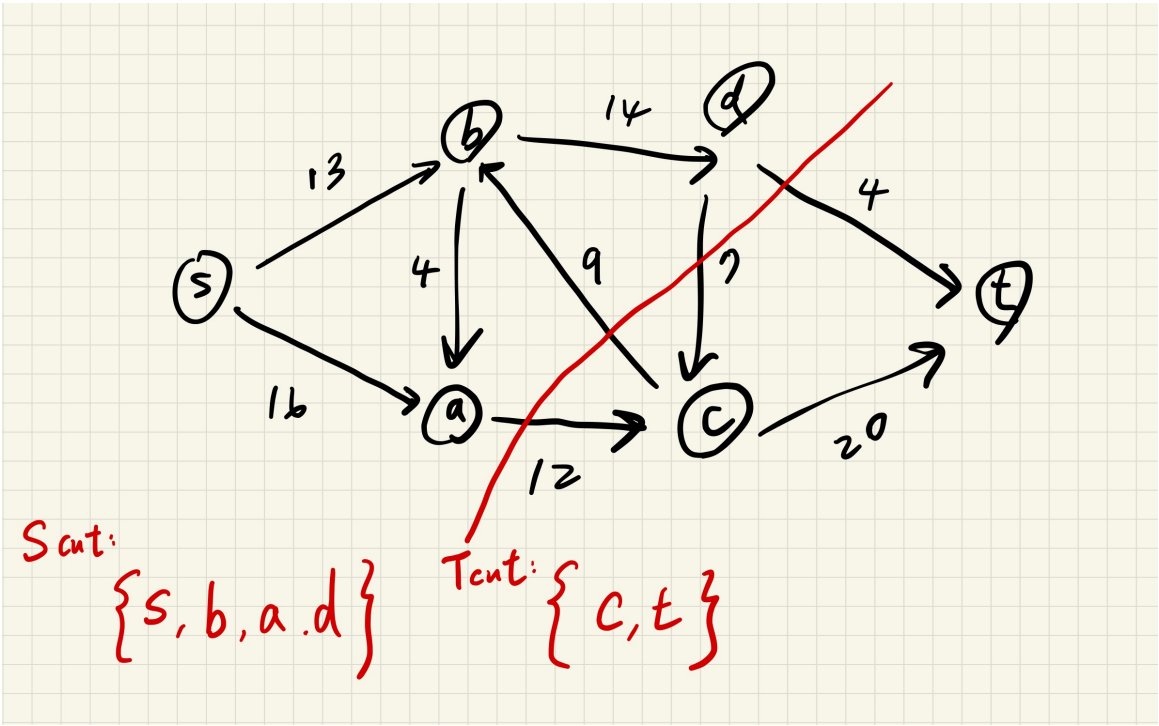
residual graph



3.3

max flow = 23


s
b
a
d
c
t
11
11
12
12
7
4
19
19+4=23


s
b
a
d
c
t
13
16
4
14
9
7
12
4
20
S cut:
{s, b, a, d}
T cut:
{c, t}

Q4.1

First, let's assume that S is a vertex cover in G. This means that for every edge uv in G, at least one endpoint (u or v) is in S. We want to show that V\S is a clique in C. Let's take any two vertices x and y in V\S. Since x and y are not in S, they are not adjacent to any vertex in S. Therefore, the edges xy are not in G, which means that they are in C. Thus, x and y are adjacent in C, and V\S is a clique in C.

Next, let's assume that V\S is a clique in C. We want to show that S is a vertex cover in G. Let's take any edge uv in G. We know that uv is not in C, which means that u and v are adjacent in G. If neither u nor v is in S, then both u and v are in V\S, which means that they are adjacent in C. But we assumed that V\S is a clique in C, which means that uv is an edge in C. This is a contradiction, which means that at least one of u and v must be in S. Therefore, S is a vertex cover in G.

In conclusion, we have shown that for any subset of vertices S, S is a vertex cover in G if and only if V\S is a clique in C. This implies that the Clique problem is NP-hard, since we can reduce VertexCover to Clique by constructing the complement graph C and setting k to be the complement of the size of the vertex cover in G.

Q4-2

we can use the fact given in the previous question that a graph 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. Here are the steps of the reduction:

Step 1: Given a graph G and a value k, we create a new graph H and a value l as follows:

- The vertices of H are the same as the vertices of G.
- For every pair of non-adjacent vertices in G, we add an edge between them in H.
- Set l to be n - k, where n is the number of vertices in G.

This reduction takes polynomial time because we only need to iterate over all pairs of non-adjacent vertices in G, which takes O(n^2) time.

Step 2: Now we need to show that the answer to Clique(H, I) can be converted to the answer of VertexCover(G, k).

- If Clique(H, I) returns YES, then there exists a subset S of at least n-k vertices in H such that every pair of vertices in S is adjacent. But since every non-adjacent pair of vertices in G was added as an edge in H, this means that there are no edges between the vertices not in S in H, and therefore all of those vertices are in a vertex cover of size at most k in G. Thus, VertexCover(G, k) returns YES.
- If Clique(H, I) returns NO, then there does not exist a subset S of at least n-k vertices in H such that every pair of vertices in S is adjacent. This means that there is no clique of size at least n-k in the complement of G, which in turn means that there is no vertex cover of size at most k in G. Thus, VertexCover(G, k) returns NO.

Therefore, we have successfully reduced VertexCover to Clique, and since VertexCover is NP-hard, this implies that Clique is also NP-hard.