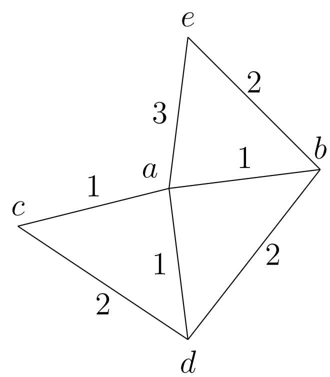
Name: Yan, Zi
Course: CIS 502
Assignment: HW2

Problem 1

a.) No. In the figure 1, a minimum-bottleneck tree can be {(c,d), (d,b), (b,e), (a,b)}, but the minimum spanning tree is {(a,b), (a,c), (a,d), (b,e)}.

Figure 1: example



b.) Yes. Suppose not. Let $T = (V, E')$ be the minimum spanning tree of the graph G, and $T' = (V, E'')$ be the minimum-bottleneck tree of G, where T and T' are not identical. The difference is that $e \neq e'$ and $w_e < w_{e'}$, where $e \in E''$ is the most weighted edge in T', and $e' \in E'$ is the most weighted edge in T, but the rest edges of two spanning tree are the same. From the assumption, we can show that $\sum\limits_{e \in E'} w_e > \sum\limits_{e \in E''} w_e$. But this contradicts the fact that T is MST.

c.) We can simply use Kruskal's Algorithm to find a minimum spanning tree in $O(m \log n)$ time. And the MST can be regarded as a minimum-bottleneck tree.

d.) We can combine binary search and DFS to implement the algorithm. Briefly, we use binary search to find a minimum bottleneck weight, using DFS to verify whether all the edges weighted no greater than this weight can make the original graph connected, namely forming a spanning tree.

Proof. Because DFS will provide a subgraph which makes all the nodes connected, if the original graph is connected. And once the binary search can give a w_{max} that makes the graph with all edge weights less than or equal to w_{max} connected, the DFS-MOD will also give a connected subgraph which can be regarded as a spanning tree. As the binary search goes on, a minimum

w_{max} can be found that it is the minimum weight and the graph with all edge weights no greater than it will still be connected. At this time, DFS-MOD will provide a minimum-bottleneck tree.

The binary search will take $O(\log m)$ time, but DFS-MOD will only visit half of the graph at first and half of the unvisited graph or the visited half in the subsequent steps. Therefore, the total runtime will be $\sum\limits_{i=1} O(\frac{m+n}{2^i}) = O(m+n) = O(n + m \log n)$ □

Algorithm 1 Using DFS to find minimum-bottleneck tree

FIND_MBT(G)
while there are still more than one vertex in G **do**
  Let w_{max} = the median number of all existing edge weights
  VERIFY-WEIGHT(G, w_{max})
  if w_{max} is a valid weight **then**
    Remove all the unvisited edges from G
  else
    Regard all visited edges and vertices as a single node in the following steps
  end if
end while
The vertices and remaining edges consist of minimum bottleneck tree

---

VERIFY-WEIGHT(G, w_{max})
for each vertex $u \in V$ **do**
  visited$[u]$ = **false**
end for
v = any vertex picked from V
DFS-MOD(v, w_{max})
for each vertex $u \in V$ **do**
  if visited$[u]$ == **false then**
    return "not a valid weight"
  else
    return "a valid weight"
  end if
end for

Problem 2

(a) It is true. Suppose not. Assume MACS (minimum altitude connected subgraph) has a distinct edge from MST (minimum spanning tree), connecting two nodes i and j to form a *winter-optimal* path. This means that a edge e_{MACS} in MACS, which is the highest edge in the path from i to j, is lower than the highest edge e_{MST} in MST. Therefore, $\sum\limits_{e \in E_{\text{MACS}}} a_e < \sum\limits_{e \in E_{\text{MST}}} a_e$, which contradicts the fact of MST.

```
DFS-MOD(u, w_max)
visited[u] = true
for each v ∈Adj[u] do
  if not visited[v] and w(u, v) ≤ w_max then
    DFS-MOD(v, w_max)
  end if
end for
```

(b) It is true. Suppose not. Assume the MACS contains no edge from the MST. This means in a cycle the highest edge e_h, which connects the vertices i and j, is in the MACS. Therefore removing e_h and connecting i and j with the "longer way" will form a better *winter-optimal* path than before. This contradicts the assumption.

Problem 3

Initially we need an additional array r, and r is identical to array d, where $r_i = d_i$, and it means that v_i can still form r_i edges to other vertices.

The algorithm is that from v_1 to v_n you pick a vertex one by one. Every time a vertex v_i is chosen, you choose any r_i other vertices, each of which has its subscript j larger than i with non-zero r_j value, to form r_i edges for each, then decrease the corresponding r_j by 1. After you go through all the vertices, all the element in array r should be zero, otherwise the graph G will contain either multiple edges between the same pair of nodes or self-loop edges, or both.

The algorithm maintains that if G will not contain either multiple edges between the same pair of nodes or self-loop edges, at vertex v_i, each vertex v_j, where $j < i$, have connected to d_j vertices with $r_j = 0$, and $r_i \leq (n - i)$.

Proof. **Base:** When picking v_1, it is trivial.

Induction Step: Assume at vertex v_i, every vertex v_j, where $j < i$, have connected to d_j vertices with $r_j = 0$, and $r_i \leq (n - i)$. So at vertex v_{i+1}, according to the operation in ith step, there should be no less than r_i vertices with non-zero r, otherwise v_i will have not enough vertices to connect to, which leads to either self-loop edges or multiple edges between v_i and any other vertices, therefore contradicts the assumption. Then, v_i can form r_i edges from itself to any r_i vertices, decreasing r_i to zero. For r_{i+1}, it should be no greater than $n - i$, otherwise, multiple edges between v_{i+1} and any other vertices or self-loop edges will be formed, which contradicts the assumption. Consequently, at v_{i+1}, the property still maintains.

At v_n, all the other vertices have their r equal to zero, and $r_i \leq (n - n) = 0$. It means every vertex v_i connects to d_i different vertices with no multiple edge between the same pair of vertices and no self-loop edge. □

For each vertex v_i, $O(d_i)$ is used to form edges and decrease r_i, so the total runtime of the algorithm should be $\sum_{k=1}^{n} O(d_k) + O(n) = O(n + m)$.

Problem 4

The algorithm maintains a property that every time a vertex v is visited, dist[v] will hold the current shortest distance from s to v, and num[v] indicates the number of shortest paths from s to v with distance dist[v].

Proof. **Base:** At the beginning, only s is visited, therefore, dist[s] =0, num[s]=1, and the dists of all the other nodes are ∞ and the nums of them are 0.

Induction Step: At ith step, all the visited nodes will hold shortest distance from s and how many of them. Then, at $(i+1)$th step, a vertex v will be visited from a vertex u, which is visited at i step, on the following two conditions:

- v is not visited. At this time, at least one shortest path from s to v is established. But the actual number of shortest paths is just the number of shortest paths from s to u.
- v is visited before. So if another shortest path from s via u to v exists, the total number of shortest paths should be accumulated. And Line 15 finishes the job.

Finally, at vertex t, the shortest distance from s and the number of the of paths will maintain. □

Because the algorithm just add some constant time operations, Line 12-16, the runtime should be still $O(n+m)$.

```
1: BFS-SHORTEST(s)
2: Set visited[s] = true and visited[v] = false for all other v
3: Set dist[s] = 0 and dist[v] =∞ for all other v
4: Set num[s] = 1 and num[v] = 0 for all other v
5: Add s to Queue Q
6: while Q is not empty do
7:    Let u = Dequeue(Q)
8:    for all v which is adjacent to u do
9:       if visited[v] == false then
10:         Enqueue(Q, v)
11:         visited[v] = true
12:         dist[v] = dist[u] + 1
13:         num[v] = num[u];
14:      else
15:         if dist[v] == dist[u] + 1 then
16:            num[v] = num[v] + num[u]
17:         end if
18:      end if
19:   end for
20: end while
21: return num[t]
```