Answer to Problem Set 9

- Student Name: Gaole Dai
- Student ID: S01435587

Problem 1

Assume the relation between the sizes of E and V is that $E = \mathcal{O}(V^2)$

Thus, it could be deduced that

$\log(|E|) = \log(\mathcal{O}(|V|^2)) = \log(|V|^2) = \mathcal{O}(2\log(|V|)) = \mathcal{O}(\log(|V|))$

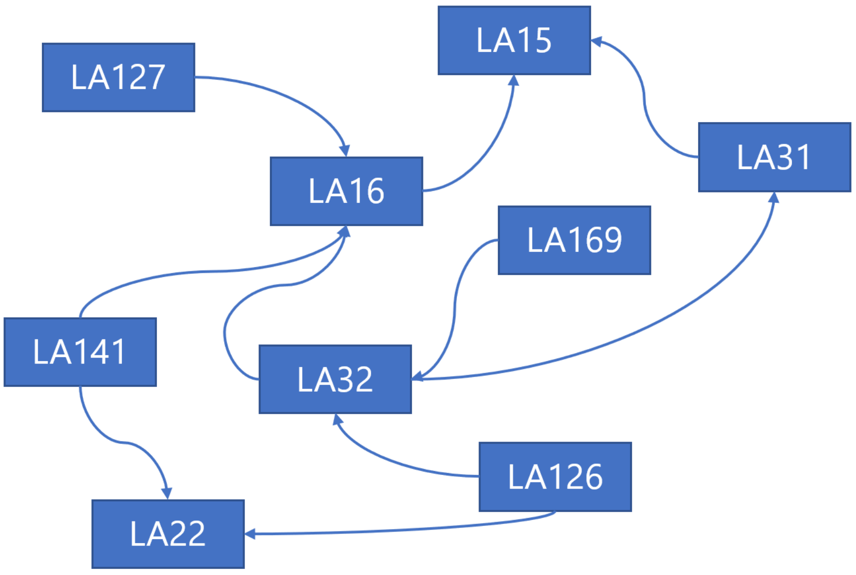
Assume another relation between the sizes of E and V is that $|E| = |V|^2$

Then, $\log(|E|) = \log(|V|^2) = 2\log(|V|)$.

Therefore, $\mathcal{O}(\log(E)) = \mathcal{O}(\log(V))$, however, $\mathcal{O}(E) = \mathcal{O}(V^2) > \mathcal{O}(V)$

Thus, $\mathcal{O}(\log(E)) = \mathcal{O}(\log(V))$, but $\mathcal{O}(E)$ is not necessarily $\mathcal{O}(V)$

Problem 2



Apply the adjacency matrix

	LA15	LA16	LA22	LA31	LA32	LA126	LA127	LA141	LA169
LA15	0	0	0	0	0	0	0	0	0
LA16	**1**	0	0	0	0	0	0	0	0
LA22	0	0	0	0	0	0	0	0	0
LA31	**1**	0	0	0	0	0	0	0	0
LA32	0	**1**	0	**1**	0	0	0	0	0
LA126	0	0	**1**	0	**1**	0	0	0	0
LA127	0	**1**	0	0	0	0	0	0	0

	LA15	LA16	LA22	LA31	LA32	LA126	LA127	LA141	LA169
LA141	0	**1**	**1**	0	0	0	0	0	0
LA169	0	0	0	0	**1**	0	0	0	0

The course plan that meets the prerequisites is **LA15, LA16, LA31, LA32, LA127, LA169, LA22, LA126, LA141**.

Problem 3

The order that DFS visits the node is **0, 1, 2, 3, 5, 4, 6, 7**.

```
def DFSvisit(G, v):
    if marked(v):
        return
    mark(v)
    for w in G.adj(v):
        DFSvisit(G, w)
```

Cur Visit Node	Adjacent	Visited Node
0	1, 2, 3	/
1	0, 2, 3	0
2	0, 1, 3	0, 1
3	0, 1, 2, 5	0, 1, 2
5	3, 4, 6	0, 1, 2, 3
4	5, 6, 7	0, 1, 2, 3, 5
6	4, 5, 7	0, 1, 2, 3, 5, 4
7	4, 6	0, 1, 2, 3, 5, 4, 6
No unmarked node		0, 1, 2, 3, 5, 4, 6, 7

Problem 4

The order that BFS visits the node is **0, 1, 2, 3, 5, 4, 6, 7**.

```
def BFS(G):
    Initialize WQ to empty queue
    Set Markd[v] = false for all vertices
    Picked some s as first vertex in V & mark it
    Insert s into WQ
    While (not empty(WQ)):
        v = first(WQ)
        visit(v)
        for (w in adj(v)):
            if (not marked(w)):
                marked(w)
                push(WQ, w)
```

Cur Visit Node	Adjacent	Visited Node
0	1, 2, 3	/
1	0, 2, 3	0
2	0, 1, 3	0, 1
3	0, 1, 2, 5	0, 1, 2
5	3, 4, 6	0, 1, 2, 3
4	5, 6, 7	0, 1, 2, 3, 5
6	4, 5, 7	0, 1, 2, 3, 5, 4
7	4, 6	0, 1, 2, 3, 5, 4, 6
No unmarked node		0, 1, 2, 3, 5, 4, 6, 7

Problem 5

The python code is shown below:

```
Class Ipq:
    # prio: Number
    # ipq: List[Number]
    self.ipq = []

    def empty():
    def ins(v, prio):
    def showin():
    def popmin():
    def upd(v, prio):
        elem = self.ipq[prio]
        self.ipq[prio] = v
        # if the new value is smaller than the original value, move up
        if elem > v:
            self.swim(ipq, prio)
        # if the new value is larger than the original value, move down
        elif elem < v:
            self.sink(ipq, prio)
    def sink(heap, idx):
        while vc := valid_chd(heap, idx):
            cval, cidx = min((heap[i], i) for i in vc)
            val = heap[idx]
            if val <= cval:
                break
            heap[idx], heap[cidx], idx = cval, val, cidx
        return heap

    def swim(heap, idx):
        while (pi := _p(idx)) >= 0:
            val, pval = heap[idx], heap[pi]
            if pval <= val:
                break
            heap[idx], heap[pi], idx = pval, val, pi
        return heap

    def _p(idx):
        return (idx-1)//2

    def _l(idx):
        return 2*idx+1

    def _r(idx):
        return 2*idx+2

    def valid_chd(arr, idx):
        if (lc := _l(idx)) >= len(arr):
            return []
        return [v for v in lc(lc, lc+1) if v < len(arr)]

    def hp_min(heap):
        return heap[0] if heap else None

    def hp_ins(heap, val):
        heap.append(val)
        return swim(heap, len(heap)-1)

    def hp_delmin(heap):
        heap[0], heap[-1] = heap[-1], heap[0]
        del heap[-1]
        return sink(heap, 0)
```

Problem 6

The modified Dijkstra's Algorithm for shortest path is shown below:

```
# named tuples
Graph =  nt('Graph','vert adj')
Edge = nt('Edge','vert weight')
# graph.adj = List(Edge)
def dijkstra(G,s):
    ipq = Ipq()
    # init dist for storing the distance from s to v, the length is the number of
vertexes
    dist = []
    for v in G.vert:
        ipq.ins(v,float('inf'))
        dist[v] = 0
    dist[s] = 0
    ipq.upd(s,0)
    while not ipq.empty():
        v = ipq.showmin()
        d = ipq[v]
        _ = ipq.popmin()
        for w, wt in G.adj[v]:
            d1 = d + wt
            if d1 < ipq[w]:
                # update the current shorstest value to dist
                dis[w] = d1
                ipq.upd(w,d1)
```

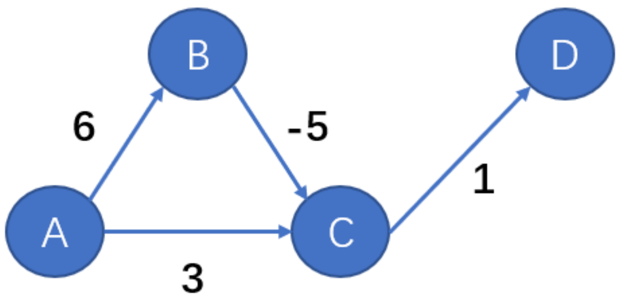
The last element in `dist` is the shortest path from `s` to `v`.

Problem 7

The counterexample is shown in the figure below.

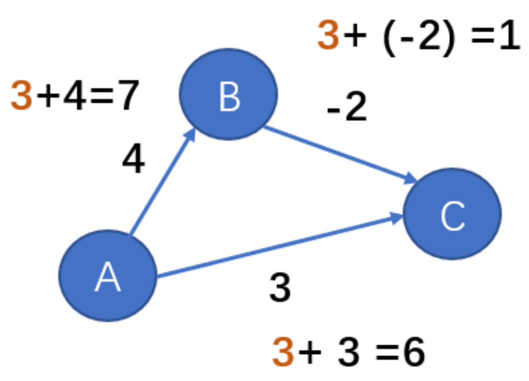
Node `A` is pushed into the priority queue first, and the adjacent nodes `B` and `C` will be pushed with value 6 and 3, and node `C` will be marked as visited and popped because it has min value, node `B` is popped, but since its adjacent node `C` is visited and marked, it did nothing. Then node `D` is pushed and updated with value 4, and popped.

The final wrong result is `A->C->D` rather then `A->B->D`.



Problem 8

A offset value **3** is added to all edge weights, similar to Problem 7, the shortest path is calculated as `A->C` (`C` has shorter value compared to `B` and then be popped, `B` has no unmarked adjacent), however, the actual shortest path is `A->B->C`, which shows that adding a positive offset will not succeed in computing shortest paths.



Problem 9

A universal sink has **no emanating edge**, and have the pattern that its row in adjacency matrix is all 0 and the corresponding column is all 1. The algorithm eliminate vertexes that does not fit the pattern. If no vertex is sink, return "No sink found", else return the index of the sink.

The time complexity is $\mathcal{O}(|V|)$, since it did not iterate the whole matrix and each iterate only one element will be checked to eliminate the vertex according to the element in row or column.

```
# Reference: Code based on https://www.geeksforgeeks.org/determine-whether-
universal-sink-exists-directed-graph/
class Graph:

    def __init__(self, vertices):
        self.vertices = vertices
        self.adjacency_matrix = [[0 for i in range(vertices)]
                                   for j in range(vertices)]

    def insert(self, s, destination):

        # Assign 1 to adjacency_matrix[i][j] if there is an edge from i to j
        self.adjacency_matrix[s - 1][destination - 1] = 1

    def issink(self, i):
        for j in range(self.vertices):

            # determine if any element in the row i is 1, if has, return false
            if self.adjacency_matrix[i][j] == 1:
                return False

            # determine if any element other than i in the column i is 0, if
has, return false
            if self.adjacency_matrix[j][i] == 0 and j != i:
                return False

        # otherwise, return true
        return True

    # eliminate n-1 non sink vertices
    def eliminate(self):
        i = 0
        j = 0
        while i < self.vertices and j < self.vertices:

            # increment the row if index is 1, else increment the column
            if self.adjacency_matrix[i][j] == 1:
                i += 1
            else:
                j += 1

        # If i exceeds the number of vertices, no sink
        if i > self.vertices:
            return -1
        elif self.issink(i) is False:
            return -1
        else:
            return i
```

```
if __name__ == "__main__":

    number_of_vertices = 6
    number_of_edges = 5
    graph = Graph(number_of_vertices)

    graph.insert(1, 6)
    graph.insert(2, 3)
    graph.insert(2, 4)
    graph.insert(4, 3)
    graph.insert(5, 3)

    vertex = g.eliminate()

    if vertex >= 0:
        print("Sink found at vertex %d" % (vertex + 1))
    else:
        print("No Sink found")
```

Problem 10

The kruskal algorithm is shown below:

```
def kruskal(G):
    workl = sorted(G.edges(),
            key=lambda e:e.wt)
    chk_cy = UF(len(G.vert))
    spantree,edg_idx = [],0
    while len(spantree) < len(G.vert)-1:
        u,v,wt = workl[edg_idx]
        if chk_cy.find(u) != chk_cy.find(v):
            spantree.append(workl[edg_idx])
            chk_cy.union(u,v)
          edg_idx += 1
    return spantree
```

The execution trace is shown below:

The edges are sorted and being popped and detect if exist the cycle to form the MST until reach the number of $|V| - 1$.

```
Number of edges |V| is 7.
Step 1: B -- E == 4, no cycle is formed, include
Step 2: A -- D == 5, no cycle is formed, include
Step 3: D -- F == 6, no cycle is formed, include
Step 4: A -- C == 7, no cycle is formed, include
Step 5: B -- C == 8, no cycle is formed, include
Step 6: C -- D == 9, result in cycle, discard
Step 7: C -- E == 10, result in cycle, discard
Step 8: G -- F == 11, no cycle is formed, include
The number of edges included in the MST equals to (|V| - 1), stop the algorithm.
Thus, the Minimum Spanning Tree Weight = 41
```