Module 8 Problem Set

Jiapeng Pei S01435611

1.

The number of edges could be very large in a dense graph. So in worst case, $E = O(V^2)$. We have:

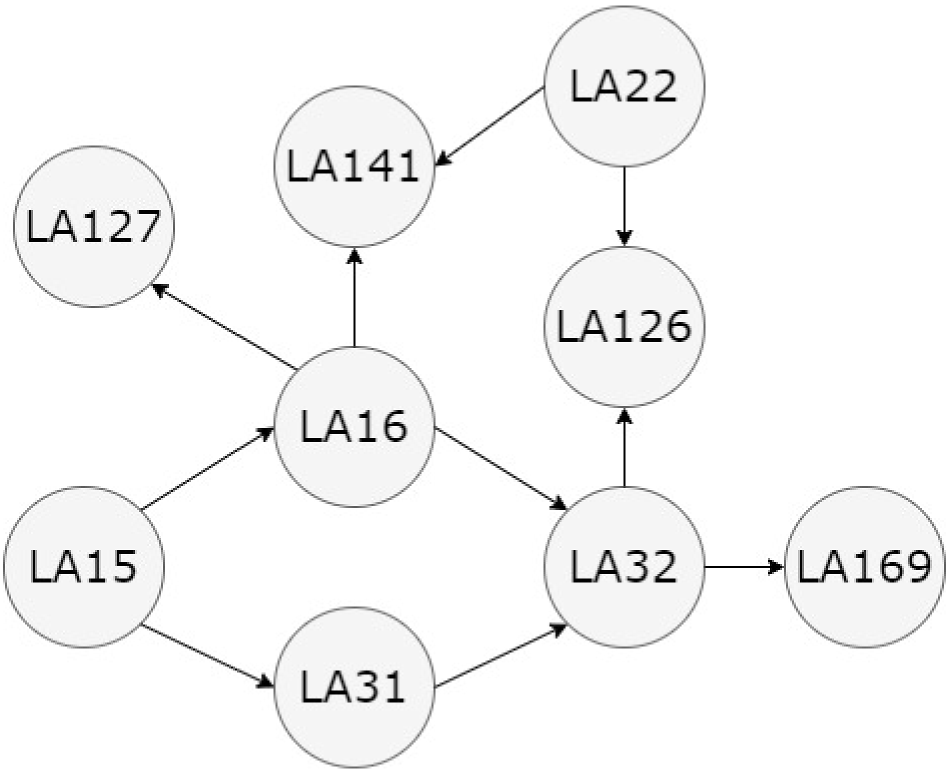
$$O(\log(E)) = O(2\log(V)) = O(\log(V))$$

But still,

$$O(E) = O(V^2) \neq O(V)$$

2. LA15, LA31, LA16, LA127, LA32, LA169, LA22, LA126, LA141

Following is the graph representation of the courses. We could find a valid course plan with topological sorting using DFS. By the way, there exist other valid course plans.



3. 1, 2, 3, 4, 6, 5, 7, 8

Following is the visit sequence:

begin at 1 -> visit 2 -> pass 1 and visit 3 -> pass 1, 2 and visit 4 -> pass 1, 2, 3 and visit 6 -> pass 4 and visit 5 -> pass 6 and visit 7 -> pass 5 and visit 8.

4. 1, 2, 3, 4, 6, 5, 7, 8

- deque = [2, 3, 4], sequence = [1]
- deque = [3, 4], sequence = [1, 2]
- deque = [4], sequence = [1, 2, 3]
- deque = [6], sequence = [1, 2, 3, 4]
- deque = [5, 7], sequence = [1, 2, 3, 4, 6]
- deque = [7, 8], sequence = [1, 2, 3, 4, 6, 5]
- deque = [8], sequence = [1, 2, 3, 4, 6, 5, 7]
- deque = [], sequence = [1, 2, 3, 4, 6, 5, 7, 8]

5.

```
class IndexPriorityQueue:
    # This is a 1-indexed PriorityQueue.
    # So the 1st element in self.array and the 1st element in self.keyToIndex are
meaningless
    def __init__(self, capacity):
        # array store (key, value) pairs
        self.array = [(-1, -1)]
        self.capacity = capacity
        self.keyToIndex = [-1 for i in range(capacity + 1)]
        self.n = 0

    def contains(self, key):
        return key >= 1 and key <= self.capacity and self.keyToIndex[key] != -1

    def push(self, key, val):
        # invalid index or index already exist.
        if key < 1 or key > self.capacity or self.keyToIndex[key] != -1:
            return False

        # push to back and swim.
        self.n += 1
        self.array.append((key, val));
        self.keyToIndex[key] = self.n
        self.swim(self.n)
        return True

    def update(self, key, val):
        # invalid index or index does not exist.
        if not self.contains(key):
            return False

        # update and swim
        index = self.keyToIndex[key]
        self.array[index] = (key, val)
        self.swim(index)
        return True

    def top(self):
        if self.n == 0:
            return None
```

```
        return self.array[1]

    def pop(self):
        if self.n == 0:
            return None

        # swap front and back elements, and sink the front
        ret = self.array[1]
        self.swap(1, self.n)
        self.n -= 1
        self.keyToIndex[ret[0]] = -1
        self.sink(1)
        return ret

    # i, j are indexes in self.array
    def swap(self, i, j):
        key1 = self.array[i][0]
        key2 = self.array[j][0]
        self.array[i], self.array[j] = self.array[j], self.array[i]
        self.keyToIndex[key1], self.keyToIndex[key2] = self.keyToIndex[key2],
self.keyToIndex[key1]

    # i is index in self.array
    def sink(self, i):
        while i <= self.n // 2:
            child = 2 * i
            if child+1 <= self.n and self.array[child+1][1] < self.array[child]
[1]:
                child += 1
            if self.array[i][1] < self.array[child][1]:
                break

            self.swap(i, child)
            i = child

    # i is index in self.array
    def swim(self, i):
        while i > 1:
            child = i // 2
            if self.array[i][1] >= self.array[child][1]:
                break
            self.swap(i, child)
            i = child

    def getSize(self):
        return self.n

    def getCapacity(self):
        return self.capacity

ipq = IndexPriorityQueue(8)
for i in range(1, 9):
    ipq.push(i, i)
ipq.update(3, 2)
ipq.update(8, 0)
```

```
while ipq.getSize() > 0:
    print(ipq.pop(), end=' ')
# (8, 0) (1, 1) (2, 2) (3, 2) (4, 4) (5, 5) (6, 6) (7, 7)
```

6.

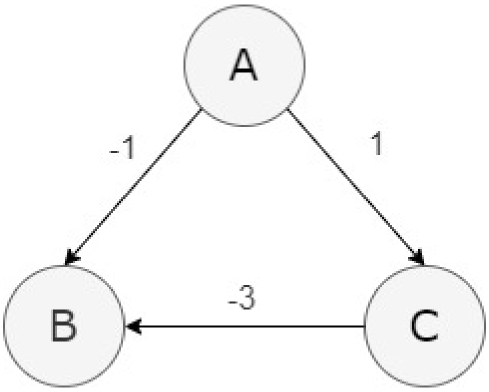
```
def DijkstraAlgo(graph, start, end):
    # initialize data structures
    n = graph.getSize()
    ipq = IndexPriorityQueue(n)
    distTo = [1e10 for i in range(n+1)]
    wayTo = [-1 for i in range(n+1)]
    for i in range(1, n+1):
        if i == start:
            continue
        pq.push(i, 1e10)
    distTo[start] = 0
    pq.push(start, 0)

    # modified Dijkstra Algorithm to find the path
    while ipq.getSize() > 0:
        cur, val = ipq.pop()
        if cur == end:
            break
        for edge in graph.getAdjEdges(cur):
            to = edge.to()
            if not ipq.contains(to):
                continue
            if distTo[to] > distTo[cur] + edge.weight():
                distTo[to] = distTo[cur] + edge.weight()
                wayTo[to] = cur
                pq.update(to, distTo[to])

    # add the path to result
    ret = []
    while end != -1:
        ret.append(end)
        end = wayTo[end]
    ret.reverse()

    return ret;
```

7.

In the following case, our priority queue will be like:

- A: 0   mark A as visited.
- B: -1, C: 1   Since -1 < 1, mark B as visited and set its distance as -1.
- C: 1   marked C as visited and set its distance as 1.

The problem is that the shortest distance to B is $1 + (-3) = -2$ instead of -1. So Dijkstra algorithm won't work on graph with negative edge weights.

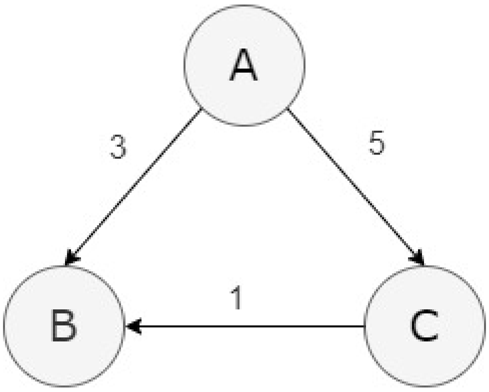


8.

In the following case, I add 4 to all edge weights to make them positive. Following is the process of running Dijkstra algorithm:

- A: 0 mark A as visited.
- B: 3, C: 5 Since 3 < 5, mark B as visited and set its distance as 3.
- C: 5 marked C as visited and set its distance as 5.

Still, the distance to B is $3 - 4 = -1 \neq -2$. So adding a positive offset won't work.



9.

```
def findUniversalSink(matrix, n):
    candidate = 0
    for i in range(1, n):
        if matrix[candidate][i] == 1:
            candidate = i

    for i in range(n):
        if i == candidate:
            continue
        if matrix[i][candidate] == 0 or matrix[candidate][i] == 1:
            return -1

    return candidate
```

```python
mat = [[0, 1, 1, 1], [0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 1, 0]]
print(findUniversalSink(mat, 4)) # 2

mat = [[0, 1, 1, 1], [0, 0, 1, 0], [1, 0, 0, 0], [1, 1, 1, 0]]
print(findUniversalSink(mat, 4)) # -1
```

- If $matrix[i][j] == 0$, then j can not be a universal sink and i may be a universal sink. So we can keep i as the candidate.
- If $matrix[i][j] == 1$, then i can not be a universal sink and j may be a universal sink. So we can discard i and keep j as the candidate.
- We will arrive at 1 candidate and check whether it is a universal sink or not.

The function will return the node index if there exists a universal sink, or -1 otherwise. The time complexity is $O(N)$.

10. 41

First, store all edges in a priority queue called $minHeap$.

Then we keep popping edges from $minHeap$ and join different sets. There are 7 vertices in total.

- $minHeap.pop()$ = B, E 4. sets = [B, E], sum = 4. edges = 1
- $minHeap.pop()$ = A, D 5. sets = [B, E], [A, D], sum = 9. edges = 2
- $minHeap.pop()$ = D, F 6. sets = [B, E], [A, D, F], sum = 15. edges = 3
- $minHeap.pop()$ = A, C 7. sets = [B, E], [A, D, F, C], sum = 22. edges = 4
- $minHeap.pop()$ = B, C 8. sets = [B, E, A, D, F, C], sum = 30. edges = 5
- $minHeap.pop()$ = C, D 9. C, D are in 1 set, continue.
- $minHeap.pop()$ = C, E 10. C, E are in 1 set, continue.
- $minHeap.pop()$ = G, F 11. sets = [B, E, A, D, F, C, G], sum = 41. edges = 6

Now #edges = #vertices - 1, which means the graph is connected. So the weight of MST is 41.