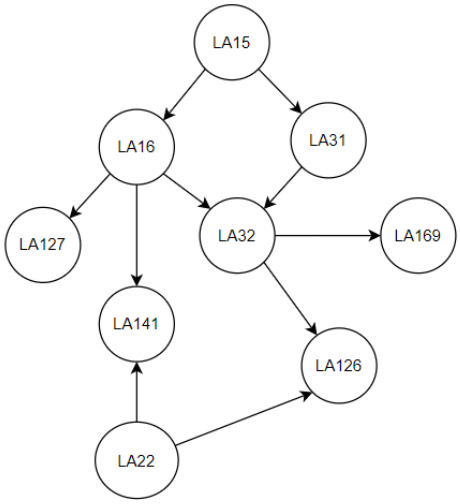
Problem Set 9

Daniel Wang (S01435533)

1. The maximum number of edges in a graph is when the graph is complete. In this scenario, $|E| = O(|V|^2)$, so $O(|E|)$ is not necessarily $O(|V|)$. However, we can induce that $O(\log|E|) = O(\log|V|^2) = O(2 \cdot \log|V|) = O(\log|V|)$

2. The graph of courses can be visualized as follow:



We can use topological sort based on DFS to solve this question. A possible answer can be derived from the following steps, where courses in boldface represents that they are being appended to the current end of list:

(1) LA15 → LA31 → LA32 → **LA169**, backtrack to LA32

(2) LA32 → **LA126**, backtrack to **LA32**, **LA31**, LA15

(3) LA15 → LA16 → **LA141**, backtrack to LA16

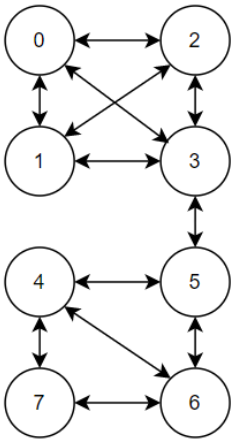
(4) LA16 → **LA127**, backtrack to **LA16, LA15**

(5) LA22, adjacent nodes are visited, so append **LA22**

Based on previous traversal sequences, we can derive a viable course plan using a reverse order:

LA22 → LA15 → LA16 → LA127 → LA141 → LA31 → LA32 → LA126 → LA169

3. The adjacency list can be translated into the following graph:



The DFS sequence can be derived as follows:

(1) 0 → 1, visited = {0}

(2) 1 → 2, visited = {0, 1}

(3) 2 → 3, visited = {0, 1, 2}

(4) 3 → 5, visited = {0, 1, 2, 3}

(5) 5 → 4, visited = {0, 1, 2, 3, 5}

(6) 4 → 6, visited = {0, 1, 2, 3, 4, 5}

(7) 6 → 7, visited = {0, 1, 2, 3, 4, 5, 6}

(8) 7 halts since all neighbors are being visited, visited = {0, 1, 2, 3, 4, 5, 6, 7}

(9) Since all nodes are being visited, DFS halts

Thus, the DFS sequence is 0 → 1 → 2 → 3 → 5 → 4 → 6 → 7

4. The BFS sequence can be derived as follows:

(1) Initialize queue = [0], mark = {0}

(2) Pop 0, queue = [1, 2, 3], mark = {0, 1, 2, 3}

(3) Pop 1, queue = [2, 3], mark = {0, 1, 2, 3}

(4) Pop 2, queue = [3], mark = {0, 1, 2, 3}

(5) Pop 3, queue = [5], mark = {0, 1, 2, 3, 5}

(6) Pop 5, queue = [4, 6], mark = {0, 1, 2, 3, 4, 5, 6}

(7) Pop 4, queue = [6, 7], mark = {0, 1, 2, 3, 4, 5, 6, 7}

(8) Pop 6, queue = [7], mark = {0, 1, 2, 3, 4, 5, 6, 7}

(9) Pop 7, queue = [], mark = {0, 1, 2, 3, 4, 5, 6, 7}

(10) Since the queue is empty, BFS halts

Thus, the BFS sequence is 0 → 1 → 2 → 3 → 5 → 4 → 6 → 7

5. Here I assume that for each element in the heap is a pair, where the first index represents the priority and the second one means the value. I will use a hash map when constructing the heap, where the key is the item name, and the value is the index inside current heap. When heapify, I will not only swap the element in the array but also change the record to match current swapped status. The Python code is shown below:

```
def heapify(arr, start, end, name_to_idx):
    """ min-heapify the array within a range [start, end],
update  name_to_idx"""
    for i in range(end//2-1, start-1, -1):
        l, r = 2*i+1, 2*i+2
        max_id = i
        if arr[max_id][0] > arr[l][0]:
            max_id = l
        if r < end and arr[max_id][0] > arr[r][0]:
            max_id = r

        # swap name_to_idx
        name1, name2 = arr[i][1], arr[max_id][1]
        name_to_idx[name1] = max_id
        name_to_idx[name2] = i

        # swap priority
        arr[i], arr[max_id] = arr[max_id], arr[i]
        if max_id != i:
            heapify(arr, max_id, end, name_to_idx)

def update_priority(arr, name, val, name_to_idx):
    idx = name_to_idx[name]
    arr[idx][0] = val
    heapify(arr, 0, len(arr), name_to_idx)
```

When we run the following code in the main loop where we try to change the priority of A from 1 to 6:

```
arr = [[1, "A"], [0, "B"], [3, "C"], [2, "D"], [5, "E"]]
n = len(arr)
name_to_idx = {arr[i][1]: i for i in range(n)}
heapify(arr, 0, n, name_to_idx)

# before update
print(arr)
print(name_to_idx)
print("----------")

# after update
update_priority(arr, "A", 6, name_to_idx)
print(arr)
print(name_to_idx)
```

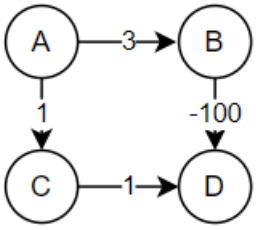
The console output satisfies the property of heap and is demonstrated below:

```
[[0, 'B'], [1, 'A'], [3, 'C'], [2, 'D'], [5, 'E']]
{'A': 1, 'B': 0, 'C': 2, 'D': 3, 'E': 4}
----------
[[0, 'B'], [2, 'D'], [3, 'C'], [6, 'A'], [5, 'E']]
{'A': 3, 'B': 0, 'C': 2, 'D': 1, 'E': 4}
```

6. Assume the input of Dijkstra algorithm contains *graph*, *weights*, and *src*. Where *graph* is stored as an adjacency list, *weights* is a dictionary where key is the edge and value is the path cost, and *src* is the vertex name of source node. The modified code is shown as follows, where we adopt the same function from Q.5:

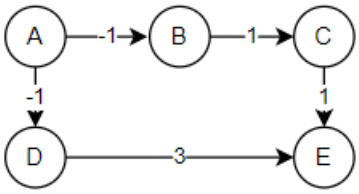
```python
def dijkstra(graph, weights, src):
    all_nodes = set(graph.keys())

    pq = [] # (vertex, current distance)
    dists = {} # {vertex: distance}
    name_to_idx = {} # {vertex: current index in pq}
    for (i, node) in enumerate(all_nodes):
        pq.append([float("inf"), node])
        dists[node] = float("inf")
        name_to_idx[node] = i

    # initialize for source node
    dists[src] = 0
    update_priority(pq, src, 0, name_to_idx)

    # traverse all nodes
    result = []
    while len(pq):
        # pop smallest while maintain name_to_idx
        dist, node = pq[0]
        last_name = pq[-1][1]
        pq[0] = pq[-1]
        name_to_idx[last_name] = 0

        pq.pop()
        heapify(pq, 0, len(pq), name_to_idx)

        # add result and traverse its neighbor
        result.append(node)
        for node_next in graph[node]:
            new = dist[node] + weights[(node, node_next)]
            if new < dists[node_next]:
                dists[node_next] = new
                update_priority(pq, node_next, new, name_to_idx)

    return result
```

7. Assume the following acyclic graph with a source node of A:

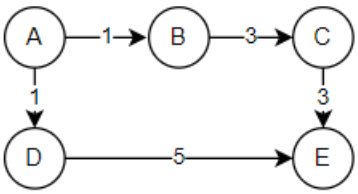


The minimum spanning tree should be {AC, AB, BD}, having a cost of -96. Using Dijkstra started with node A, it traverses to C and then to D, backtracking to C and A, and traverses to B. At this time, D is already being marked beforehand, so the edge BD will be ignored. In this scenario, Dijkstra yields a suboptimal result {AC, AD, AB}, yielding a cost of 5 and is greater than -96.

8. Consider the following acyclic graph and assume the source node is A:



The optimal solution is {AB, AD, BC, CE}, yielding a cost of 0. Now we offset the value of edges by +2 so that all edges become positive. It becomes:



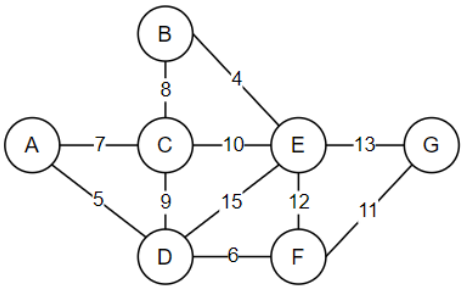
Using Dijkstra started from A, the appended edges will have the order of AB, AD, BC, DE. The edge CE will not be considered since the path A → D → E has a cost of 6, while the path A → B → C → E has a cost of 7. In this scenario, Dijkstra yields a result {AB, AD, BC, DE}, yielding a cost of 2 and is greater than 0.

9. The method can be decomposed into 2 steps.
   (1) Since universal sink node can have at most 1, the first steps we can try is to eliminate n-1 possible nodes by judging A[i][j]. If A[i][j] = 0, it means there is no link from node i to node j, implying that j cannot be a sink node. Thus, we can eliminate j from candidate pool. If A[i][j] = 1, it means there is a link from node i to node j, meaning that i cannot be a sink node. Doing n-1 iterations will make the number of candidates to be 1.
   (2) For the current candidate, we will search whether all nodes are connecting to it

and it cannot connect to others. The Python code is shown as follows:

```python
def has_sink_node(A):
    n = len(A)

    # make i < j, and eventually j will be n
    # where i will be a possible candidate
    i, j = 0, 1
    while j < n:
        if A[i][j] == 0: # j impossible
            j = j + 1
        else: # i impossible
            i, j = j, j + 1

    # check whether i is really sink node
    idx = i
    for i in range(n):
        if i == idx:
            continue
        if not (A[i][idx] == 1 and A[idx][i] == 0):
            return False

    return True
```

10. The graph can be represented as follow:



Kruskal's algorithm tries to find smallest edge in the graph for each iteration. The algorithm stops when all nodes are being visited. Let assume *result* be the path list, and *conn* be the connected components for current iteration.

(1) BE = 4 , result = [BE], conn = [{B,E}]

(2) AD = 5, result = [BE, AD], conn = [{B,E}, {A,D}]

(3) DF = 6, result = [BE, AD, DF], conn = [{B,E}, {A,D,F}]

(4) AC = 7, result = [BE, AD, DF, AC], conn = [{B,E}, {A,C,D,F}]

(5) BC = 8, result = [BE, AD, DF, AC, BC], conn = [{A,B,C,D,E,F}]

(6) CE = 10, since C and E are in the same connected component, skip it

(7) FG = 11, result = [BE, AD, DF, AC, BC, FG], conn = [{A,B,C,D,E,F, G}]

(8) Since all nodes are in the same connected component, break the loop

Eventually, the minimum spanning tree will look like below: