Math 182: Final exam

You have 3 hours to complete this exam since you open it.

Name: Jiayu Li

ID number: 605 - 348 - 766

Problem 1

The following algorithm attempts to find the minimum spanning tree of a connected (undirected) graph `G` with (potentially negative) integer edge weights.

- Simultaneously add `1` to every edge weight. Keep doing this until every edge has positive weight.
- Find the MST of this new graph (where all the edge weights are positive), using an algorithm we already know, such as Prim.

If it works, explain why. If not, give a connected graph `G` where it fails (and show what happens when you run the algorithm on it).

It works.

Consider any spanning tree on the graph: Tree(G), After the first step of the algorithm, we get Tree(G'). s.t.:

$$\text{Weight}(\text{Tree}(G')) = \text{Weight}(\text{Tree}(G)) + (n-1)\cdot K$$

Weight(·) is the weight sum of the tree, n is the number of vector in the graph G, and K is the increment of the first step of the algorithm.

Observe that for any Tree, $(n-1)\cdot K$ is constant, regardless of the tree's structure. So the partial order relationship of Weight does not change before and after the first step of the algorithm. Then the minimum spanning tree of the changed graph G' is still the minimum spanning tree of the original graph G.

Problem 2

The following algorithm attempts to find the shortest path from `s` to `t` in a DAG (directed acyclic graph) `G` with (potentially negative) integer edge weights.

- Simultaneously add `1` to every edge weight. Keep doing this until every edge has positive weight.
- Find the shortest path from `s` to `t` in this new graph (where all the edge weights are positive), using an algorithm we already know, such as Dijkstra.

If it works, explain why. If not, give a DAG `G` where it fails (and show what happens when you run the algorithm on it).

It doesn't work!

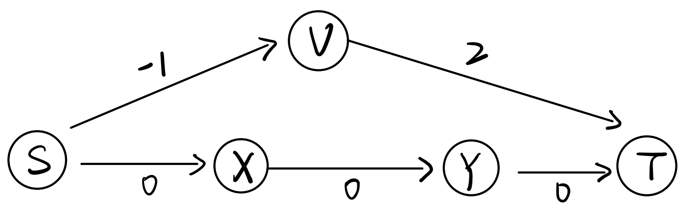
Consider any path on the graph: Way(G); After the first step of the algorithm. we get Way(G'), s.t.:

$$\text{Weight}(\text{Way}(G')) = \text{Weight}(\text{Way}(G)) + \text{Edge}(\text{Way}(G)) \cdot K.$$

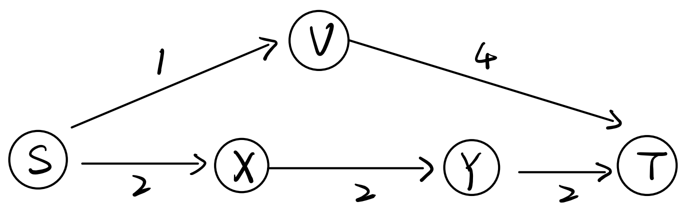
Weight(.) is the weight sum of the tree; Edge(.) is the edge number on the path; K is the increment of the first step of the algorithm.

Observed that Edge(Way(G)) · K is not constant for different paths. The longer the path, the each increment will be larger. Then the partial order relationship may change during the process.

construction counterexample is as follows:



shortest path: S→X→Y→T



shortest path: S→V→T.

Problem 3

Write an algorithm to determine the minimum number of terms required to complete every course, given their prerequisites.

More precisely, the function `terms_required` takes as input an unweighted DAG (directed acyclic graph) `G`, and returns the smallest k such that there is a partition $(P_0, P_1, \ldots, P_{k-1})$ of the vertices of `G`, where an edge can only point from a vertex in P_j to a vertex in P_i if $j > i$.

Write an efficient implementation of `terms_required`.

You can assume that there's a topological sort function already (you don't need to reimplement it).

This problem can be regarded as: Number each node num(i), so that each edge (u, v) satisfies: num(u) > num(v), when u > v. It is required to minimize max(num(i)).

We use the idea of topological sorting + greedy algorithm: after obtaining the topological order, for the edges (u,v) existing in the topological order, we let num(u) = num(v) + 1, which can make the growth of num as slow as possible.

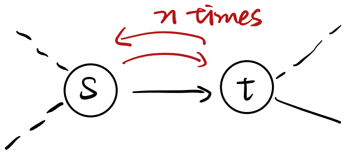
```
# G is the adjacency matrix G[i][j] of the graph
def terms_required(G):
    # Make a topological sorting on G to obtain the set of topologically ordered nodes of G
    nodeOrder = topsort(G)
    n = len(G)
    num = [0 for i->1...n]
    #Enumerate any pair of vertices in topological order
    for i in [1... n]:
        for j in [i+1 ... n]
            # greedy fill num[j] = num[i]+1 to minimize the growth of the num(i)
            if G[i][j] is True:
                num[j] = num[i] + 1
    return max(num)
```

Problem 4

The function `visit_many_times` takes as input an unweighted, undirected graph `G`, two vertices `s` and `t`, and a positive integer `n`. The output is the shortest walk from `s` to `t` in `G` which visits `s` and `t` at least `n` times each (you can assume that the input graph has such a walk).

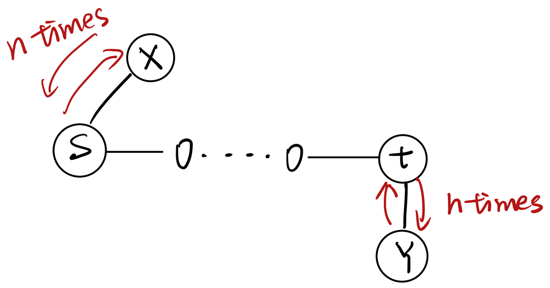
Write an efficient implementation of `visit_many_times`.

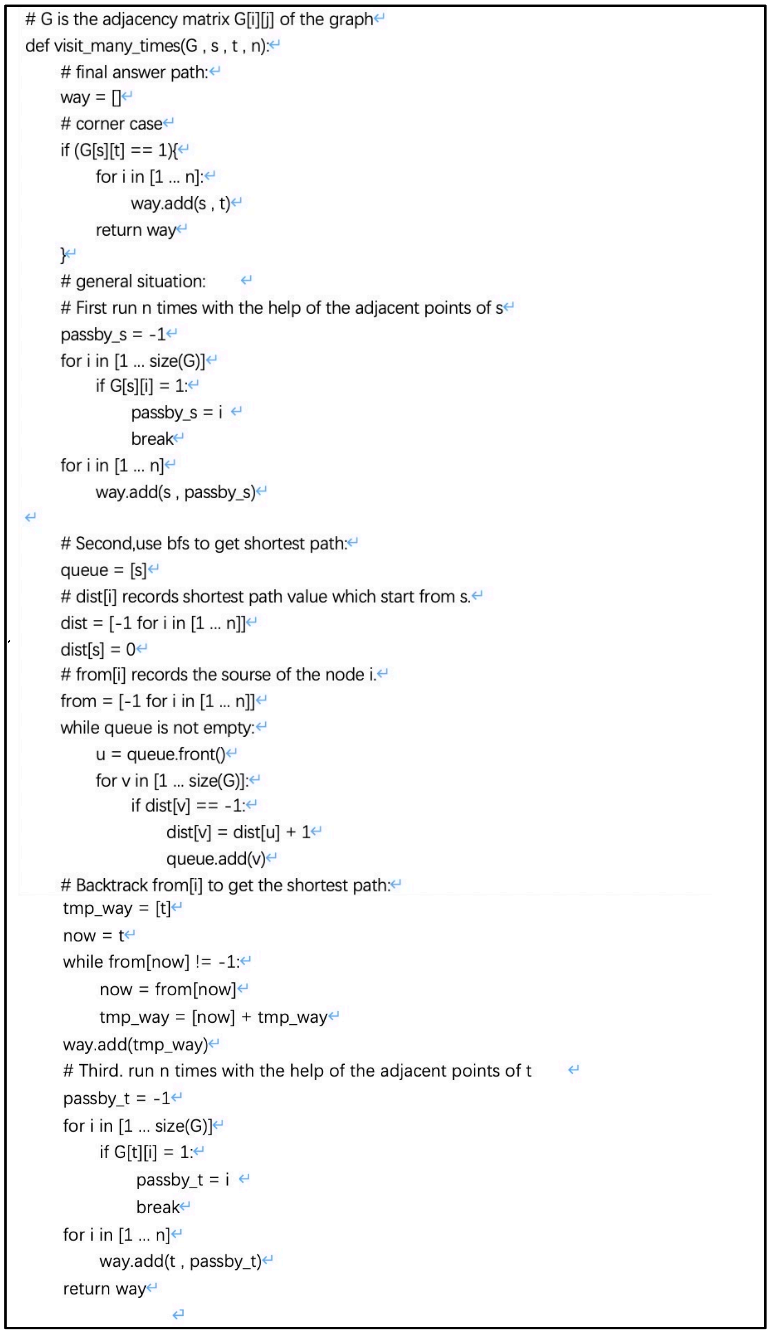
When s and t are adjacent:
it is optimal to repeatedly visit n times between s and t.



when s and t are not adjacent:

1. use the adjacent points around s to go back and forth n times,
2. use bfs to take the shortest path to t
3. use adjacent points of t to go back and forth of n times.





```
# G is the adjacency matrix G[i][j] of the graph
def visit_many_times(G , s , t , n):
    # final answer path:
    way = []
    # corner case
    if (G[s][t] == 1){
        for i in [1 ... n]:
            way.add(s , t)
        return way
    }
    # general situation:
    # First run n times with the help of the adjacent points of s
    passby_s = -1
    for i in [1 ... size(G)]
        if G[s][i] = 1:
            passby_s = i
            break
    for i in [1 ... n]
        way.add(s , passby_s)

    # Second,use bfs to get shortest path:
    queue = [s]
    # dist[i] records shortest path value which start from s.
    dist = [-1 for i in [1 ... n]]
    dist[s] = 0
    # from[i] records the sourse of the node i.
    from = [-1 for i in [1 ... n]]
    while queue is not empty:
        u = queue.front()
        for v in [1 ... size(G)]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                queue.add(v)
    # Backtrack from[i] to get the shortest path:
    tmp_way = [t]
    now = t
    while from[now] != -1:
        now = from[now]
        tmp_way = [now] + tmp_way
    way.add(tmp_way)
    # Third. run n times with the help of the adjacent points of t
    passby_t = -1
    for i in [1 ... size(G)]
        if G[t][i] = 1:
            passby_t = i
            break
    for i in [1 ... n]
        way.add(t , passby_t)
    return way
```

Problem 5

Describe how the following problem can be interpreted as a max flow problem:

The input is a list of courses, each with its own enrolment cap, and a list of students, each with their own list of courses they want to take. A student can only take up to 5 courses a term (but their list of courses might contain more than 5 courses). The goal is to assign students to courses so that the most seats are filled up (the combined total of seats over all classes).

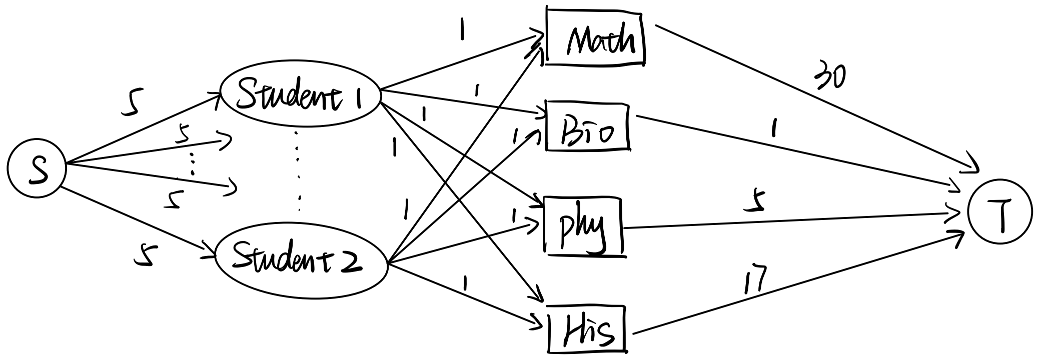
For instance, here is a possible input:

- Math has an enrolment cap of at most 30 students.
- Biology has an enrolment cap of at most 1 student.
- Physics has an enrolment cap of at most 5 students.
- History has an enrolment cap of at most 17 students.
- Student 1 wants to take Math, Biology and Physics.
- Student 2 wants to take Math, Biology and History.

In this case, if we let Student 1 take Math, Biology and Physics, and we let Student 2 take Math and History, this fills up five seats total, which is the most we can fill up. Note that both students can't take biology here, since Biology only has 1 seat.

S: source point ; T: sink point.

S has a directed edge with all students (the direction is from S to students), and the capacity is 5. The purpose is to select up to 5 cources for each student. Then T established a directed edge (direction from the course to T) will all cources, and the capacity is the enrollment cap of the course. The purpose of this operation is to make each course only be selected enrollment cap times. Then the students and courses to establish a direct edge (direction by the students to the course), capacity is 1. the purpose of this operation is that each student can only choose one of the same course.



Problem 6

Give a polynomial-time reduction from graph 3-coloring to graph 4-coloring.

More precisely, recall that a **proper vertex k-coloring** assigns one of k colors to every vertex, such that adjacent vertices get different colors. Define the following sets:

$$\begin{aligned} X &= \text{all graphs} \\ A &= \text{all graphs which admit a proper vertex 3-coloring} \\ B &= \text{all graphs which admit a proper vertex 4-coloring} \end{aligned}$$

Give a polynomial-time reduction from (X, A) to (X, B).

(Hint: You only need to add one vertex to the graph).

Add a point of the fourth color on the basis of 3 coloring to make it a 4 coloring. Then it can be proved that adding 3 color satisfy 4 coloring Meanwhile. if this satisfies 4 coloring, the situation of 3 coloring must be correct. So it is a polynomial-time reduction.