Name: Jungbin Choi
Due December 1, 2023

COMP 550-001
Homework 5 (16 points)

Notes. Collaboration is allowed, and you may use any resources you'd like. You may call any algorithm given in the notes without providing details. However, you should write and submit your solutions individually, and each submission will be graded individually. In your submission, you should list your collaborators (including course staff) and cite every resource you use. For each problem, I encourage you to attempt it for at least 30 minutes before consulting any people or resources.

Your submission should be a single PDF file. I recommend using Overleaf; a template is on Canvas. If your answers are not clear, you might not get full credit. You should describe your algorithms in pseudocode similar to the examples given in the lectures/notes. I recommend also including a brief description in full sentences.

Problem 1 (10 points). Suppose we are serving dishes to customers. There are n customers $\{c_1, \ldots, c_n\}$ and k dishes $\{d_1, \ldots, d_k\}$. We are given an array A of n lists; each list contains at least one dish. A customer c_i is "satisfied" if we serve c_i a dish in $A[i]$. We are also given an array B of k positive integers. Every dish d_j can be served to at most $B[j]$ customers. Our goal is to maximize the number of satisfied customers. None of the dishes can be shared.

a. (4p) Give an $O(n^2k)$-time DP algorithm for this problem. In particular, your algorithm should return a "serving plan" P containing k lists. The list $P[j]$ should contain the customers that get served d_j.

b. (4p) Prove that your algorithm is correct.

c. (2p) Analyze the running time of your algorithm.

Solution.

a.
```
ALG(A, B):
    P = [{}] * k   // Size k array with empty lists
    for i = 1 ... n:
        // Find the dish with the maximum servings left
        int max_dish = A[i].get(1)
        for dish in A[i]:
            if B[dish] > B[max_dish]:
                max_dish = dish
        if B[max_dish] == 0:
            for dish in A[i]:
                for prev_cust in dish:
                    if a prev customer can be served a different dish:
                        P[new dish].add(previous customer)
                        P[old dish].remove(prev customer)
```

```
                    p[old dish].add(current customer)
        else:
            P[max_dish].add(i)
            B[max_dish] -= 1
    return P
```

b. To solve the problem, I reduced the problem in the question into a semi-bipartite matching problem (some edges will share an endpoint). The two columns of vertices are the customers (left) and the dishes (right), with the intermediate edges between them being which customer is satisfied by which dish (the values of A's lists). From there, I further reduced the semi-bipartite matching problem into a maximum flow problem. The customer (left) side come from an arbitrary starting vertex S with the edges going left to right with a capacity of one (if the flow equals 1, then that customer is satisfied). The intermediate edges all get converted into a directed edge going left to right with a capacity of one for the same reasons. Then, all the dish nodes will go towards the target/end node T with each edge capacity equal to how many each dish can be served (values of B). The max flow solution would equal the max number of customers satisfied.

   Now, for each customer, we choose the dish with the highest servings left because every customer afterwards will have a better chance to get that dish to be satisfied. In the case that a customer doesn't have a dish to get (denoted by the max dish being zero), check each dish that can satisfy the customer, and check if any previous customer that has gotten that dish can be satisfied with a different dish. If yes, change, if not, move on.

c. Initializing array P takes $O(k)$ time. The first for-loop will take $O(n)$ time. The first inner for-loop will take O(number of dishes for customer i), which when added up will be a worst-case max of $O(nk)$ (when every customer can be satisfied with every dish). The second inner for-loop will take $O(nk)$ time as well for the same reasons. Everything else within and outside will take constant time. Thus, the sum of the inner loops is $O(2nk)$. Since this occurs per iteration of the outer for-loop, the total running time will be $O(k + n(2nk))$, which will simplify down to $O(n^2k)$.

Problem 2 (6 points). Suppose we want to reduce VertexCover to DomSet (as defined in the notes). Recall that the input to both problems is a connected, undirected graph and a positive integer. Consider the following reduction: given an instance (G, k) of VertexCover, simply return (G', k') as the instance of DomSet, where $G' = G$ and $k' = k$. You should assume G has $n \geq 2$ vertices.

a. (3p) Does this reduction satisfy the "forward" direction? Explain your answer.

b. (3p) Does this reduction satisfy the "backward" direction? Explain your answer.

Solution.

a. Let the "forward" direction be $VERTEXCOVER \rightarrow DOMSET$. The reduction is **SATISFIED** in this direction. The reason is that in the set S (the vertex cover), every edge $e = (u, v)$ in G would be connected to at least one of the vertices in S. Now consider any vertex u in G that is not in S. All the edges connected to u will have one of its end points be a vertex in S (due to the property of a vertex cover), meaning any vertices outside of S will be directly connected to a vertex within S. This is the property that makes S also be the dominant set (which states every vertex be either in S' or be adjacent to a vertex within S').

b. Let the "backward" direction be $DOMSET \rightarrow VERTEXCOVER$. The reduction is **NOT SATISFIED** in this direction. Consider an example G' of 5 vertices, $k = 1$, and contains the following edges: $E = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$. The shape of the graph is a square with one diagonal edge. A "yes" instance for S' (the dominant set) is just vertex 1 as it is connected to every other vertex. Now claim that S' is also a vertex cover. This is not true as if only vertex 1 is in the set, then the edges $(2, 4)$ and $(3, 4)$ would not be covered by vertex 1. Thus, a "yes" instance of dominant set was not able to be mapped to a "yes" instance of a vertex cover, meaning the backward direction is not satisfied.