CS 330 – Fall 2018, Assignment 5 Solutions

Question 1 (10 pts)**.** *Chapter 6, Problem 13, on p. 324.*

Algorithm: Construct a weighted directed graph $G = (V, E)$, where each node represents a stock, and there is a directed edge (i, j) between each pair of stocks with cost $c(i, j) = -\log r_{ij}$. Run the extension of the Bellman-Ford Shortest-Path algorithm from a single source in the lecture slides to detect if there is a negative cost cycle in this graph. To output a cycle, we run Bellman-Ford and observe a node i whose distance decreases from the source in the $|V|$'th iteration. By trying to trace back to the source from i using the previous labels, we instead will first encounter a negative weight cycle from some intermediate node k looping back to itself. If we find such a negative cost cycle, it corresponds to a opportunity cycle, otherwise there is no opportunity cycle in the graph.
Correctness: We prove equivalence between an opportunity cycle and a negative cost cycle in the constructed graph. An opportunity cycle was a sequence of trades $(k, l_1), (l_1, l_2), \ldots (l_m, k)$ that resulted in value greater than where we started. If we started with \$1, our ending value is:

$$\$1 \cdot r_{k,l_1} \cdot r_{l_1,l_2} \ldots r_{l_m,k}$$

which is greater than \$1 iff $\Pi[r_{i,j}] > 1$. Taking the log of both sides, this is equivalent to: $\log \Pi[r_{i,j}] > \log 1$, or (using the hint in the handout): $\Sigma \log r_{i,j} > 0$.

Similarly, a cycle in the graph $(k, l_1), (l_1, l_2), \ldots (l_m, k)$ is a negative weight cycle iff $\Sigma[c_{i,j}] < 0$. By definition of $c_{i,j}$, this is $\Sigma[-\log r_{i,j}] < 0$, or also $\Sigma \log r_{i,j} > 0$.
Running time: The Bellman-Ford Algorithm runs in O(mn) time, then scanning to find a node that contains a negative cycle and backtracking takes additional O(n) time. So we have a polynomial-time algorithm running in O(mn) time.

Question 2 (10 pts)**.** *Chapter 7, Exercise 10, on p. 419.*

First, observe that by reducing one of the edge weights, the maximum flow can only reduce and not increase.

There are two cases. In the simpler case, edge e^* is NOT part of the minimum cut. (Thus $f(e^*) < c(e^*)$). In this case, the old max flow remains the max flow in this new setup.

Let us assume now that $f(e^*) = c(e^*)$ and e^* is part of the min cut. Then, reducing its capacity only decreases the value of the cut, hence it will still be part of the min cut. Thus the total value of the new flow will be one unit less than the total value of the old flow. We need to make sure to reinstate the flow conservation property. Thus, find one path from s to t with positive flow values using edge e^* and reduce the flow value along each edge by 1. This can be done in worst case in $O(n + m)$ time. (By for example doing BFS (using only edges with positive flow) from s to the left end of e^* and from the right end of e^* to t, respectively.) When we reduce the values by one, we take into consideration that there is an integer solution to the max flow problem, hence each non-zero edge can be reduced by at least one.

Question 3 (10 pts)**.** *Chapter 7, Exercise 16, on pp. 422-23.*

We will build a graph which will enable us to decide whether every advertiser can advertise to r users per minute based off of the max flow of the graph. We will add a node a_i for each advertiser and a node u_j for each user. Add a directed edge from u_j to a_i with capacity 1 if the user is part of some demographic G_i that the advertiser wants to advertise to. Lastly, we add a source and a sink to the graph, with an edge of capacity 1 from the source to each of the users and an edge of capacity r_i from the advertiser i to the sink.
Claim: There is an arrangement of advertisements that reaches their goal number of users iff the max flow of this graph is R = the sum of $(r_1, r_2, \ldots, r_m)$.

($\leftarrow$) If the max flow is R, then there is an integral max flow of value R, and in this integral flow, values of flow between advertisers and users are 0/1. Exactly R of these edges are equal to 1 since the flow on R edges from the source to the users is 1 (there must be at least R users). Then, the flow R will saturate the flow from the advertisers to the sink. Therefore, there is an arrangement of advertisers that reaches the goal number of users.

Proof: ($\rightarrow$) Take a schedule where there is an arrangement of advertisements that reaches advertisers goal number of users. To derive a max flow of value R, assign a flow value of 1 to any user-advertiser edge used in the schedule. For every user used, put a flow of 1 from the source to that user and from the advertiser to the sink (to conserve flow). That means that R users reach the advertisers. Therefore, there is a flow of value R, and since it saturates the capacity to the sink, it is a max flow.

The running time of this algorithm is the time that it takes to build the graph plus the time it takes to run Ford-Fulkerson to find the max flow. In order to build the graph we pair up each advertiser with users that are a part of demographics the advertisers are interested in. This will take O($n * m * k$) time because there are m advertisers, n users, and up to k groups to consider. In the worst case, the graph will have an edge between every advertiser and every user, plus the edges between the source and the advertisers and the users and the sink, which is O($n*m+n+m$) = O($n * m$) edges. Ford-Fulkerson takes O(mC) time where m is the number of edges in the graph and C is the total capacity out of the source, so Ford-Fulkerson takes O($n * m * (n)$) time. In total then the running time is O(nmk) + O(n^2m).

Question 4 (20 pts)**.** *Programming Assignment*

(a) As in lab, we define a recurrence, M, that asks whether sum i is achievable using a subset of the first j elements. We say that a sum of zero is always achievable, and observe that a larger sum is achievable with j elements if: it was EITHER attainable with $j-1$ elements OR it was attained by adding the jth element (with value a_j) to a correspondingly smaller sum achievable with $j-1$ elements. In a recurrence:

$$M(i,j) = \begin{cases} 1 & \text{if } i = 0; \\ 1 & \text{else if } j = 0; \\ M(i, j-1) & \text{else if } i < a_j; \\ M(i-a_j, j-1) \text{ OR } M(i, j-1) & \text{otherwise} \end{cases}$$

Now let B denote the sum of the whole sequence, $B = \sum_{i=1}^{n} a_i$. The smallest residue is related to the best achievable sum closest to $B/2$, which corresponds to a residue that is *twice* the difference between that sum and $B/2$. So, our implementation needs to compute a matrix M with rows 0 to $B/2$, corresponding to the range of possible sums we care about, and has columns 0 to n, corresponding to the elements considered. At each iteration, to compute $M[i][j]$, the program follows the 2-way choice in the recurrence: either discard or use element j:

```
def partition(a):
    B = sum(a)
    num = len(a)

    M = [[0 for i in range(num+1)] for i in range(B/2+1)]

    best_achievable_sum = 0
    for i in range(num+1):
        M[0][i]=1
    for i in range(1,(B/2)+1):
        M[i][0]=0
    for i in range(1,(B/2)+1):
        for j in range(1,num+1):
            M[i][j] = M[i][j-1]
            if i >= a[j]:
                M[i][j] = M[i][j-1] or M[i-a[j]][j-1]
            if j==num and M[i][j] == 1:    // in last column...
                best_achievable_sum = i;   // ... track largest sum
```

Now, we want to step backwards through the matrix to determine which elements are "included" in the subset. These elements will be assigned -1 in the output sequence, s. The stepping back process mirrors the matrix-filling process. Beginning in the $j = n$ column at i = best_achievable_sum, we check to see if the a_{j-1} element was included in the sum by: 1) Checking that current sum is greater than a_{j-1}, then 2) if the cell $M_{(i-a_{j-1}),j-1}$ is true. If yes, s is updated to give the $j-1$ element -1, and i is updated to be the new sum. The following python code implements the stepping back process. It prints s and checks what residue s achieved.

```
s = [1 for i in range(num)]
i = best_achievable_sum
while i > 0 :
    for j in range(num, 0, -1):
        if i >= a[j]:
            if M[i - a[j]][j-1] == 1:
                i -= a[j]
                s[j]=-1
        if i == 0:
            print("s: " + str(s))
            part = [a*b for a,b in zip(a, s)]
            res=abs(sum(part))
            print("Residue achieved: " + str(res))
            return
```

The matrix uses space n*(B/2), and in the process of filling, each cell is visited once, with constant time calculations and lookups being performed at each iteration. The stepping backwards process is even quicker as i is updated to be the new sum, making it that every cell does not need to be visited. This leads to a pseudopolynomial running time of O($n * B$).

(b) Let's do the implementation first, achieving an O($n \log n$) running time. By using a priority queue in our implementation (max-heap), we store all n elements, then repeatedly pop the two max elements and push back just the difference between the two (it's fine to discard the zeros). We continue this process until we reach a priority queue of just one element. With a priority queue, it takes O($\log n$) time to insert, pop, and push, and we do exactly n insertions at the start of the algorithm, $2(n-1)$ pops and $n-1$ pushes. (We can just output the residue without a final push). All told, O($n \log n$) time.

To see that this residue is achievable, consider forming a graph with n vertices, one per element, and draw an edge between i and j when that pair of elements is popped and differenced against one another. Also, associate that edge with a label equal to the value of the difference performed. Each differencing operation eliminates one element, so our graph will have $n-1$ edges at the end, and it spans all vertices, since every element was involved in at least one comparison. Recall that a graph with $n-1$ edges spanning all vertices is a tree. Now take the root of the tree to be the last element remaining (at level 0 of the tree) and label all elements at even levels of the tree '+' and all nodes at odd levels of the tree '-'. Since every edge connects a '+' to a '-', each differencing operation removed an equal amount of weight from the positive and negative subsets. The leftover, the residue, is the unaccounted-for amount remaining at the root.

(c) A full-credit answer to this problem would plot the residues as a function of n varying as powers of 10. What we expected that you would observe is that the exact algorithm from (a) produces relatively small residues, often many orders of magnitude smaller than the random inputs, but that the running time gets painfully slow as the quantity nB gets large. Maybe some of you managed to get it to run up to $B = 10^10$.

The heuristic produces decent residues, but they start to become substantially worse than optimal, perhaps by several orders of magnitude when the exact minimum residue is very small for large B. However, with no running time dependence on B, this algorithm will run blazingly fast regardless of the range of the random numbers.