CSOR 4246 Algorithms for Data Science (Fall 2022)
Problem Set #3t

Xinhao Li - xl2778@columbia.edu

2022/11/23

Problem 1

Algorithm to find the length of their common subsequence

This problem is asked us to design an algorithm which can efficiently compute the length of their longest common subsequence and also return longest common subsequences. This is clearly a dynamic programming problem. I will follow the several steps to introduce the algorithm.

(a) Clearly define the subproblems.

Since we have three sequences here, three pointers i, j, and k will be defined first and will be used to traverse the X, Y, and Z sequences, respectively. The length of the longest common subsequence will be stored in a three dimensional matrix M with lengths of each dimension being len(X)+1, len(Y)+1, and len(Z)+1, respectively. The element in matrix M, M[i][j][k] has the meaning of the length of the longest common subsequence among the first i-th sequences in X, j-th sequences in Y, and k-th sequences in Z.

(b) Explain the recurrence.

As we introduced before, M[i][j][k] stores the length of the longest common subsequence among the first i-th sequences in X, j-th sequences in Y, and k-th sequences in Z (which not including X[i], Y[j], and Z[k] themselves). If X[i] == Y[j] == Z[k], this means the current element in each sequence is the same. So, we will add 1 to the current length of the longest common subsequence and we can define: M[i][j][k] = M[i-1][j-1][k-1] + 1.
If not X[i] == Y[j] == Z[k], this means the current element in each sequence is not the same. We will just simply find the maximum length among all the values stored in matrix M which are adjacent to the current step (M[i][j][k]).
Mathematically, we have:

$$M[i][j][k] := \begin{cases} M[i-1][j-1][k-1]+1 & \text{if } X[i] == Y[j] == Z[k] \\ max(M[i-1][j][k], M[i][j-1][k], M[i][j][k-1]) & \text{otherwise} \end{cases} \tag{1}$$

(c) State boundary conditions.

When i = j = k = 0, this means that currently, there is no any element in the sequence. So consequently, we have:

$$M[0][j][k] = M[i][0][k] = M[i][j][0] = 0 \tag{2}$$

(d) Analyze time.

We have three pointers here and each pointer will traverse the whole sequence. So the time complexity will be $O(n^3)$.

(e) Analyze space.

As mentioned in (a), a three dimensional matrix M will be initialized and the length of each dimension being len(X)+1 , len(Y)+1 , and len(Z)+1. Overall the space needed is: $O((n+1)^3) = O(n^3)$

(f) The order to fill in subproblems.

The sequence to fill the 3D matrix should follow:

1. Fill in the last dimension first and we will have M[0][0][k] where k is from 0 to len(Z). This is like we are filling a line in the box.
2. Then we can start to fill in the second dimension and we will have M[0][j][k] where j is from 0 to len(Y) and k is from 0 to len(Z). In this step, we are filling in the plane.
3. The last step is to fill in the last dimension and we will have M[i][j][k] where i is from 0 to len(X), j is from 0 to len(Y) and k is from 0 to len(Z). This step is like to stack planes in a box.

(g) Pseudocode

```
def longest_common_subsequence(X, Y, Z):
 x = len(X)
 y = len(Y)
 z = len(Z)
 Initialize M = matrix[x+1][y+1][z+1]
 for i in range(x):
  for j in range(y):
   for k in range(z):
    if i==0 or j==0 or k==0:
     M[i][j][k] = 0
    elif X[i]==Y[j]==Z[k]:
```

```
      M[i][j][k] = M[i-1][j-1][k-1] + 1
    else:
      M[i][j][k] = max(M[i-1][j][k], M[i][j-1][k], M[i][j][k-1])
return M[x][y][z]
```

Algorithm to return the longest common subsequence

To find the longest common subsequence, we will start from the results found by running the previous algorithm, which is M[x][y][z] and three pointers will be initialized as well. But this time, they will just traverse from the end to the beginning. Starting from M[x][y][z], we will see first if the current last letters in each sequence are equal. If they are equal, then this letter can be added to the results. If not all of them are equal, this means the current element in the matrix is obtained by maximizing the values of three adjacent elements. At this point, there are several possibilities. First of all, if there is only one maximum value, we can just move the pointer one step back on its own direction/axes. For example, the point will be moved from M[i][j][k] to M[i][j][k-1]. If there are two or three elements in the matrix which are equal to the maximum value, then we can use recursion on their own direction-s/axes. For example, if M[i][j-1][k] = M[i][j][k-1] = max value, then we will initialize two new searches starting from M[i][j-1][k] and M[i][j][k-1], respectively. Finally, the longest common subsequences will be stored in the opposite direction. Notice that, there could be more than one longest common subsequence for the given sequences. We can simply reverse all found longest common subsequence and return them.

Pseudocode

```
lcs = list()
i = len(X)
j = len(Y)
k = len(Z)
def find_longest_common_subsequence(i, j, k, lcs):
 res = list()
 while i > 0 and j > 0 and k > 0:
  if X[i]==Y[j]==Z[k]:
   lcs.append(X[i])
   i -= 1
   j -= 1
   k -= 1
  else:
   if M[i-1][j][k] > Max(M[i][j-1][k], M[i][j][k-1]):
    i -= 1
   elif M[i][j-1][k] > Max(M[i-1][j][k], M[i][j][k-1]):
    j -= 1
   elif M[i][j][k-1] > Max(M[i-1][j][k], M[i][j-1][k]):
    k -= 1
   elif M[i-1][j][k]==M[i][j-1][k]:
```

```
            find_longest_common_subsequence(i-1, j, k, lcs)
            find_longest_common_subsequence(i, j-1, k, lcs)
        elif M[i-1][j][k]==M[i][j][k]-1:
            find_longest_common_subsequence(i-1, j, k, lcs)
            find_longest_common_subsequence(i, j, k-1, lcs)
        elif M[i][j-1][k]==M[i][j][k-1]:
            find_longest_common_subsequence(i, j-1, k, lcs)
            find_longest_common_subsequence(i, j, k-1, lcs)
        else:
            find_longest_common_subsequence(i-1, j, k, lcs)
            find_longest_common_subsequence(i, j-1, k, lcs)
            find_longest_common_subsequence(i, j, k-1, lcs)
    res.append([reverse(x) for x in lcs])
    return res
```

Problem 2

(i)

This question is asked us to derive a necessary condition for a feasible circulation with demands to exist. We know that each node v has an integer demand d_v; if $d_v > 0$, v is a sink and requires that amount of flow. If $d_v < 0$, v is a source and has the ability to provide the flow. If $d_v = 0$, v is neither a sink nor a source. Based on the above arguments, a necessary condition for a feasible circulation with demands to exist is:

$$\sum_{source} -d_v = \sum_{sink} d_v \tag{3}$$

This makes sense because of the conservation of flows. If the circulation is feasible, the flow is not possible to be generated or disappear suddenly. So it may either provided by the source or consumed by the sink. Therefore, the flow provided by the source has to be equal to the flow consumed by the sink to make the circulation feasible.

(ii)

Here to reduce the problem of finding a feasible circulation with demands to max flow, we can construct a new network G' first. Then we will create a super source node *super_source* and a super sink node *super_sink*. The next step is to add new edge between source node with *super_source* node and between sink node with *super_sink* node. We will have:

$$\begin{cases} \text{Every source node } v & \text{adding a new edge (super\_source, v) with capacity } -d_v \\ \text{Every sink node } u & \text{adding a new edge (u, super\_sink) with capacity } d_v \end{cases} \tag{4}$$

The inputs to the two problems

Original problem: $G = (V, E, c, d)$
Reduction: $G' = (V', E', c, d)$, where $V' = V + super\_source + super\_sink$ and $E' = E + (super\_source, v) + (u, super\_sink)$.

The Reduction transformation and the polynomial time requirements

After having the new G' defined, we can run max-flow in G' to find the max flow.

$$\begin{cases} \text{maxflow} = \sum_{sink} d_v & \text{return True} \\ \text{maxflow} \neq \sum_{sink} d_v & \text{return False} \end{cases} \tag{5}$$

The running time required for the reduction problem is $O(n + m)$.
And the time needed to compute the flow is $O(nm)$.

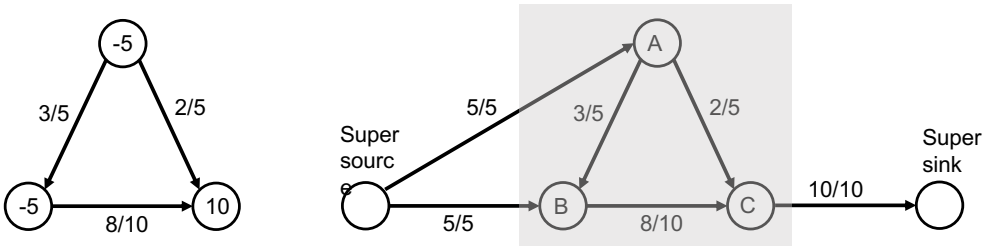

Figure 1: G (left) and G' (right)

Prove equivalence of the original and the reduced instances.

1. From G to G'.

   - As been mentioned before, G' has the super source and super sink nodes. Assume that there is a feasible circulation in G. Therefore, by assigning the capacity of (super_source_node, source) edge to be the demand of source and the capacity of (sink, super_sink_node) edge to be the demand of the sink, we have created a flow f' in G' where the total flow leaves the super source node is equal to the flow enters the super sink node.
   - This setting satisfies the capacity and flow demand constraints.

2. From G' to G.

   - Based on the figure 1, we know that the every edge leaves the super source node and every edge enters the super sink node are full. This means every source and sink connected with super source and super sink node, respectively have their desired demand and supply. So this makes the graph satisfies the flow demand constrains. And obviously, the capacity constraint is also satisfied.

3. The detailed G and G' networks are shown below (figure 1).

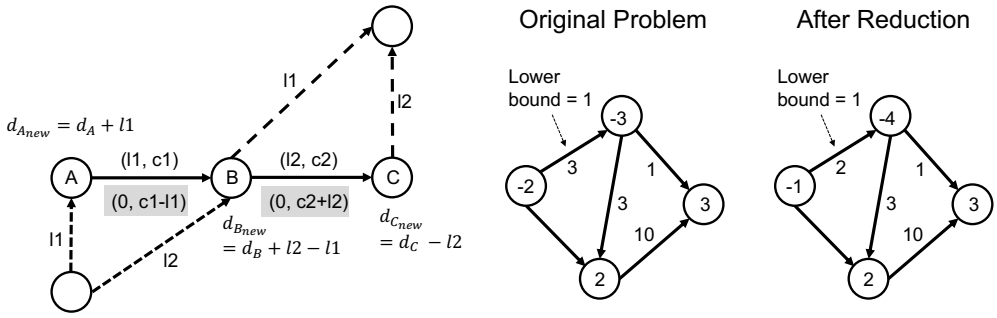

Figure 2: G (left) and G' (right)

Problem 3

In this question, we are asked to determine whether a feasible circulation exists for a flow network with the lower bounds. In order to solve this problem, we can reduce this problem to a standard circulation problem as problem 2 we solved before by introducing initial flow f_0. Basically, the initial flow f_0 will provide the amount of lower limit flow and the additional flow introduced by initial flow f_0 will be added to A's demand and subtracted from B's demand. Please see the detailed network shown in figure 2.

The inputs to the two problems

Original problem: $G = (V, E, c, d, l)$.
Reduction: $G = (V, E, c', d', 0)$.

The Reduction transformation and the polynomial time requirements

Suppose we defined an initial flow f_0 by setting the flow along each edge equal to the lower bound. In other words: $f_0(e) = l_e$.
Lets's define: $L_v = f_0^{in}(v) - f_0^{out}(v)$. Given the circulation flow network G with lower bounds, we can:

1. Subtract l_e form the capacity of each edge e
2. Subtract L_v from the demand of each node v.

We then solve the circulation problem on this new graph G' to get a flow f'. To find the flow that satisfies the original constraints, we add l_e to every $f'(e)$.
The running time required for the reduction problem is $O(n + m)$.
And the time needed to compute the flow is $O(nm)$.

Prove equivalence of the original and the reduced instances.

- New demand constrains: $f^{in}(v) - f^{out}(v) = d_v - L_v$
- New capacity constrains: $0 \leq f(e) \leq c_e - l_e$

1. From G to G'.

   - Assume that G has a feasible circulation, then let $f(G') = f(G) - f_0$. It should be still a valid circulation for G'.

2. From G' to G.

   - Assume that G' has a feasible circulation, then let $f(G') + f_0 = f(G)$. It should be still a valid circulation for G.

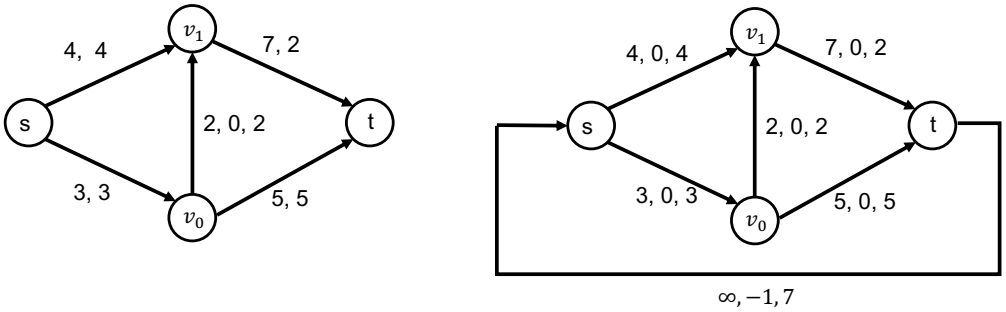

Figure 3: Pure Network flow problem (left) and Network flow problem by formulating a min-cost flow problem (right)

Problem 4

This problem is asked us to show that the max flow problem can be formulated as a min-cost flow problem. We can following the several steps:

1. We set cost of all edges in the network to 0.
2. We then introduce a edge from sink to source (t, s) with cost c(t, s) = 1 and an infinite capacity.
3. Then the minimum cost flow maximizes the flow on the new introduced edge (t, s) since the minimum cost flow will try to visit edge (t, s) as many times as possible because this edge has a negative cost.
4. However, in order to have the flow that can be traveled from t to s, some amount of flows must travel from s to t.
5. In order to minimize the cost, we will maximize the flow from s to t in the original network.
6. Please refer to figure 3 which shows a better demonstration.

Appendix

References

- Problem 3: https://www.cs.cmu.edu/ ckingsf/bioinfo-lectures/flowext.pdf

Useful discussions with

- Hongyi Hue (hh2955)