CS345: Algorithms - II Due: 11/11/19

Assignment #4

Name: **Ayush Hitesh Soneria** Instructor: Prof. Nitin Saxena

Roll-No: **170192** TA: Nikhil

Email: ayushhs@iitk.ac.in OR ayushhs@cse.iitk.ac.in TA's Email: nshagri@cse.iitk.ac.in

Max Points: 50

Question 1: Longest monotonically increasing sub-sequence [5 points]

Pseudo-code

```
   Data: num ← input sequence of numbers in form of array
   Data: n ← input array size
 1 Function inc-sub-sequence(num,n):                  /* algorithm as a iterative function */
 2 |  initialize ans[n]; initialize temp[n];
 3 |  ans[0] ← 1; temp[0] ← 0;
 4 |  ref-best = 0;
 5 |  for (i = 1 to n-1) do
 6 |  |  ans[i] ← 1;
 7 |  |  temp[i] ← 0;
 8 |  |  for (j = 0 to i-1) do
 9 |  |  |  if ((num[i] > num[j]) and (ans[i] < ans[j]+1)) then
10 |  |  |  |  ans[i] = ans[j]+1;
11 |  |  |  |  temp[i] = j;
12 |  |  |  end
13 |  |  end
14 |  |  if (ans[i] > ans[ref-best]) then
15 |  |  |  ref-best = i;
16 |  |  end
17 |  end
18 |  initialize out[n];
19 |  while ref-best != 0 do
20 |  |  out.append(num[ref-best]);
21 |  |  ref-best = temp[ref-best];
22 |  end
23 |  return out.reverse;                                /* needed output of numbers */
24 End Function
```

Algorithm 1: Longest monotonically increasing sub-sequence as Iterative function in $O(n^2)$ time

Proof of correctness & Analysis

ans[i] will store length of longest increasing sub-sequence ending at index i.
temp[i] stores a back-reference for longest increasing sub-sequence ending with i.
out.reverse[i] will contain the i^{th} element of the resultant Longest monotonically increasing sub-sequence.
ref-best will store the index, say j, such that ans[j] is greater than any ans[k] for all k less than j

Lines 5-17 will take $O(n^2)$ time due to the two nested for loops, actually the number of iteration = 1+2+3...+n-1 which is $O(n^2)$ The while loop from 19-22 and lines 23 (reverse function) both take O(n) time. Hence Overall time complexity to determine Longest monotonically increasing sub-sequence is $O(n^2)$.

Question 2: Optimal Triangulation of convex polygon [8 points]

Pseudo-code

```
   Data: n ← number of points in convex polygon P
   Data: w ← weight function of triplet of points
 1 Function Optimal-Triangulation(1,n,w):          /* algorithm as a iterative function */
 2 |  initialize 2D array, array[n,n];
 3 |  for (s = 1 to n - 1) do
 4 |  |  array[s,s + 1] ← 0;
 5 |  end
 6 |  for (h = 2 to n - 1) do
 7 |  |  for (i = 1 to n - h) do
 8 |  |  |  j ← i + h;
 9 |  |  |  array[i,j] ← ∞;
10 |  |  |  for (k = i + 1 to j - 1) do
11 |  |  |  |  temp ← array[i,k] + array[k,j] + w(i,k,j);
12 |  |  |  |  if (array[i,j] > temp) then
13 |  |  |  |  |  array[i,j] ← temp;
14 |  |  |  |  end
15 |  |  |  end
16 |  |  end
17 |  end
18 |  return array[1,n];
19 End Function
```

Algorithm 2: Optimal-Triangulation as Iterative function in $O(n^3)$ time

Proof of correctness & Analysis

There are 3 for loops. They are in a nested manner. Lines 10-12 take O(1) time. Lines 7-8 also takes O(1) time. Hence the time taken is

$$<= n * [n * \{(n * O(1)) + O(1)\}]$$

which gives us time complexity as $O(n^3)$ or cubic time complexity

Line 10 in the pseudo code represents the optimal sub structure property used in our bottom up approach to solve this problem.

Question 3: Minimum length of negative-weight cycle [10 points]

Pseudo-code

```
   Data: A ← Input Adjacency List
   Data: n ← number of vertices
 1 Function Shortest-Paths-Matrix(A,n):              /* algorithm as a iterative function */
 2 |  convert given adjacency list, A, into adjacency matrix, M;
 3 |  L(1) = M;
 4 |  for (x = 2 to n - 1) do
 5 |  |  L(x) ← new n * n matrix;
 6 |  |  L(x) ← Helper(L(x-1), M,n);                          /* function defined below */
 7 |  |  if at least one of the diagonal elements (L(x)[i,i], where i ∈ [0,n-1]) is negative then
 8 |  |  |  return x;          /* output needed, x = minimum length of negative-weight cycle */
 9 |  |  end
10 |  end
11 |  return "No negative cycles";
12 End Function
```

Algorithm 3: Shortest-Paths-Matrix as Iterative function in $O(n^4)$ time

```
   Data: n ← number of vertices
   Data: L, M ← two n * n matrices
 1 Function Helper(L,M,n):                           /* algorithm as a iterative function */
 2 |  T ← new n * n matrix;
 3 |  for (i = 1 to n) do
 4 |  |  for (j = 1 to n) do
 5 |  |  |  T[i,j] ← ∞;
 6 |  |  |  for (k = i to n) do
 7 |  |  |  |  T[i,j] ← min[ T[i,j] , ( L(i,k) + M(k,j) )]
 8 |  |  |  end
 9 |  |  end
10 |  end
11 |  return T;
12 End Function
```

Algorithm 4: Helper function as Iterative function in $O(n^3)$ time

Analysis & Proof of Correctness

The helper function has time complexity $O(n^3)$ due to the 3 nested for loops each having O(n) iterations. The shortest-paths-matrix function has time complexity $O(n^4)$ because it calls the helper function at most n times and the conversion of adjacency list to adjacency matrix is negligible compared to $O(n^4)$ time complexity.

The helper function called in Shortest-Pairs-Matrix, Helper(L(x-1),M,n), calculates the following: a matrix T, whose elements, some T[i,j] represent minimum weight of any path from vertex i to vertex j that contains at most x-1 edges.

A cycle in the graph is represented by the diagonal T[i,i] in the matrix as start vertex = end vertex. We observe that if the graph contains a negative-weight cycle, then there is a negative-weight path of length k <= n from some vertex i to i. Therefore we can detect the existence of negative cycles simply by checking whether there are negative values on the diagonal of the matrices obtained at each iteration of our shortest-paths-matrix function. Lines 7-9 in shortest-paths-matrix function return the minimum number of edges as needed because value of x increases from 2 to n.

Hence the efficient algorithm (Shortest-Pairs-Matrix) to find the length of a negative-weight cycle, in G, that uses the minimum number of edges takes $O(n^4)$ time.

Question 4: k-edge-connected undirected graph [11 points]

We will define our flow network as (G,u,v,c) where u,v *in* V and c(a,b) = 1 for all (a,b) ∈ E. We want all capacities to be 1 so that the number of edges crossing a cut equals the capacity of the cut. Let f(u,v) be maximum flow in the network (G,u,v,c). Also let us take a source node s, where s ∈ V

Pseudo-code: in order to determine k, which is the edge-connectivity

```
   Data: G ← input undirected graph with edge set E and vertex set V
   Data: s ← source node, ∈ V (choice does not matter)
1  Function edge-connectivity(G,s):                    /* algorithm as a iterative function */
2  |  c(a,b) = 1; for all (a,b) ∈ E
3  |  for each (v ∈ V-{s}) do
4  |  |  compute max flow f(s,v) on network (G,s,v,c);
5  |  end
6  |  return minimum from all |f(s,v)| where v ∈ V-{s};   /* minimum max flow from all max flows */
7  End Function
```

Algorithm 5: determining k, edge-connectivity, as Iterative function in $O(|E| * |V|^2)$ time

Analysis

Line 4 will be done using the Ford-Fulkerson algorithm. We know that the Ford-Fulkerson algorithm finds each maximum flow in time $O(|E| * |V|)$. Since each network can have a maximum flow of cost at most $|V| - 1$, and all capacities are 1, so the algorithm uses $|V| - 1$ maximum flow computations.
Hence the overall running time is $O(|E| * |V|^2)$

Proof of Correctness

Claim(as implemented in algorithm): edge connectivity, k = minimum of all $|f(s,v)|$ where v ∈ V-{s}.
In other words k = minimum of all max flows in the network.

Proof:

Let i = minimum of all $|f(s,v)|$ where v ∈ V-{s}. Let us take the case where i-1 edges are removed from G. For any vertex v ∈ V-{s}, max flow from s to v is at least i, so any cut which separates s and v will have capacity i, which means at least i edges cross any such cut due to unit capacities. Thus even after removing i-1 edges, one edge cross the cut. Using max-flow min-cut theorem, we can say that s and v are still connected. And hence every node is connected to s which means that the graph is still connected even after removing i-1 edges. Hence in order for the graph to be disconnected, at least i edges must be removed or in other words minimum number of edges to be removed is i (which is the edge connectivity) for graph to be disconnected.
Hence k = i, or in other words k = minimum of all $|f(s,v)|$ where v ∈ V-{s}. Hence proved.

Question 5: Perfect Matching [16 points]

Claim: G=(L,R,E) is a bipartite graph and has a perfect matching iff for all A $\subseteq$ L, $|A| <= |N(A)|$.

Proof

Due to the iff(if and only if) there will be two directions to prove here.

Claim(i) if there is a perfect matching in G then for all A $\subseteq$ L, $|A| <= |N(A)|$

Proof(i):

This proof is a bit trivial, as we can clearly see that in perfect matching each vertex in A is mapped to a distinct vertex in N(A), so $|N(A)|$ is at least $|A|$ meaning $|A| <= |N(A)|$. Hence proved claim(i).

Claim(ii) if for all A $\subseteq$ L, $|A| <= |N(A)|$ then G has a perfect matching.

Proof(ii):

Let us prove the contrapositive of the Claim(ii), which will be if G does not have a perfect matching then there exists A $\subseteq$ L such that $|A| > |N(A)|$.
We will try and use the max flow - min cut theorem here to prove the above statement.

Let us construct a network X=($(V^{'}, E'$),s,t,c) as follows:
1. $V^{'}$ = V $\cup$ {s,t} where s,t are two new vertices.
2. $E^{'}$ will contain (i) directed edges (s,u) for all u $\in$ L, (ii) directed edges (u,v) for all edges (u,v) $\in$ E where u $\in$ L and v $\in$ R, (iii) directed edges (v,t) for all v $\in$ R.
3. all edges have capacity, c(edge), equal to 1.

Let n = $|L|$. If G does not have a perfect matching then maximum size of any matching in G can be at most n-1. Hence the maximum flow in X can be at most n-1. Which tells us that there is a cut in X, call it P, such that capacity of P is at most n-1, or capacity(P) <= n-1 — (1).

Let L1 = P $\cap$ L, L2 = L – S, R1 = P $\cap$ R, R2 = R – S.

We know all edges have capacities 1 in network X, so capacity of cut P is number of edges that go from P to complement of P, therefore capacity(P) = $|L2| + |R1|$+ [# of edges from L1 to R2] — (2)

combining (1) and (2) we get, n – 1 >= $|L2| + |R1|$+ [# of edges from L1 to R2]
Using the fact that $|L1| + |L2|$ = n, we will get $|L1| >= |R1|$+ [# of edges from L1 to R2] + 1 — (3)

Now we come across a important observation, that the neighborhood of L1 will contain all of R1 but not all of R2. To be precise N(L1) can at most have all vertices of R1 and vertices in R2 who have incoming edges from L1 (edges which go from L1 to R2). Hence, $|N(L1)| >= |R1|$+ [# of edges from L1 to R2] — (4)

Combining (3) and (4) we get, $|L1| >= |N(L1)| + 1$, which is $|L1| > |N(L1)|$.

Hence we found A(=L1) $\subseteq$ L such that $|A| > |N(A)|$ by using the condition that G does not have a perfect matching. Hence we proved the contrapositive of the needed claim(ii). Hence we have indirectly proved the claim(ii) we wanted.

Now we have proved both Claim(i) and Claim(ii), so it suffices to say we have proved the original Claim.

Hence Proved.