/

G 8/14

CS 1510
Algorithm Design

Greedy Algorithms & Dynamic Programming

Problems 16, 2, and 3

Due Wednesday September 10, 2014

Buck Young and Rob Brown

32/48

Problem 16

(a)

Input: A linear graph of n nodes (ie, a linked list of length n) and set of k packets $p_1, p_2, ..., p_k$ where each packet $p_i = (A, r_p, s_p, t_p)$ where A is the packet name, r_p is the packet release time, s_p is the packet start node, and t_p is the packet's target node.

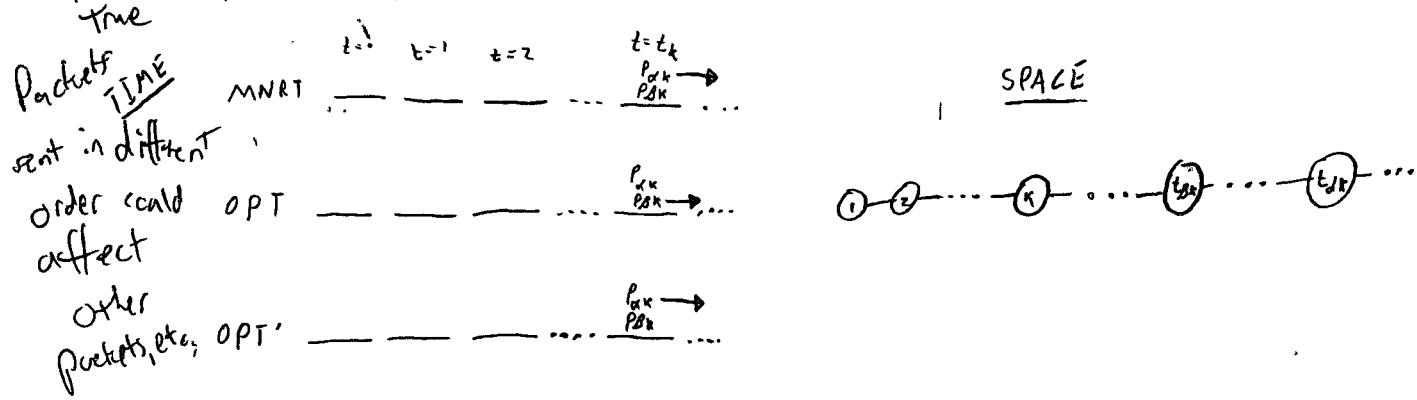
Output: The order of "sent" packets for each of the n nodes such that the maximum time that a packet takes to leave the network is minimized (ie, the network is cleared of all packets as quickly as possible).

Algorithm: For for the i^{th} node, take the the j^{th} packet from its packet-list such that the number of remaining nodes $NR_j = t_{pj} - i$ is the maximum of all packets in the node's packet-list. Put p_j in i's "sent" list for this time-step, and move p_j to node $i+1$.

Proof: The given "Most Remaining Nodes First" (MRNF) algorithm is correct.

Suppose for the sake of reaching a contradiction that there exists some input I on which MNRF produces unacceptable output. Let MNRF(I) be the output of MNRF on this input, and let OPT(I) be the output of some optimal algorithm which agrees with MNRF for the most number of steps.

Consider a given point where the output of MNRF disagrees with that of OPT. We define a "disagreement" to be any point where OPT sends a packet from a given node that violates the "Most Remaining Nodes First" principle. Note, however, that the only important differences are those which occur on the last packet to leave the network (in time), since we are concerned solely with the minimization of this value. Thus, consider this point to be the first disagreement and label this the k^{th} step.

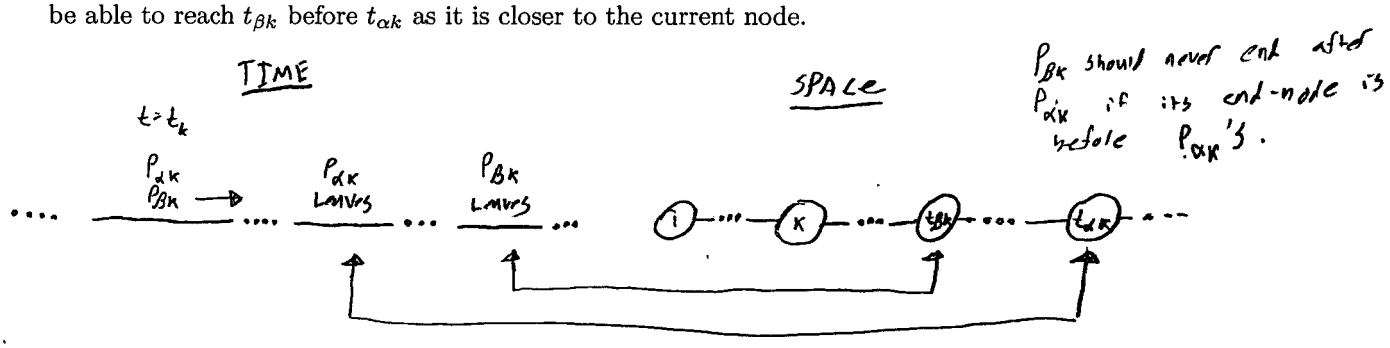


Let OPT' be some algorithm such that its output is that of OPT, but where $p_{\alpha k}$ is sent at the point of disagreement instead of $p_{\beta k}$.

Note that if neither $p_{\beta k}$ or $p_{\alpha k}$ are the last packets to leave the network in the schedule produced by OPT, then this single modification can be made without repercussion.

Now consider the more interesting cases where one of these two packets is the final packet to leave the network according to the schedule of OPT. If $p_{\alpha k} \in OPT$ is the last packet to leave the network, then we have that $MNRF \leq OPT$ since $p_{\alpha k} \in MNRF$ is defined to be the last packet to leave the system (ie, both OPT and MNRF have $p_{\alpha k}$ leaving the system last). OPT choosing to hold this packet up can only have detrimental effects on its time in the system. Thus $MNRF \leq OPT$.

In the case where $p_{\beta k} \in OPT$ is the last packet out the schedule produced by OPT, we have the following two diagrams of the system (one for space, one for time). There are $t_{\alpha k} - t_{\beta k}$ nodes between the exit node of $p_{\alpha k}$ and $p_{\beta k}$, and we know that the exit node of $p_{\alpha k}$ comes later later in the network than that of $p_{\beta k}$ by the definition of $MNRF$. We also know that $p_{\beta k}$ takes longer to leave the network than $p_{\alpha k}$ (and in fact takes the longest time to leave the network). This should never be, since an optimal algorithm would be able to reach $t_{\beta k}$ before $t_{\alpha k}$ as it is closer to the current node.



So in this case $OPT' \geq OPT$ and we can safely make our exchange (or simply raise a contradiction on the definition of OPT).

$\therefore$ for all covered cases $OPT \leq OPT'$ and OPT' is clearly more similar to $MNRF$ than OPT $\square$

Problem 16

(b)

For this problem, a first-in first-out (FIFO) algorithm would be optimal. That is, we can queue up the packets as they are released and move them along the edge in the order they are queued. Ties would be broken arbitrarily.

Problem 2

The following iterative, array-based, bottom-up algorithm will return the longest sequence S that is a subsequence of three strings A, B, and C in polynomial (n^3) time.

```
string x, y, z, result = ""
string LCS[A.length][B.length][C.length] = "" // Initialize to empty strings

// Traverse from 1 to size length (leave all a=0, b=0, c=0 as empty strings)
for a = 1 to A.length:
    for b = 1 to B.length:
        for c = 1 to C.length:

            if A[a-1] == B[b-1] == C[c-1]: // Access the strings from 0 to length-1
                LCS[a][b][c] = LCS[a-1][b-1][c-1] + A[a-1] // Add the character to the solution

                if LCS[a][b][c].length > result.length: // Store iff longest substring
                    result = LCS[a][b][c]
            else:
                x = LCS[a-1][b][c]
                y = LCS[a][b-1][c]
                z = LCS[a][b][c-1]

                // Find the maximum substring amongst x, y, and z and add it to the solution
                if x.length > y.length && x.length > z.length:
                    LCS[a][b][c] = x
                elif y.length > x.length && y.length > z.length:
                    LCS[a][b][c] = y
                else:
                    LCS[a][b][c] = z // All equal or z.length is greatest

return result
```

Problem 3

The following iterative, array-based, bottom-up algorithm will compute a table for finding the shortest common super-sequence of two strings A and B.

```
SCSS[A.length][B.length] = min(A.length, B.length) // Initialize to shortest string size

for a = 1 to A.length:
    for b = 1 to B.length:
        if A[a-1] == B[b-1]: // Access strings from 0 to length-1
            SCSS[a][b] = SCSS[a-1][b-1] - 1
        else:
            SCSS[a][b] = min( SCSS[a-1][b], SCSS[a][b-1] )
```

(a)

		Z	X	Y	Y	Z	Z
	6	6	6	6	6	6	6
Z	6	5	5	5	5	5	5
Z	6	5	5	5	5	4	4
Y	6	5	5	4	4	4	4
X	6	5	4	4	4	4	4
Z	6	5	4	4	4	3	3
Y	6	5	4	3	3	3	3

(b) The length of the SCSS will be the smallest number in the table plus min(A.length, B.length).

(c) Unfortunately, the created algorithm doesn't seem to be correct – as a backwards trace is confusing and inconsistent. However, we were hoping for something along the lines of: when the letters are unequal, travel towards min(vertical up, horizontal left) and write the letter depending on the direction / when the letters are equal travel diagonally and write the letter.