CMSC 641, Design and Analysis of Algorithms, Spring 2010

Sandipan Dey,
Homework Assignment - 1

February 16, 2010

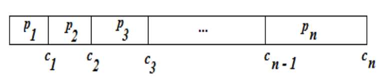
Problem 1 Solution

Algorithm 1 Greedy algorithm to minimize the average completion time of tasks

---

1: sort tasks (a_i) in the increasing (non-decreasing) order of their processing times (p_i).
2: schedule the tasks from the sorted list in that order (according to increasing p_i, i.e., jobs with shorter processing time to be scheduled first).

Let $\{p_1, p_2, \ldots, p_n\}$ be n tasks sorted in monotonically increasing order (non-decreasing order, in case of ties, by resolving ties in arbitrary manner) of their processing time ($p_i \leq p_j$ if $i < j$), then the greedy algorithm (G) schedules the tasks $a_1, a_2, \ldots a_n$ in that order, with completion time for the i^{th} task $= c_i = \displaystyle\sum_{j=1}^{i} p_j$.



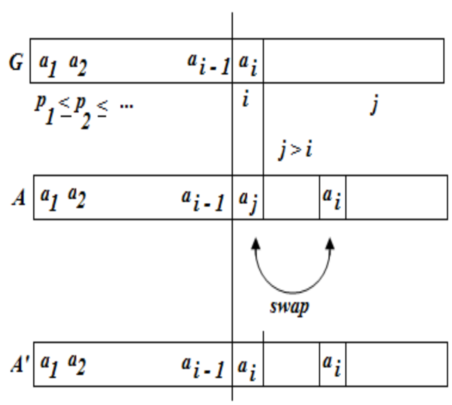
Now, average completion time $= \bar{c} = \frac{1}{n}\displaystyle\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{i} p_j$

$$= \frac{1}{n}\left(p_1 + (p_1 + p_2) + (p_1 + p_2 + p_3) + \ldots + (p_1 + p_2 + p_3 + \ldots + p_n)\right)$$
$$= \frac{1}{n}\left(n.p_1 + (n-1).p_2 + (n-2).p_3 + \ldots + 1.p_n\right) = \frac{1}{n}\sum_{i=1}^{n}(n-i+1).p_i.$$

Claim: Greedy minimizes the average completion time

Proof (by iterative transformation and contradiction)

It is obvious that G minimizes the completion time locally (by choosing the shortest time task from the remaining tasks at each scheduling step), but let's assume to the contrary that G does not (globally) minimize $\bar{c} \Rightarrow \exists$ another algorithm A that performs strictly better than $G \Rightarrow (\bar{c})_A < (\bar{c})_G$. Let's further assume (without loss of generality) that a_i is the first task that is differently scheduled by G and A, all the tasks $a_1, a_2, \ldots a_{i-1}$ are scheduled in the same order by both the algorithms. If algorithm A schedules a_j instead of a_i ($j > i$ by earlier assumption, s.t. $p_j \geq p_i$), by swapping a_i and a_j the average completion time changes by $(n-i+1)p_i + (n-j+1)p_j - (n-j+1)p_i - (n-i+1)p_j = (j-i)(p_i - p_j) \leq 0$, i.e., does not increase (either decreases or remains same). We can repeat the earlier step with another algorithm $A\prime$ that still does not increase the average completion time, by choosing the next differently scheduled task from A. We can go on getting a non-increasing chain of average completions times by a series of algorithms, but since the number of tasks n is finite and at each iteration the total difference between scheduling by the successive algorithms and with algorithm G keeps on decreasing, ultimately we get back G in this iterative process $\Rightarrow (\bar{c})_G > (\bar{c})_A \geq (\bar{c})_{A\prime} \geq (\bar{c})_{A\prime\prime} \geq \ldots \geq (\bar{c})_G$, hence a contradition, our initial assumption was wrong.



$(n - i + 1)\, p$

$$\bar{c}_{A'} - \bar{c}_A = (n-i+1)\,p_i + (n-j+1)\,p_j - (n-i+1)\,p_j - (n-j+1)\,p_i$$

$$= (j-i)\,(p_i - p_j) \leq 0 \Rightarrow \bar{c}_{A'} \leq \bar{c}_A$$

Runtime

Step 1 takes $\theta(nlogn)$ time to sort. Step 2 takes $\theta(n)$ time to schedule, where n is the total number of tasks. Hence, total runtime $= \theta(nlogn)$.

Problem 2 Solution

a) First let's define the following:

S = seqence of characters $s_1, s_2, \ldots s_n$ (as string).
$S_{ij} = substr(S, i, j)$ = contiguous seqence of characters $s_i, s_{i+1}, \ldots s_j$ of S, with $i \leq j$ (as substring of S).

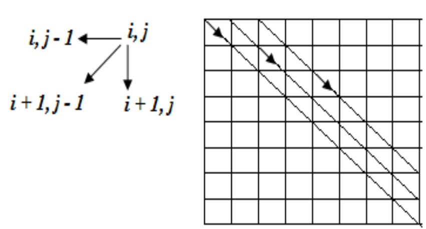
$LPSL(i, j)$ = Length of the longest palindrome subsequence in the substring S_{ij} of the string S.

Now, we can recursively define the $LPSL$ function as:

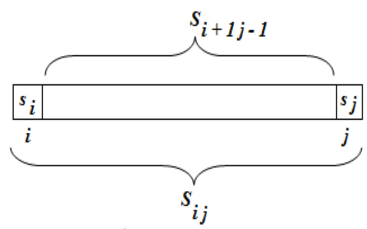
$$LPSL(i,j) = \left\{ \begin{array}{ll} 0 & \text{if } i > j \ (1) \\ 1 & \text{if } i = j \ (2) \\ 2 + LPSL(i+1, j-1) & \text{if } s_i = s_j \ (3) \\ max\{LPSL(i, j-1), LPSL(i+1, j)\} & \text{if } s_i \neq s_j \ (4) \end{array} \right\}$$

By the usual cut and paste argument, the optimal substructure property can be proved, i.e., if there exists a palindrome subsequence longer than that for any of the smaller subproblems, then cut/paste will result in a still longer palindrome subsequence for the original problem, a contradiction.

b) As can be seen, if we construct table for storing $LPSL$ and reuse optimal solutions to the subproblems already computed to solve the problem, finding the optimal solution from the optimal subproblem solutions can be done in contant time for every such problem (since all the steps $1, 2, 3, 4$ are $O(1)$). Hence, the runtime of the dynamic programming will be $= \theta(n^2)$, where n is the length of the input string, since we have to solve $\theta(n^2)$ subproblems, each takes $O(1)$ time.



LPSL Matrix (upper triangular)
Diagonal fill-up



$$LPS(S_{ij}) = \begin{cases} s_i & \text{if } i = j \\ s_i + LPS(S_{i+1\,j-1}) + s_j & \text{if } s_i = s_j \\ LPS(S_{i+1\,j}) & \text{if } |LPS(S_{i+1\,j})| \geq |LPS(S_{i\,j-1})| \\ LPS(S_{i\,j-1}) & \text{otherwise} \end{cases}$$

$$LPS('abcca') = 'a' + LPS('bcc') + 'a' = 'a' + LPS('cc') + 'a' = 'acca'$$

To meet the dependency requirement, one way will be to initialize the table diagonally as shown. Note that the elements below the principal diagonal are never used, hence the table takes $\frac{n(n+1)}{2} = \theta(n^2)$ space. We use memoization technique and use another table b to find the longest palindrome sequence. The runtime of the algorithm is $\theta(n^2)$, as discussed earlier.

In order to find the longest palindrome subsequence the memoized array is used in the recursive algorithm PRINT-PALINDROME.

Call PRINT-PALINDROME(b, S, 1, n) to find the longest palindrome subsequence in S. The complexity is $\theta(n)$, since for every recursive invocation, the length of the string to be considered ($|j - i|$) dereases by at least 1. Hence the total runtime of LPS-LENGTH and PRINT-PALINDROME is $\theta(n^2)$.

Alternative approach

Alternatively, the problem of finding the longest palindrome subsequence can be formulated using (can be reduced to) the longest common subsequence problem. The problem basically reduces to the longest common subsequence between a string and its reverse string.

Reversing a string takes $\theta(n)$ time, where $n = length(S)$. Then $LCS(S, S^R)$ takes $\theta(n.n)$ time. Hence, Runtime $= \theta(n) + \theta(n^2) = \theta(n^2)$.

Output of sample implementation is shown in figure-1.

Algorithm 2 LPS-LENGTH: Finds longest palindrome subsequence length

LPS-LENGTH(S)

$n \leftarrow length[S]$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to $i-1$ **do**
    $LPSL[i,j] \leftarrow 0$
  end for
end for
for $i \leftarrow 1$ to n **do**
  $LPSL[i,i] \leftarrow 1$ {set $LPSL[i,i] \leftarrow 0$ to find the longest even-length palindrome sequence}
end for
for $k \leftarrow 1$ to n **do**
  for $j \leftarrow i+k$ to n **do**
    if $s\_i = s\_j$ **then**
      $LPSL[i,j] \leftarrow LPSL[i+1,j-1]+2$
      $b[i,j] \leftarrow$ '$\nwarrow$'
    else if $LPSL[i+1,j] \geq LPSL[i,j-1]$ **then**
      $LPSL[i,j] \leftarrow LPSL[i+1,j]$
      $b[i,j] \leftarrow$ '$\rightarrow$'
    else if $LPSL[i+1,j] < LPSL[i,j-1]$ **then**
      $LPSL[i,j] \leftarrow LPSL[i,j-1]$
      $b[i,j] \leftarrow$ '$\leftarrow$'
    end if
    return LPSL and b
  end for
end for

Algorithm 3 PRINT-PALINDROME: Finds the longest palindrome subsequence

PRINT-PALINDROME(b, S, i, j)

if $i > j$ **then**
  return
end if
if $b[i,j] =$ '$\nwarrow$' **then** {$s\_i = s\_j$}
  print(s_i)
  PRINT-PALINDROME(b, S, i + 1, j - 1)
  print(s_i)
else if $b[i,j] =$ '$\rightarrow$' **then**
  PRINT-PALINDROME(b, S, i + 1, j)
else
  PRINT-PALINDROME(b, S, i, j - 1)
end if

Algorithm 4 LPS: Finds the longest palindrome subsequence

LPS(S)

1: $S^R \leftarrow reverse\_string(S)$ {reverse the string and store in S^R, in $\theta(length(S))$ time}
2: $LCS(S, S^R)$ {find and print the longest common subsequence of S and S^R}

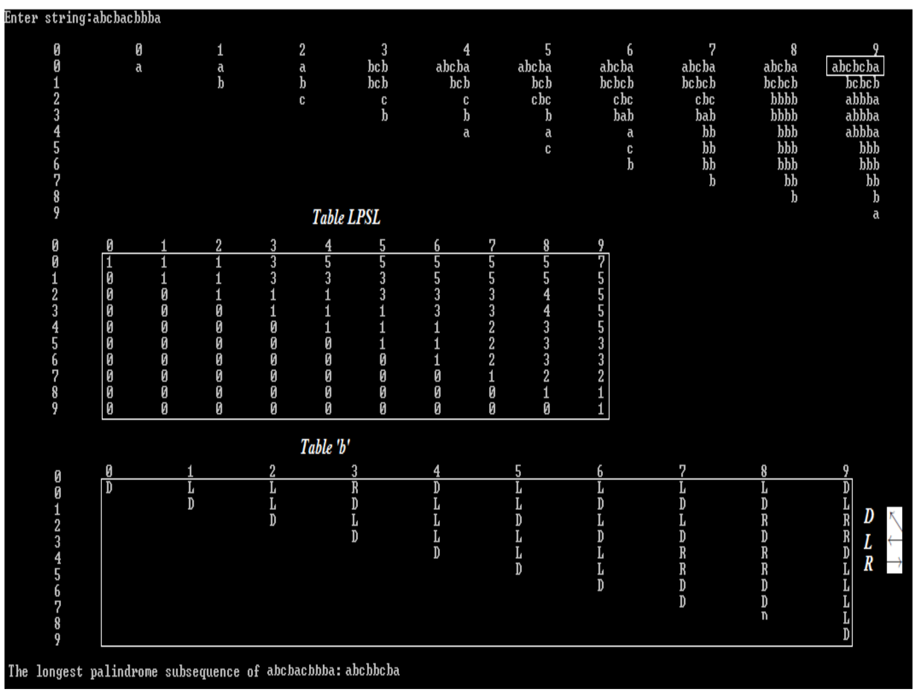


Figure 1: Output of sample implementation for the longest palindrome subsequence

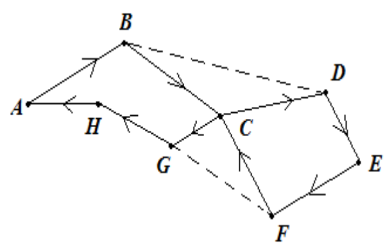
Problem 3 Solution

We claim the following:

The shortest bitonic tour must be a disjoint (can only have common start and end points, but strictly no common points in between) union of exactly 2 paths:

- one strictly going from left to right ($\rightarrow$).
- the other one strictly going from right to left ($\leftarrow$).

Assuming to the contrary, if they are not disjoint, a shorter bitonic tour can always be found by applying triangle inequality, a contradiction, as shown in the following figure.



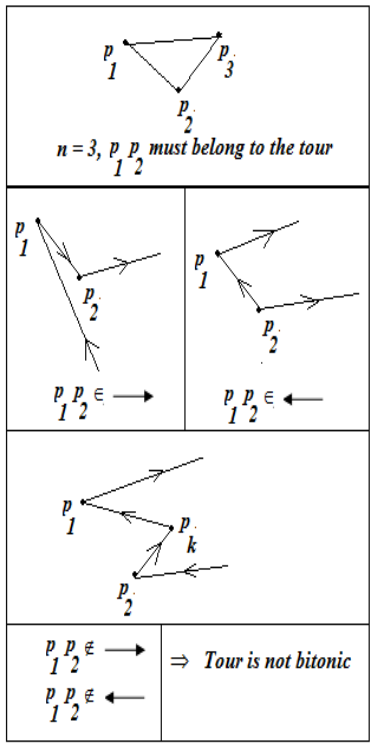
Bitonic tour (A-B-C-D-E-F-C-G-H-A)	$\longrightarrow$ ABCDE $\longleftarrow$ EFCGHA $\rightarrow, \leftarrow$ non-disjoint

Triangle Inequality $\Rightarrow FC + CG > FG$ $\Rightarrow$ length (ABCDEFGHA) < length (ABCDEFCGHA)	
Bitonic tour (A-B-C-D-E-F-G-H-A)	$\longrightarrow$ ABCDE $\longleftarrow$ EFGHA $\rightarrow, \leftarrow$ disjoint

Triangle Inequality $\Rightarrow BC + CD > BD$ $\Rightarrow$ length(ABDEFCGHA) < length (ABCDEFCGHA)	
Bitonic tour (A-B-D-E-F-C-G-H-A)	$\longrightarrow$ ABDE $\longleftarrow$ EFCGHA $\rightarrow, \leftarrow$ disjoint

Also we assume that all the points have different x coordinates. First the points are sorted w.r.t. their x-coordinates and let us represent the sorted list of n points by $p_1, p_2, \ldots, p_n$.

We claim that the edge p_1p_2 must be part of any bitonic tour. As seen from the following figure it is obvious that p_1p_2 must belong to the tour when $n = 3$. For general case, it can be seen from the figure that p_1p_2 must belong either to the $\rightarrow$ or the $\leftarrow$ path, otherwise the tour no longer remains bitonic. Being part of a cycle, there must be two disjoint paths from p_1 to p_2. If the edge p_1p_2 is not part of the cycle, then there must be 2 disjoint bitonic tours from p_1 to p_2 for all $n > 3$ in order to complete the cycle, as obvious from the figure. But we want a single bitonic tour, hence the edge p_1p_2 must be part of the bitonic tour. Also, quite obviously, the rightmost point p_n must be the end of the $\rightarrow$ path and beginning of the $\leftarrow$ path.



Now, let's define the following:

$LBP(i, j)$ =Length of the Least bitonic path starting from p_i and ending in p_j, covering all the points $p_i, p_{i+1}, \ldots, p_n$, with $i < j$.

e.g., $LBP(1, 2)$ will represent the length of the least bitonic tour staring at point p_1 and ending at point p_2 covering all the points $p_1, p_2, \ldots, p_n$. We are interested to find $LBP(1, 1)$. Precisely, there will be a $\rightarrow$ path that will start from p_i and end at p_n, where there will be a $\leftarrow$ path that will start from p_n and end at p_j.

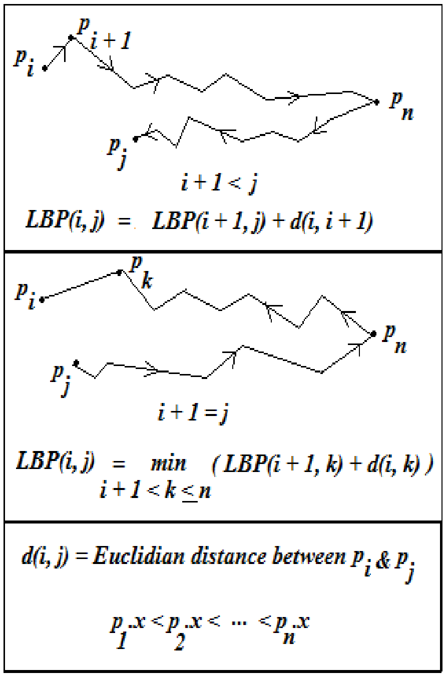
The problem can be recursively formulated as follows:

$$LBP(i,j) = \left\{ \begin{array}{ll} LBP(i+1,j) + d(i,i+1) & \text{if } i+1 < j \\ \min\limits_{i+1<k\leq n} \{LBP(i+1,k) + d(i,k)\} & \text{if } i+1 = j \end{array} \right\}$$

$$LBP(n-1,n) = d(n-1,n).$$

$$LBP(1,1) = LBP(1,2) + d(1,2).$$

Here $d(i,j)$ represents the Euclidian distance between points p_i and p_j.



As can be seen, LBP-LENGTH is $\theta(n^2)$ (including sorting time $\theta(nlogn)$) while PRINT-TSP is $\theta(n)$.

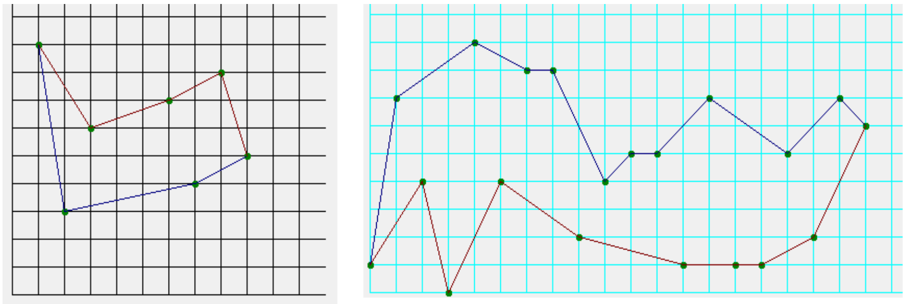


Figure 2: Output of sample implementation for the bitonic TSP

Algorithm 5 LBP-LENGTH: Finds the least bitonic TSP tour length

LBP-LENGTH(P: set of points)

Sort the points in P according to their x axis coordinates, with sorted list $\{p_1, p_2, \ldots, p_n\}$.
$LBP[n-1, n] \leftarrow d(n-1, n)$.
$path[n-1, n] \leftarrow n$.
for $i \leftarrow n-2$ to 1 **do**
    $min \leftarrow \infty$.
    for $k \leftarrow i+2$ to n **do**
        if $min > LBP[i+1, k] + d(i, k)$ **then**
            $min \leftarrow LBP[i+1, k] + d(i, k)$.
            $mink \leftarrow k$.
        end if
    end for
    $LBP[i, i+1] \leftarrow min$.
    $path[i, i+1] \leftarrow mink$.
    for $j = i+2$ to n **do**
        $LBP[i, j] \leftarrow LBP[i+1, j] + d(i, i+1)$.
        $path[i, j] \leftarrow i+1$.
    end for
end for
$LBP[1, 1] \leftarrow LBP[1, 2] + d(1, 2)$.
$path[1, 1] \leftarrow 2$.

Algorithm 6 PRINT-TSP: Finds the least TSP tour

PRINT-TSP(path, i, j, n)

if $n \leq 0$ **then**
    return.
end if
if $i \leq j$ **then**
    $k \leftarrow path[i, j]$.
    print(k).
    PRINT-TSP(path, k, j, n - 1).
else
    $k \leftarrow path[j, i]$.
    PRINT-TSP(path, i, k, n - 1).
    print(k).
end if

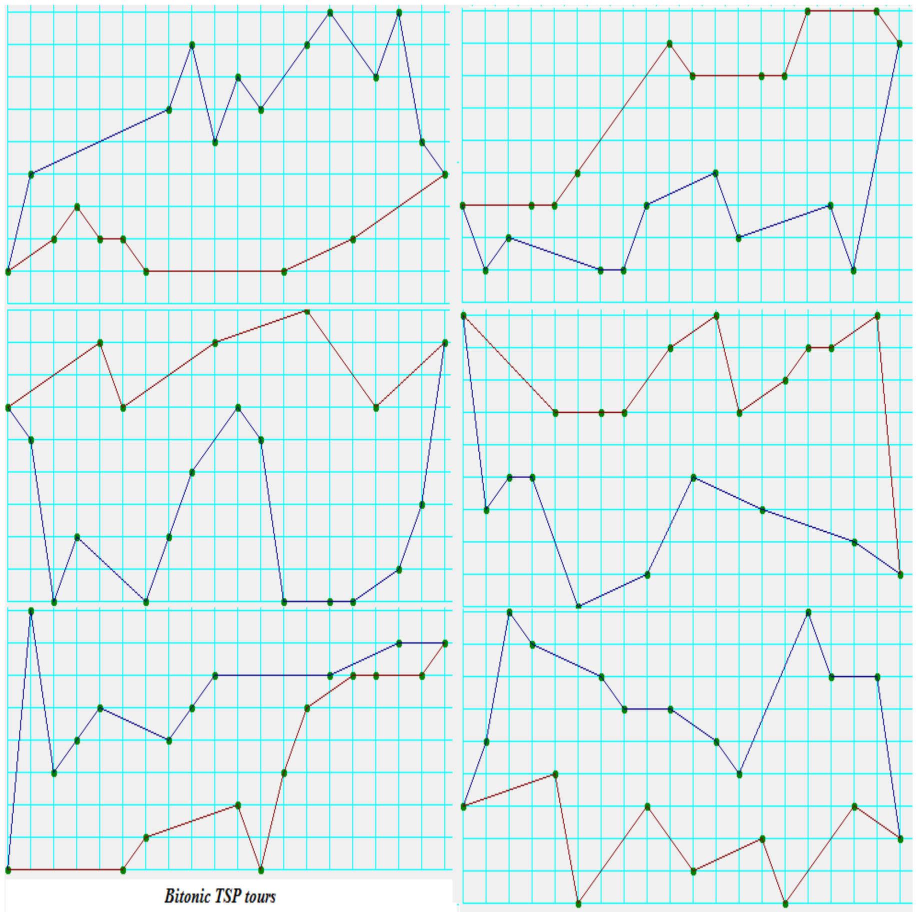


Figure 3: Output of sample implementation for the bitonic TSP