Solutions to Homework 6

Debasish Das
EECS Department, Northwestern University
ddas@northwestern.edu

1 Problem 5.24

We want to find light spanning trees with certain special properties. Given is one example of light spanning tree.
Input: Undirected graph G = (V,E); edge weights w_e; subset of vertices U $\subset$ V
Output: The lightest spanning tree in which the nodes of U are leaves (there might be other leaves in this tree as well)
Consider the minimum spanning tree T = (V,$\hat{E}$) of G and the leaves of the tree T as L(T). Three possible situations are feasible

$$\begin{aligned} U &\subseteq L(T) \\ U &= U_1 \cup U_2 \text{ where } \begin{cases} U_1 \subseteq L(T) \\ U_2 \subseteq V - L(T) \end{cases} \end{aligned}$$

Note that when U satisfies Equation 1, the algorithm we are seeking is same as Kruskal's algorithm. Define a graph $\tilde{G}$=$(V - U, \{(u, v) : u, v \in V - U \wedge (u, v) \in E\})$.

Lemma 1 *If there is no minimum spanning tree of the graph $\tilde{G}$, then lightest spanning tree with U vertices as leaves is not feasible*

Proof: Consider that no spanning tree for the graph $\tilde{G}$ exists. Without any loss of generality assume that there are two trees in the spanning tree forest for the graph $\tilde{G}$. The lightest spanning tree of G must be a tree. Therefore some node $u \in U$ must connect the two tress in the spanning tree forest of the graph $\tilde{G}$. But then u is no longer a leaf node which is a contradiction. □

Property 1 *Minimum spanning tree obtained on the graph $\tilde{G}$ is the lightest spanning tree if we greedily select and add edges such that all vertices U are leaves*

Proof: The minimum spanning tree $\tilde{T}$ is well defined on the graph $\tilde{G}$. Consider each node $u \in U$. We greedily select an edge $(u, v) : v \in \tilde{T}$ and add the edge (u, v) to $\tilde{T}$. Note that the edge (u, v) we greedily chose, $v \in \tilde{T}$. If $v \notin \tilde{T}$ then $v \in U$. But we cannot choose such an edge because to form a spanning tree either of u or v must be connected to $\tilde{T}$ but connecting one of them (suppose u) to $\tilde{T}$ will no longer keep u as leaf. Cost of the new tree is $cost(\tilde{T}) + w_{(u,v)}$. Since $cost(\tilde{T})$ is optimal and we cannot select any edge other than (u, v) (because of the constraint on U vertices as leaves), cost of the new tree is lightest (cannot be decreased any further). □
Based on Lemma 1 and Property 1 we present the following algorithm

```
procedure lightest-spanning-tree(G,w,U)
Input:  Graph G = (V,E); edges weights w_e; U ⊂ V
Output:  Lightest spanning tree if exists
Construct graph G̃ = (Ṽ, Ẽ):
```

```
  Ṽ = V − U
  Ẽ = {(u, v) : u, v ∈ V − U ∧ (u, v) ∈ E}
Apply Kruskal to get MST(G̃) = T̃:
  if T̃ does not exist:
    lightest spanning tree infeasibile
Construct edge set Ē:
  ∀(u, v) ∈ Ē : u ∈ U ∧ v ∉ U
for each u ∈ U:
  makeset(u) sort the edges Ē by w_e
for all edges u,v ∈ E, in increasing order of weight:
  if find(u) ≠ find(v):
    add edge u,v to T̃
    union(u,v)
return T̃
```

Complexity analysis: Other than construction of edge set $\bar{E}$, rest of the complexity analysis is analogous to Kruskal's algorithm which runs in $O(|E| \log |V|)$ time. We construct a set U. For each edge $e = (u, v)$ if $find(u) \neq find(v)$ then we keep that edge in $\bar{E}$. Therefore this takes $O(|E| \log |U|)$ time. Hence total complexity of the algorithm is still bounded by $O(|E| \log |V|)$.

2 Problem 5.32

A server has n customers waiting to be served. For each customer i the service time required is t_i. The cost function for this problem is given by

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer i}) \tag{1}$$

Let the time spent waiting by customer i be w_i. Then cost $T = \sum_{i=1}^{n}(w_i)$. w_i can be defined recursively as $w_i = w_{i-1} + t_{i-1}$. Following the recurrence

$$w_i = w_1 + \sum_{j=1}^{i-1} t_j \tag{2}$$

Without any loss of generality weight time of customer 1 can be considered 0. Therefore $w_i = \sum_{j=1}^{i-1} t_j$. The total cost can be summarized as

$$T = \sum_{i=2}^{n} \sum_{j=1}^{i-1} t_j \tag{3}$$

Equation 3 can be rewritten as

$$T = \sum_{i=1}^{n} f_i t_i \tag{4}$$

where f_i denotes number of times (frequency) t_i appears in cost function T. If we look at Equation 3 t_i has a frequency of (n-i). Frequency of t_1 is $n - 1$.

Lemma 2 *To minimize the cost we choose t_1 as the minimum service time.*

Proof: Assume that t_1 is the minimum service time. For some other t_i, $t_i > t_1$. Our claim is that choosing t_i in place of t_1 can only increase the cost function. Note that only two terms of cost function will change. Previously cost $T_{old} = (n-1)t_1 + (n-i)t_i$. With the proposed change new cost $T_{new} = (n-1)t_i + (n-i)t_1$. But $T_{new} > T_{old}$ and so t_1 must be minimum service time. □

Now as we can see at each step i if we choose t_i to be the i-th minimum service time our problem reduces to subproblems of smaller size. Based on our discussion the greedy algorithm is as follows

```
procedure optimal-ordering(N,T)
Input:  Number of customers N;Set of service time t_i : i ∈ (1, ..., N) T
Output:  Optimal ordering of processing customers
Sort the set T:
  increasing order of t_i
return T
```

3 Problem 6.3

Yuckdonald's is considering opening a series of restaurant along QVH. n possible locations are along a straight line and the distances of these locations from the start of QVH are in miles and in increasing order m_1, m_2, ..., m_n. The constraints are as follows:
1. At each location, Yuckdonald may open one restaurant and expected profit from opening a restaurant at location i is given as p_i
2. Any two restaurants should be at least k miles apart, where k is a positive integer
We use a dynamic programming formulation to solve this problem. We define $P[i]$ as follows

Definition 1 *$P[i]$ is defined as the maximum expected profit at location i*

Based on the constraints we have, we come up with the following recursive definition of $P[i]$

$$P[i] = max \begin{cases} max_{j<i}\{P[j] + \alpha(m_i, m_j) \cdot p_i\} \\ p_i \end{cases}$$

The function α is defined as follows

$$\alpha(m_i, m_j) = \begin{cases} 0 \text{ if } m_i - m_j < k \\ 1 \text{ if } m_i - m_j \geq k \end{cases}$$

Maximum expected profit of location i comes from the maximum of expected profits of location j and whether we can open a location at location i. The profit from opening a restaurant at location i is p_i. Note that P[j] shows the maximum expected profit at location j. There may or may not be a restaurant at location j. Those cases when there is no restaurant at location j will be considered by case k where $k < j$. It is also likely that profit at location i, p_i is higher than the $P[j] + \alpha(m_i, m_j) \cdot p_i$. In those cases we need to do a comparison with p_i as well. Based on the recursion the algorithm writes itself

```
procedure expected-profit(N,P)
Input:  N locations; P[1,..,N] where P[i] denotes profit at location i
Output:  Maximum expected profit P_max
Declare an array of maximum Expected profit Profit[1..N]:
  Profit[i] denotes maximum expected profit at location i
for i = 1 to N:
  Profit[i] = 0
for i = 2 to N
  for j = 1 to i-1
    temp = Profit[j] + α(m_i, m_j) · P[i]
    if temp > Profit[i]:
       temp = Profit[i]
  if Profit[i] < P[i]:
    Profit[i] = P[i]
```

Complexity analysis : There are two for loops which gives a O(n^2) complexity to this algorithm.

4 Problem 6.7

A subsequence is palindromic if it is the same whether read left to right or right to left. We have to devise an algorithm that takes a sequence x[1,...,n] and returns the length of the longest palindromic subsequence. We give the following definition of a palindrome

Definition 2 *Sequence x is called a palindrome if we form a sequence $\bar{x}$ by reversing the sequence x then $x = \bar{x}$*

This definition allows us to formulate an optimal substructure for the problem. If we look at the sub-string x[i,..,j] of the string x, then we can find a palindrome of length at least 2 if $x[i] = x[j]$. If they are not same then we seek the maximum length palindrome in subsequences x[i+1,...,j] and x[i,...,j-1]. Also every character x[i] is a palindrome in itself of length 1. Therefore base cases are given by x[i,i] = 1. Let us define the maximum length palindrome for the substing x[i,...,j] as L(i,j)

$$\begin{aligned} L(i,j) &= \begin{cases} L(i+1,j-1)+2 \text{ if } x[i]=x[j] \\ max\{L(i+1,j), L(i,j-1)\} \text{ otherwise} \end{cases} \\ L(i,i) &= 1 \ \forall i \in (1,...,n) \end{aligned}$$

Based on the recursive definition we present the following algorithm

```
procedure palindromic-subsequence(x[1,...,n])
Input:  Sequence x[1,...,n]
Output:  Length of the longest palindromic subsequence
Declare a 2 dimensional array L:
  L[i,j] stores the length of longest palindromic subsequence in x[i,...,j]
for i = 1 to n:
  L[i,i] = 1 (Each character is a palindrome of length 1)
for i = 1 to n:
  for s = 1 to n-i:
    j = s+i
    L[s,j] = compute-cost(L,x,s,j)
    L[j,s] = compute-cost(L,x,j,s)
return L[1,n]
```

The procedure compute-cost is an implementation of recursive formulation shown above. Note that we include details not mentioned in the recursive formulation

```
procedure compute-cost(L,x,i,j)
Input:  Array L from procedure palindromic-subsequence; Sequence x[1,...,n]
    i and j are indices
Output:  Cost of L[i,j]
if i = j:
  return L[i,j]
else if x[i] = x[j]:
  if i+1 < j-1:
    return L[i+1,j-1] + 2
  else:
    return 2
else:
  return max(L[i+1,j],L[i,j-1])
```

Complexity analysis: First for loop takes $O(n)$ time while the second for loop takes $O(n-i)$ which is also $O(n)$. Therefore the total running time of the algorithm is given by $O(n^2)$.

5 Problem 6.12

A convex polygon P on n vertices in the plane (specified by their x and y coordinates). A triangulation of P is a collection of n-3 diagonals of P such that no towo diagonals intersect. Cost of a triangulation is given by the sum of lengths of diagonals in it. In this problem we are deriving an algorithm for finding a triangulation of minimum cost.
We label the vertices of P by 1,...,n starting from an arbitrary vertex and walking clockwise. For $1 \leq$ i $<$ j $\leq$ n, we denote a subproblem A(i,j) which computes the minimum cost triangulation of the polygon spanned by vertices i,i+1,...,j. Formulating the subproblem this way we are enumerating all the possible diagonals such that the diagonals do not cross each other. Since we are only considering in clockwise directions, diagonals generated by the subproblems can not cross each other. Recursive definition of A[i,j] is as follows

$$A[i,j] = min_{i \leq k \leq j}\{A[i,k] + A[k,j] + d_{i,k} + d_{k,j}\} \tag{5}$$

Base cases are given by A[i,i] = 0. This is because a diagonal can not be formed due to the same vertex. Next equation for d takes care of the fact that a diagonal can be evaluated only if the order of the vertices differ by 2.

$$d_{i,j} = \begin{cases} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} & \text{if } j - i \geq 2 \\ 0 \text{ otherwise} \end{cases}$$

Based on the recursive definition and base cases, we present the algorithm as follows

```
procedure min-cost-triangulation(p_1,...,p_n)
Input:  n vertices p_1,...,p_n of a convex polygon P:
    Vertices are ordered in clockwise direction
Output:  Minimum cost triangulation of P
Declare a 2 dimensional array A:
  A[i,j] stores minimum cost triangulation of polygon spanned by vertices i,i+1,...,j
for i = 1 to n:
  A[i,i] = 0
for s = 1 to n-1:
  for i = 1 to n-s:
    j = i + s:
    A[i,j] = min_{i≤k≤j}{A[i,k] + A[k,j] + d_{i,k} + d_{k,j}}
return A[1,n]
```

Complexity Analysis: Complexity analysis of the algorithm is analogous to chain matrix multiplication problem presented on page 171 of textbook. Computation of each A[i,j] takes O(j-i) time which is O(n). There are two outer loops which takes O(n^2). The algorithm runs in O(n^3) time.

6 Problem 6.17

Given an unlimited supply of coins of denomination x_1, x_2,...,x_n, we wish to make change for a value v. We are seeking an O(nv) dynamic-programming for the following problem
Input: x_1,...,x_n;v
Question: Is it possible to make change for v using coins of denominations $x_1, ..., x_n$
We define D(v) as a predicate which evaluates to true if it is make change for v using available denominations x_1, x_2,...,x_n. If it is possible to make change for v using given denominations x_1, x_2,...,x_n, then it is possible to make change for $v - x_i$ using the same denominations with one coin of x_i being chosen. But since we don't know which i will finally give us true value (which indicates change of v is possible) we will do a logical or over all i. Recursive definition of D(v) is written as follows

$$D(v) = \vee_{1 \leq i \leq n} \begin{cases} D(v - x_i) & \text{if } x_i \leq v \\ false & \text{otherwise} \end{cases}$$

Based on Equation 6 the algorithm is straigtforward

```
procedure make-change(x_1, ..., x_n, V)
Input:  Denominations of available coins x_1,...,x_n,
    Value V for which we are seeking denominations
Output:  true if making change with available denomination is feasible
    false otherwise
Declare an array D of size V + 1
D[0] = true
for i = 1 to V
  D[i] = false
for v = 1 to V:
  for j = 1 to n:
    if x_j ≤ v:
       D[v] = D[v] ∨ D[v − x_j]
    else:
       D[v] = false
return D[V]
```

Complexity Analysis: There are two for loops in the algorithm where one loop runs $|V|$ times while the other loops run $|n|$ times. Therefore the complexity of the algorithm is $O(|n||V|)$.