Chapter 16: Greedy Algorithms

Activity Selection Problem

16.1-1) Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i,j]$ as defined above and also produce the maximum-size subset of mutually compatible activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

Solution: Let $S_{ij}$ be the interval between activity $a_i$ and $a_j$ exclusive. The following algorithm builds the table for the activity selection problem where $s$ and $f$ are arrays of length $n$ containing the start and finish times, respectively, for the $n$ different activities.

BUILD-ACTIVITY-TABLE(s,f)

1. $n = s.length$
2. Initialize $c$, an $n \times n$ array
3. for $j=0$ to rows-1
4.  for $i=j$ to 0
5.     // base case
6.     $c[j][j] = -1$
7.     $c[i][j] = 0$
8.     // general case
9.     if $(j-1)>1$
10.    // for every $a_k$ in $S_{ij}$ (Compute $c[i][j]$ and store activity number $k$)
11.       for $k=i+1$ to $j-1$
12.          if $f[i] <= s[k] AND f[k] <= s[j]$ // if activity fits in interval
13.             if $c[i][j] < (c[i][k] + c[k][j] + 1)$ // update max num activities
14.                $c[i][j] = c[i][k] + c[k][j] + 1$
15.                $c[j][i] = k$ // store activity number
16.     return $c$

The return value, $c$, is an $n \times n$ array. BUILD-ACTIVITY-TABLE(s,f) fills in the values of $c$ above the diagonal and saves the $k$ values below the diagonal. It takes $O(n^2)$ to fill in the values above the diagonal but for each $c[i][j]$, we must compute for each $a_k$ in $S_{ij}$. This is shown in the inner loop. Just like in the MATRIX-CHAIN-MULTIPLY, the total running time is $O(n^2)$.

The optimal solution is then built from the table using the following algorithm:

OPTIMAL-ACTIVITY-SELECTION($c, s, f, i, j$)

1. return $\{a_i\} U$ SUBOPTIMAL-ACTIVITY-SELECTION($s, f, i, j$) U $\{a_j\}$
where \([i, j]\) is the index of the cell above the diagonal of \(c\) containing the largest value, number of selected scheduled activities. (This can be returned by BUILD-ACTIVITY-TABLE(s,f) as well but has been omitted for simplicity)

SUBOPTIMAL-ACTIVITY-SELECTION(c,s,f,i,j)
if \(i<j\) AND \(j<\text{a.length}\) //if time interval is positive
\[ k=c[i][j] \]
if \(i<k\) AND \(k<j\) //kth activity was added
return SUBOPTIMAL-ACTIVITY-SELECTION(c,s,f,i,k) U \{a_k\} U
SUBOPTIMAL-ACTIVITY-SELECTION(c,s,f,k,j)

Note that \(s\), and \(f\) together form the list of activities where activity \(a_i\) has starting time \(s_i\) and final time \(f_i\). The dynamic-programming algorithm runs in \(O(n^2)\) while the greedy algorithm runs in \(O(n)\). Therefore, the GREEDY-ACTIVITY-SELECTOR is a much better solution to the activity-selection problem.

16.1-2 Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that in compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

Solution:
Let \(n\) be the total number of activities, \(a_1,a_2,...,a_n\).

GREEDY-ACTIVITY-SELECTOR-JMC(s,f)
\[ n = s.\text{length} \]
\[ A = \{a_n\} \]

for \(m=n-1\) to 1
if \(f[m]<=s[k]\) //greedy step
\[ A=\{a_m\} U A \]
\[ k=m \]

return A

where \(n\) is the number of activities,
\(s\) is an \(n\) array and \(s[k]\) contains the starting time of \(a_k\),
Assume \(s\) is monotonically increasing sorted array,
\(f\) is an \(n\) array and \(f[k]\) contains the finish time of \(a_k\),

This algorithm iterates through the activities starting from the activity with the latest starting time. If the current activity has not finished before the last activity has started, then that activity is skipped and not added to optimal solution. However, if the candidate activity, \(a_k\), does finish before the last one starts, then that activity is added to the solution. This is the greedy step and we know that the first activity with the latest starting time is going to be chosen before all the other ones because the array of activities is sorted in increasing order.

In order for this approach to yield an optimal solution, it is sufficient to prove that any activity with the latest starting time belongs to a maximum-size subset of mutually compatible activities of \(S_k\).

Claim: Consider any nonempty subproblem \(S_k\) and let \(a_m\) be an activity in \(S_k\) with the last starting time. Then \(a_m\) is included in some maximum-size subset of
mutually compatible activities of $S_k$.

Proof: Let $a_i$ be an activity with starting time $s_i$ and final time $f_i$. Let 
\{a_1, a_2, ..., a_n\} be a set of activities monotonically increasing based on their starting time. That is, $s_1 \leq s_2 \leq s_3 \leq ... \leq s_n$. Let $A_k$ be a maximum-size subset of mutually compatible activities $S_k$, and let $a_j$ be the activity in $A_k$ with the latest starting time.

Case 1: $a_j = a_m$
Then, since $a_j \in A_k$, $a_j$ is in some maximum-size subset of mutually compatible activities of $S_k$. □

Case 2: $a_j \neq a_m$
Then set $A'_k = A_k - \{a_m\} + \{a_j\}$. Since $A_k$ is some maximum-size subset of mutually compatible activities in $S_k$, then $f_1 \leq f_2 \leq f_3 \leq ... \leq s_m$. Since $a_j$ and $a_m$ are both activities with the latest starting time in $S_k$, $s_m = s_j$. Then we have that $f_1 \leq f_2 \leq f_3, ... \leq s_m = s_j$ and $|A'_k| = |A_k|$. Necessarily, $A'_k$ must be a maximum-size subset of mutually compatible activities. Since $a_j \in A'_k$, $a_j$ is in some maximum size subset of mutually compatible activities of $S_k$. □

16.1-3 Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

Solution: This problem can be solved like the activity-selection problem with one single lecture hall with the exception that when an activity cannot be scheduled in the first lecture hall, it should be scheduled in the second. If it cannot be scheduled in lecture 2, schedule in lecture 3. If it cannot be scheduled in lecture 3, schedule in lecture 4, so on and so forth until it is scheduled in some lecture hall. For all activities, if an activity cannot be scheduled in the first lecture, it must be attempted to be scheduled in the order lecture 2, lecture 3, ....

16-2.1 Prove that the fractional knapsack problem has the greedy-choice property.

Solution

Proof:
Let $n$ be the number of items, each item $i \in \{1, 2, ..., n\}$, with value $v_i$ and weight $w_i$. Let $W$ be the total weight the knapsack can hold. The task is to fill the knapsack with as many items as possible while maximizing the total knapsack value. An item does not need to be completely added. That is, a fraction of a particular item may be added to the knapsack if it maximizes the total value. To do this, first compute the value per pound $v_i/w_i$. Sort the items in ascending order based on the value per pound. Let item $j$ be the item with the largest value per pound.

Case 1: If $W = w_j$, then item $j$ fills up the knapsack perfectly and we are done.

Case 2: If $W < w_j$, then the whole item does not fit into the knapsack, in which case, only add as much of item $j$ as can fit into the knapsack and we are done.

Case 3: Otherwise, $W > w_j$. Add item $w_j$ to the knapsack. The knapsack can now take up to $W - w_j$ additional pounds. Item $j$ was the item with the most value per pound, so we necessarily choose this item in a greedy manner to maximize the knapsack value. This
leaves the following subproblem: fractional knapsack with maximum weight of $W - w_j$ and $n - 1$ objects to choose from. Note that item $j$ was chosen without considering results from subproblems. Therefore the fractional knapsack problem has the greedy-choice property. □

16.2-2 Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where $n$ is the number of items and $W$ is the maximum weight of items that the thief can put in his knapsack.

Solution:

```plaintext
KNAPSACK(w,v, W, n)
\\\initialization
for i=1 to n //knapsack value is 0 if knapsack weight is 0
  B[i,0]=0

for j=0 to W //knapsack value is 0 if there are 0 items to choose from
  B[0,j] = 0

for i=1 to n
  for j=0 to W
    if w[i]<= j // item i can be part of the solution
      if v[i] + B[i-1, j-w[i] ] > B[i-1, w] // (1)
        B[i,j]=v[i] + B[i-1, j-w[i] ]
      else // (2)
        B[i,j]=B[i-1,w]
    else //item does not fit
      B[i,j]=B[i-1, j-w[i] ]

w and v are arraya of length n
W is the maximum weight that the knapsack can hold B is an (w+1) x (n+1) array where B[i][j] is the maximum value for a knapsack of weight $j$ with the first $i$ items to choose from.

(1) for current weight $j$, look at the the knapsack of max weight $j-1$. For current item $i$, if the item is added to the knapsack, we must look at the knapsack with i-1 items and weight j-w[i]. The value of the knapsack would be the value of the new item plus the max value for knapsack with i-1 items and j-w[i] weight, v[i] + B[i-1, j-w[i] ]]. If adding this new item yeilds a larger knapsack total value, then “add” the item by updating the maximum value ( B[i,j] = v[i]+ B[i-1, j-w[i] ] )

(2) if adding this new item i does not yield a larger knapsack total value, then don’t add the item. So, the total value of the knapsack is still the value as if you only had i-1 items to choose from. Thus, B[i,j]=B[i-1, j]

We can obtain the list of items from array B using the following algorithm:

```plaintext
KNAPSACK-ITEMS(v,w,B)
  i=n
  k=W
  while i>0 AND k>0
    if B[i,k] != B[i-1,k] // (3)
      add i to the items in the knapsack
    k=k-w[i]
```
look at the last entry in B with item i, and weight W. If the knapsack value changed from 
i-1 to i items, then item i was added and we now look at the suboptimal knapsack of weight 
k-w[i]. Otherwise, item i was not added to the knapsack and we simply look at the optimal 
knapsack with the same weight but i-1 items.

In the KNAPSACK algorithm, there are two for loops. The inner loop iterates for each item 
while the inner loop iterates for each weight. Since there are n items and the maximum 
knapsack weight is W, the algorithms runs \( O(nW) \). □

A task-scheduling problem as a matroid

16.5-1 Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty 
\( w_i \) replaced by \( 80 - w_i \).

**Solution:**

List of activities where \( a_i \) is activity i with deadline \( d_i \) and penalty weight \( w_i \):

\[
\begin{array}{c|cccccccc}
 a_i & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 d_i & 4 & 2 & 4 & 3 & 1 & 4 & 6 \\
 w_i & 10 & 20 & 30 & 40 & 50 & 60 & 70 \\
\end{array}
\]

Execution where \( a_i^j \) is activity i with deadline \( d_i = j \):

\[
\begin{array}{ccccccccc}
 t_i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
 a_1^1 & & & & & & & \\
 a_2^2 & a_1^4 & & & & & \\
 a_2^2 & a_1^4 & a_3^4 & & & & \\
 a_2^2 & a_4^3 & a_1^4 & a_3^4 & & & \\
 a_1^5 & a_4^3 & a_1^4 & a_3^4 & a_2^2 & & \\
 a_1^5 & a_4^3 & a_6^6 & a_3^6 & a_2^2 & a_1^4 & \\
 a_1^5 & a_4^3 & a_6^6 & a_3^6 & a_6^5 & a_2^2 & a_1^4 & \\
\end{array}
\]

Note that at \( t=4 \), \( a_5 \) took the place of \( a_2 \) because \( a_5 \) has a higher penalty value than all the 
current activities, so \( a_5 \) replaces \( a_2 \), the activity with the smallest penalty value (\( w_i = 20 \)). 
Likewise, at \( t=5 \) \( a_6 \) replaces \( a_1 \), the activity with the smallest penalty value (\( w_i = 10 \)) with 
early arrival. Early activities are in monotonically increasing order.

The final optimal schedule is: \{ \( a_5, a_4, a_6, a_3, a_7, a_2, a_1 \) \} which 
has a total penalty incurred of \( w_2 + w_1 = 30 \).

16.2 Suppose you are given a set \( S = \{ a_1, a_2, ..., a_n \} \) of tasks, where task \( a_i \) requires \( p_i \) units of 
processing time to complete, once it has started. You have one computer on which to run 
these tasks, and the computer can run only one task at a time. Let \( c_i \) be the completion time 
of task \( a_i \), that is, the time at which task \( a_i \) completes processing. Your goal is to minimize 
the average completion time, that is, to minimize \( \frac{1}{n} \sum_{i=1}^{n} c_i \).
(a) Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task $a_i$ starts, it must run continuously for $p_i$ units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

**Solution:** An activity’s completion time, $c_i$, is dependent on the previously scheduled activities. If $a_i$ has a short processing time $p_i$ and it gets scheduled after activities with longer processing time, $c_i$ will be larger than it could have been if it were scheduled before. Thus, a greedy algorithm for scheduling activities $a_1, a_2, ..., a_n$ requires to first sort the activities in increasing order based on their processing time $p_i$ and schedule the activities in order starting with activity with the shortest processing time.

Sorting the activities can be done in $O(n \log n)$ while scheduling the activities in done in $O(n)$ since it would only require to transverse the array of sorted activities. The total running time is thus $O(n \log n)$.

**Proof:**
To prove that scheduling the activities in ascending order based on their processing time $p_i$, yields a minimized average completion time, it is sufficient to prove that the average completion time is at its minimum when an activity of shorter processing time gets scheduled before an activity with longer processing time. Let $S, a_i, p_i$ and $c_i$ be as defined in the problem description. Let us assume that we have scheduled all activities in such a way that the average completion time is minimal with the exception of the last two activities $a_i$ and $a_j$ with $i \leq j$. The the average completion time is given by:

$$\frac{1}{n} \sum_{k=1}^{n} c_k = \frac{1}{n} \cdot c_m + ... + \frac{1}{n} \cdot c_{m-2} + \frac{1}{n} \cdot c_i + \frac{1}{n} \cdot c_j$$

where \( \{m_1, m_2, ..., m_n\} = \{1, 2, ..., n\} - \{i, j\} \)

If $a_i$ gets scheduled first and $a_j$ second, we have that

$$c_i = p_i + \sum_{k=1}^{n-2} p_{m_k}$$

$$c_j = p_i + p_j + \sum_{k=1}^{n-2} p_{m_k}$$

If $a_j$ gets scheduled first and $a_i$ second, we have that

$$c'_j = p_j + \sum_{k=1}^{n-2} p_{m_k}$$

$$c'_i = p_j + p_i + \sum_{k=1}^{n-2} p_{m_k}$$
Suppose now that the tasks are not all available at once. That is, each task cannot start until its release time \( r_i \). Suppose also that we allow preemption, so that a task can be suspended and restarted at a later time. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

**Solution:** As was proven in the previous subproblem, the average completion time is minimized when the activities are scheduled in ascending order. So, given that preemption is allowed, when an activity arrives, it must be inserted into its proper location within the sorted list of activities while scheduling continues to be done in ascending order of activity processing time. The initial sort will run in \( O(n \log n) \). The worst case of insertion into a sorted array is \( O(n) \) for each new activity arrival (runs \( O(n + n) = O(n^2) \)) and the best case is \( O(1) \) for each new activity arrival (runs \( O(n) \)). Thus, the running time of this algorithm is in the worst case \( O(n \log n) + O(n^2) = O(n^2) \) and in the best case \( O(n \log n) + O(n) = O(n \log n) \)

**References**
