Homework 8

Daoyu Wang PB21030794

November 2

Contents

1 16.1-3

1.1 Greedy Approach 1

Always selecting the activity of least duration from among those that are compatible with previously selected activities.

Assume that we have a set of activities and their duration sequence is $\{[0:10), [5:14), [10:20)\}$. If we select the activity of least duration, the maximum-size set of mutually compatible activities is $\{[5:14)\}$ because it has duration of 9, shorter than other activities who have duration of 10. However, the maximum-size set of mutually compatible activities is **actually** $\{[0:10), [10:20)\}$.

1.2 Greedy Approach 2

Always selecting the compatible activity overlaps the fewest other remaining activities.

Assume that we have a set of activities and their duration sequence is $\{[0:2), [1:3), [1:3), [1:3), [2:5), [4:6), [5:8), [7:9), [7:9), [7:9), [8:10)\}$. If we select the activity overlaps the fewest other remaining activities, we will select $\{[4:6), [0:2), [8:10)\}$. However, the maximum-size set of mutually compatible activities is **actually** $\{[0:2), [2:5), [5:8), [8:10)\}$.

1.3 Greedy Approach 3

Always selecting the compatible remaining activity with the earliest start time.

Assume that we have a set of activities and their duration sequence is $\{[0:10), [1:2), [2:3)\}$. If we select the compatible remaining activity with the earliest start time, we will select $\{[0:10)\}$. However, the maximum-size set of mutually compatible activities is **actually** $\{[1:2), [2:3)\}$.

2 16.2-3

2.1 Algorithm

The Greedy Algorithm is as follows:

Algorithm 1 Greedy Algorithm of specific 0-1 bag

```
1: X ← ∅
2: value ← 0
3: i ← 1
4: while i ≤ n do
5:     if w_i ≤ c then
6:         x_i ← 1
7:         c ← c − w_i
8:         value ← value + v_i
9:     else
10:        break
11:    end if
12:    i ← i + 1
13: end while
```

2.2 Prove

Assume that $X = (x_1, x_2, \cdots, x_n)$ is the solution of Greedy Algorithm.

Case 1 $\forall x \in X, x = 1$. In this case, the solution of Greedy Algorithm is the optimal solution.

Case 2 $\exists m < n, k \geq m+1, x_k = 0; k \leq m, x_k = 1$. This represents: $X = (1, 1, \cdots, 1, 0, \cdots, 0)$. We will prove that it is the optimal solution as follows:

Assume that $Y = (y_1, y_2, \cdots, y_n)$ is the optimal solution and $\exists k \leq n$, k is the first k that $y_k \neq x_k$.

Prepare some precondition:

1. $0 < \sum_{j=1}^{m} w_j \leq c$
2. $\sum_{j=1}^{m+1} w_j > c$

Case 2.1 $y_k = 1$
In this case, we have

1. $m + 1 \leq k \leq n$
2. $i \leq k - 1, y_i = 1$
3. $$\begin{aligned}\sum_{j=1}^{n} w_j y_j &\geq \sum_{j=1}^{k} w_j y_j \\ &= \sum_{j=1}^{k} w_j \geq \sum_{j=1}^{m+1} w_j \\ &= \sum_{j=1}^{m} w_j x_j > c\end{aligned}$$

So, $y_k = 1$ is not a feasible solution.

Case 2.2 $y_k = 0$
In this case, we have

1. $1 \leq k \leq m$
2. $Y = (1, 1, \cdots, 1, 0, y_{k+1}, \cdots, y_n)$

We use proof by **contradiction**, assuming that X is not the optimal solution. Assume that there are s items of $Y[k + 1 : n]$ is 1, we record them as $y_{t_1}, y_{t_2}, \cdots, y_{t_s}$.

It is known that Y is optimal. Now, we have some conclusions:

$$\begin{aligned}Y.Weight &= \sum_{i+1}^{k-1} w_i + \sum_{i=1}^{s} w_{t_i} \\ &= \sum_{i=1}^{m} w_i + \sum_{i=1}^{s} w_{t_i} - \sum_{i=k}^{m} w_i\end{aligned}$$

If $s \geq m - k + 2$:

1. $t_1 \geq k + 1$
2. $t_i \geq t_1 + i - 1 \geq i + k$
3. $t_s \geq t_1 + s - 1 \geq k + 1 + (m - k + 2) - 1 = m + 2$
4. w_i is **monotonically increasing**

So, we have:

$$\begin{aligned} Y.Weight &\geq \sum_{i=1}^{m} w_i + w_{t_s} + \sum_{i=1}^{m-k+1} w_{t_i} - \sum_{i=k}^{m} w_i \\ &\geq \sum_{i=1}^{m} w_i + w_{m+2} + \sum_{i=1}^{m-k+1} (w_{t_i} - w_{i+k-1}) \\ &\geq \sum_{i=1}^{m} w_i + w_{m+1} \\ &> c \end{aligned}$$

It doesn't satisfy the limit of capacity. So, $s \leq m - k + 1$. According to v_i is **monotonically decreasing**:

$$\begin{aligned} Y.Value &= \sum_{i=1}^{k-1} v_i + \sum_{i=1}^{s} v_{t_i} \\ &< \sum_{i=1}^{k-1} v_i + \sum_{i=k}^{m} v_i \\ &= \sum_{i=1}^{m} v_i = X.Value \end{aligned}$$

As we can see, Y is the optimal solution but its value is fewer than X which is not the optimal solution in the assumption. It's **contradictory**. So, X is the optimal solution.

3 16.2-5

3.1 Algorithm

The **left endpoint** of the new interval is always defined as the leftmost point among the remaining points that have not been included in the existing intervals.

3.2 Prove

We have a restrict to a new interval: it must includes the leftmost point among the remaining points.

Of all the choices that could include the leftmost point, placing the left endpoint at this point is the smartest choice because it can include the most points it can. So the Greedy Algorithm must be optimal.

Similarly, always placing the **right endpoint** of the new interval at the rightmost point of the remaining points can achieve the same result.

4 16.3-3

4.1 Specific Case

Applying Huffman coding, the prefix coding of all letters is as follows:

letter	**a**	**b**	**c**	**d**
prefix code	0000000	0000001	000001	00001
letter	**e**	**f**	**g**	**h**
prefix code	0001	001	01	1

4.2 General Case

Firstly, we can prove than in Fibonacci array, $F_{n+2} = 1 + \sum_{i=1}^{k} F_i$.

1. **Proof:** $F_{n+2} = 1 + \sum_{i=1}^{k} F_i$
2. **Base Case:** $n = 1$, $F_3 = 2 = 1 + 1 = 1 + F_1$.
3. **Inductive Hypothesis:** Assume that $F_n = 1 + \sum_{i=1}^{k-1} F_i$ is true.
4. **Inductive Step:** $F_{n+1} = F_n + F_{n-1} = 1 + \sum_{i=1}^{k-2} F_i + F_{n-1} = 1 + \sum_{i=1}^{k-1} F_i$.

So, $F_{n+2} = 1 + \sum_{i=1}^{k} F_i$ is true.

To use this fact, to show the desired Huffman code is optimal, we claim that as we are greedily combining nodes. At each stage we can maintain one node which contains all of the least frequent letters.

Initially, this consists of just the least frequent letter.

At stage k, inductively, we assume that it contains the k least frequent letters. This means that it has weight $\sum_{i=1}^{k} F_i = F(k+2) - 1$. Now, all of the other nodes at this stage have weights $\{F_{k+1}, F_{k+2}, F_{k+3}, \cdots, F_n\}$

Clearly the two lowest weight nodes are this stage is $Fk + 2 - 1$ and F_{k+1}. Therefore, the greedy algorithm tells us that we will choose the node with the minimum frequency among the current remaining nodes. So F_i corresponds with a prefix code: 0 of length $n - i$ and 1 of length $i - 1$

5 sch2-1

5.1 Algorithm

Algorithm 2 Work Distribution

Input: $c[n][n]$

Output: $vectorres$ indicates that the work of the $i-th$ row needs to be assigned to the $res[i]-th$ person

```
1: res ← ∅
2: bestspend ← INTMAX
3: function BackTrace(cols, spend)
4:     if cols.size() = n then
5:         if spend < bestspend then
6:             bestspend ← spend
7:             res ← cols
8:             return
9:         end if
10:    end if
11:    for j ← 1 to n do
12:        if !isValid(cols, i) or spend > bestspend then
13:            continue
14:        end if
15:        cols.push(j)
16:        spend = spend + c[cols.size()][j]
17:        BackTrace(cols, spend)
18:        spend = spend − c[cols.size()][j]
19:        cols.pop()
20:    end for
21:    return
22: end function
```

Algorithm 3 Judge if the column is Valid

```
1: function isValid(cols, i)
2:     for j ← 1 to cols.size() do
3:         if cols[j] = i then
4:             returnfalse
5:         end if
6:     end for
7:     returntrue
8: end function
```

5.2 Statement

Use backtracking. We can just call $BackTrace(\emptyset, 0)$ to get the result.

6 sch2-2

6.1 Algorithm

Algorithm 4 Task Scheduling

Input: n, k and $time[n]$
Output: $res[n]$ indicates which machine the $i-th$ task is assigned to.

```
1: res ← ∅
2: bestspend ← INT_MAX
3: Arr ← ∅ indicates the Temporary Arrange of every task
4: MQT ← ∅ indicates TotalTime in every Machine Queue
5: function BackTrace(taskid, totalspend, Arr)
6:     if then task_id = n
7:         if then totalspend < bestspend
8:             bestspend ← totalspend
9:             res ← Arr
10:            return
11:        end if
12:    end if
13:    for i ← 1 to k do
14:        if totalspend < bestspend then
15:            if MQT[i] + time[taskid] < bestspend then
16:                Arr[taskid] ← i
17:                MQT[i] ← MQT[i] + time[taskid]
18:                BackTrace(taskid + 1, max(totalspend, MQT[i]), Arr)
19:                MQT[i] ← MQT[i] − time[taskid]
20:                Arr[taskid] ← 0
21:            end if
22:        end if
23:    end for
24:    return
25: end function
```

6.2 Statement

Use backtracking and here we only consider the case that $n > k$: We can just call $BackTrace(0, 0, Arrange)$ to get the result.

Furthermore, we can use the greedy result to initialize bestspend to speed up the pruning process. The greedy method is selected as follows:

1. Pre-sort the task sequence from longer time to shorter time
2. Push the first k tasks into k machines
3. Always select the machine with the least total time and assign remaining task to it

Unfortunately, It can only get a good, more average result, but it may not necessarily get the best result. Consider the following case: task sequence is $\{16, 14, 12, 11, 10, 9, 8\}$ and there are 3 machines. If we use the greedy method, we will get the arranging sequence as $\{1, 2, 3, 3, 2, 1, 3\}$ which has **TotalTime** of 12+11+8 = 31 However, the optimal solution is $\{1, 1, 2, 2, 1, 1, 1\}$ which has **TotalTime** of $16 + 14 = 30$.

It is obvious that the greedy result is definitely not the worst or even worse result. It is generally closer to the optimal result, so the pruning effect is very good. The original time complexity of the backtracking method is close to $O(k^n)$. Using greedy result optimization can significantly reduce this complexity.