Dynamic programming, 1

1 Weighted interval scheduling problem

Just like interval scheduling problem, in the current set-up we have n requests r_1, …, r_n such that each request r_i specifies the starting time $s(i)$ and finishing time $f(i)$. The main difference from the interval scheduling problem is that each request r_i also submits the amount v_i prepared to be paid for the use of the resource. This leads to the following formulation of the problem:

Design a schedule such that the selected set of requests is a compatible set and achieves the maximum value for the use of the resource.

Clearly, the original interval scheduling problem is a subproblem of the weighted interval scheduling problem because we can associate with each interval the value 1. Thus, more formally the problem asks to find a compatible set S of requests such that the sum

$$\sum_{i \in S} v_i$$

achieves the maximum value. Note that we identify each request r_i with its index i.

Exercise 1. Show that applying the greedy algorithm developed for the interval scheduling problem does not solve the weighted interval scheduling problem.

2 Designing the algorithm

We order all requests according to their finishing times in $O(n \log(n))$ time. Let us assume that the ordering obtained is

$$f(1) \leq f(2) \leq \ldots \leq f(n).$$

We fix this order, and say that request r_i comes before the request r_j if $i < j$. Clearly not every request r_i that comes before r_j is compatible with r_j. We would like to select the rightmost interval on the left side of r_j compatible with r_j. In other words, we want to select the largest i such that r_j starts on or after r_i finishes the use of the resource. We denote the index of this i by $p(j)$. If all r_i that come before i are not compatible with j then we simply put $p(j) = 0$.

Exercise 2. Consider the sequence $p(1), p(2), \ldots, p(n)$.

1. Give examples where this sequence consist of 0s only.
2. Give examples where the sequence alternates between 0 and 1.
3. Give examples where the sequence has the form 0, 1, …, $n - 1$.

Let $\mathcal{O}$ be an optimal solution for the problem. Our goals is to study the properties of $\mathcal{O}$. Our hope is that studying the optimal solution will give us some ideas on designing the algorithm.

Let us take the last request r_n. Obviously, either r_n belongs to $\mathcal{O}$ or r_n does not belong to $\mathcal{O}$. We would like to understand what each side of this dichotomy implies.

Case 1: $r_n \in \mathcal{O}$. In this case, by the definition of $p(n)$ we have that the requests

$$p(n)+1, p(n)+2, \ldots, n-1$$

do not belong to $\mathcal{O}$ because they all are incompatible with r_n. Moreover, the optimal solution $\mathcal{O}$ with r_n removed from it must be an optimal solution for the set of requests:

$$1, 2, \ldots, p(n).$$

Case 2: $r_n \notin \mathcal{O}$. In this case the solution $\mathcal{O}$ must be optimal for the set of requests:

$$1, 2, \ldots, n-1.$$

These observations give us a good insight into building an optimal solution. We need to find optimal solutions for smaller problems with requests $1, 2, \ldots, j$, where j ranges from 1 to n. So, lets us denote the optimal solution for the requests 1, ..., j by $\mathcal{O}_j$. We also introduce the notation $OPT(j)$ for the sum of values of the solution $\mathcal{O}_j$. The two cases above we consider tell us the following:

1. If $r_n \in \mathcal{O}$ then $OPT(n) = OPT(p(n)) + v_n$.
2. If $r_n \notin \mathcal{O}$ then $OPT(n) = OPT(n-1)$.

In fact, the arguments above tell us more.

Observation 1 *For all j the following holds:* $OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$.

The next observation, which is easy to prove, tells us when request j belongs to $\mathcal{O}_j$:

Observation 2 *Request j belongs to an optimal solution set for $\{1, \ldots, j\}$ if and only if*

$$OPT(p(j)) + v_j \;\geq\; OPT(j-1).$$

Exercise 3. Explain why the observations are true.

Thus, the following algorithm solves the weighted interval scheduling problem:

Algorithm 1 Compute-Opt(j)-algorithm

```
If $j = 0$ then return 0
− Else return max{$v_j$+Compute-Opt($p(j)$), Compute-Opt($j-1$)}.
Endif.
```

How would we go about proving that the algorithm provides a correct solution to the problem? If $j = 0$, then clearly the solution is correct. If $j = 1$ then again it is clear that the solution gien by the algorithm is a right one. One can verify that the solution of the problem for $j = 2, 3$ is also correct. Now suppose that for all $i < j$ the algorithm computes the correct value for requests r_1, ..., r_i. This implies, in particular, the following:

$$OPT(p(j))\text{=Compute-Opt}(p(j)) \quad \text{and} \quad OPT(j-1)\text{=Compute-Opt}(j-1).$$

Now from Observation 1, we conclude:

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\} =$$
$$\max\{\text{Compute-Opt}(p(j))\} + v_j,\ \text{Compute-Opt}(j-1)\} = \text{Compute-Opt}(j).$$

Thus, we proved that the algorithm above is correct.

Implementing the algorithm in the way it is written gives, unfortunately, and exponential running time. The reason is that computing Compute-Opt(j) generates two independent calls for Compute-Opt($p(j)$) and Compute-Opt($j-1$). Therefore, the number of independent calls at every step of the algorithm doubles, and this leads to exponential blow-up.

3 Refining the algorithm

The crucial step in reducing the number of calls in the algorithm above is to realise that there are only n subproblems, and each call needs a solution to one of them. In other words, the calls are only made to Compute-Opt(0), Compute-Opt(1), Compute-Opt(2), ..., Compute-Opt($n-1$), and Compute-Opt(n). So, to make the algorithm effective we need to do the following. Once Compute-Opt(i) is found, we memorise the solution. Every time when a call is made to Compute-Opt(i), we need to re-use our solution without executing Compute-Opt(i) again. This technique of remembering certain values is called *memorisation*. Here is now a refined algorithm, where M is an array of size $n+1$ that stores the values $OPT(j)$, where $j = 0, \ldots, n+1$.

Algorithm 2 Smart-Compute-Opt(j)-algorithm

```
If $j = 0$ then return 0
− Else if $M(j)$ is not empty then return $M(j)$
− Else define $M(j) = \max\{v_j$+Smart-Compute-Opt($p(j)$), Smart-Compute-Opt($j-1$)$\}$
− Return $M(j)$
Endif.
```

4 Running time of Smart-Compute-Opt(j)-algorithm

Let R be the sequence $r_1, \ldots, r_n$ be an input to the algorithm. We assume that these requests are ordered according to their finishing times. During the execution, accessing the value of $M(s)$ for any given s takes a constant time. If $M(s)$ is empty then Smart-Compute-Opt routine is called at most twice. The time spent on each such call, excluding the time spent on subsequent recursive calls, is a constant time as well. Thus, the running time of the algorithm is bounded by the number of calls made to Smart-Compute-Opt routine times the constant time. Now note that each call to Smart-Compute-Opt routine results in one of the entries of the array M being filled. The size of the arrays is $n+1$. Hence, the running time of the algorithm is $O(n)$.

5 Solving the problem through iteration

The main observation in designing the Smart-Compute-Opt routene was that we only have $n+1$ sub problems; hence, we need to remember the values $Opt(0), Opt(1), \ldots, Opt(i), \ldots, Opt(n)$. That is why we created the array M for memorisation of these values, and the array M became key to the efficiency of the algorithm. The array M in Smart-Compute-Opt routine is built recursively. The principle in building M is the relation

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}.$$

from Observation 1. Thus, the value $M(j)$ is determined by the values of M on positions (of M) smaller that j. Hence, instead of computing the values of the array M from top-down approach by recursion (through calls to Smart-Compute-Op routine), we can compute M using bottom-up approach iteratively without using the recursive calls:

Algorithm 3 Iterative-Compute-Opt(M)

```
$M(0) = 0$
− For $j = l, 2, \ldots, n$
− Set $M(j) = \max\{v_j + M[p(j)), M(j-1)\}$
Endfor.
```

This algorithm builds up the array starting at position 0, then 1, then 2, etc. For every j, the value of $M(j)$ is obtained in constant time, once we know the smaller values $M(j-1)$ and $M(p(j))$. Therefore the running time of this algorithm is easier to analyse, and it is clear that the running time is $O(n)$. It is also easier to argue that the algorithm computes the optimal value for the input sequence of requests.

Exercise 4. Explain why the iterative algorithm above produces the optimal solution for the input sequence r_1, …, r_n of requests.

This new view of the algorithm that is based on iteratively building the solution from the solutions of smaller sized problems is the essence of dynamic programming.

6 Finding a solution set

So far we found the value of the optimal solution for the weighted interval scheduling problem. A natural question is if we can explicitly output a set of intervals that give the optimal solution. Here we present two natural ways of outputting the optimal solution.

The first way consists of scanning through the array M, and for each j position of the array associate a list of requests $S(j)$ that are optimal for the request set $r_1, \ldots, r_j$. So, if $M(j) = v_j + M(p(j)) \geq M(j-1)$, then the list $S(j)$ consists of the list $S(p(j))$ appended with the request r_j. Otherwise, the list $S(j)$ is simply the list $S(j-1)$. If we keep the lists $S(0)$, $S(1)$,

…, $S(n)$ separately from each other, then building each list takes $O(n)$, and hence the running time complexity of such implementation will be $O(n^2)$. Instead, for each position j of the array M we can act as follows:

1. If $M(j) = v_i + M(p(i)) \geq M(j-1)$, then we insert j, using the link list data structure, into the first position of $S(p(j))$, and create a pointer from j pointing to $p(j)$ in the new list $S(p(j))$ appended with j.
2. If $M(j) = v_i + M(p(i)) < M(j-1)$, then we create a pointer pointing to the first element in the list $S(j-1)$.

Doing this for each j takes a constant amount of time. If we now want to output the entire optimal solution, we just go to the position n in the array and follow the pointers that we created in the process above. Following the pointers we output the entire optimal solution.

The second way is the following. Instead of using the iteration process as above, we can use recursion by utilising the array M. Here is the algorithm:

Algorithm 4 Find-Solution (j)

```
If $j = 0$ then output the nil list
Else
− If $v_i + M[p(j)] \geq M(j-1)$ then output $j$ followed by Find-Solution($p(j)$)
− Else output Find-Solution($j-1$)
− Endif
Endif
```

The running time analysis of this algorithm is very similar to the running time analysis of Smart-Compute-Opt(j)-algorithm. Each call for Find-Solution routine is constant time. The routine is called at most n times. Hence, the running time of this algorithm is $O(n)$.

7 Principles of dynamic programming

The algorithm that we discussed above is a good example of a dynamic algorithm. Just like for greedy or divide and conquer algorithms, it is hard to give a formal mathematical definition to the concept of dynamic programming. But we can outline some of its principles and features:

1. The original problem is reduced to solving polynomially many subproblems. These subproblems have smaller size inputs than the original problem but the sizes of the isubproblems might vary.
2. The solution to the original problem can be computed, in polynomial time, from the solution for the subproblems.
3. The subproblems are ordered by their sizes; this defines a natural order in which the subproblems should be solved. The relationship between the large problems and smaller problems is expressed via a simple recurrence relation. This recurrence relation establishes the way how solutions to larger subproblems are connected with the solutions to small subproblems.

These are just principles, and they can take various forms depending on the problem at hand. Perhaps, the important part of these principles consists of finding the relationship between small subproblems and larger subproblems expressed via the recurrence relation.

Exercise 5. Identify all these three principles in our algorithms that solve the weighted interval scheduling problem.

Exercise 6. The longest increasing subsequence problem is the following. The input is a sequence of integers $x_1, ..., x_n$. The problem asks to find the longest increasing subsequence of the input sequence. Design a dynamic algorithm that solves the problem. Analyse the running time of the algorithm.