**CSC 373 - Algorithm Design, Analysis, and Complexity** **Summer 2016**

*Lalla Mouatadid*

# Dynamic Programming: Weighted Interval Scheduling

After that "smooth" (?) transition from greedy algorithms to dynamic programming, we now formally introduce this algorithmic paradigm. As we will see in the next two weeks, dynamic programming is a powerful tool. Solving a problem using DP involves coming up with a recursive definition where sub-problems can be solved optimally and put together to construct a final optimal solution to the original problem. However, in order to use this approach, we need to make sure that the number of sub-problems we have to solve is "small": That is, polynomial in the size of the input. This way we can just cache the solutions to the sub-problems, and reuse them if needed. We also need to come up with a precise recursive definition of the problem. This is usually done in the following manner:

1. Give a formal description of what an optimal solution to the problem is.
2. Come up with a recursive structure to the problem. Define these sub-problems in a way such that a combination of their solutions will give you a solution to step 1.
3. Define an array that you will use to store the solutions to the sub-problems of step 2.
4. Give a precise recurrence relation that solves the sub-problems and whose solutions are stored in the array
5. Give an iterative bottom-up algorithm that uses the recurrence relation to compute values of the array.

The problem we will consider for this powerful technique is the weighted interval scheduling problem, which is similar to the interval scheduling problem, except now each interval has a weight $w$ and the goal is to maximize the total weight of non-intersecting intervals. However, as we saw in class, the greedy approaches we took to solve this problem can fail miserably. In particular, the EFT algorithm does not take into consideration the weight constraints; whereas the sorting by decreasing weight does not consider the non-overlapping constraint.

**Weighted Interval Scheduling** (WIS):
**Input**: A set $\mathcal{I} = \{I_1, I_2, \dots, I_n\}$ of $n$ intervals where for all $i \in [n], I_i = (s_i, f_i, w_i > 0)$.
**Output**: A subset $S \subseteq \mathcal{I}$ of non-overlapping intervals that maximizes the sum $\sum\limits_{I_i \in S} w_i$.

Following the steps above, we can try to solve the WIS problem as follows:
1. We define S to be the optimal solution of the WIS problem when we consider all n intervals.
2. A recursive definition would involve defining $S$ in terms of itself. This means we need to identify the parameter that will have to be "reduced".
Intuitively, $n$ sounds like a valid choice of parameter, since the solution $S$ can be constructed by considering all $n-1$ intervals plus the last interval. Similarly, just before considering the $n-1$ intervals, we can first consider the first $n-2$ intervals then the $(n-1)^{th}$ interval and so on. Generally speaking, a recursive definition would be to identify $S_i$ to be the optimal solution of the WIS problem when we consider only the first $i$ intervals. We also need to define our base case: $S_0 = 0$ since no interval has been considered.
3. Step 3 is now trivial as we can just let $S[0..n]$ be our array: $S[i]$ would then store the optimal solution to the sub-problem that deals with the first $i$ intervals.
4. Notice that in step 2, we have defined the sub-problems with respect to some input of size $i$. In this step, we need to precisely describe how to express $S_i$ in terms of an $S_j$ for some $j < i$. So what exactly changes when we process the first $i$ intervals as opposed to the first $i-1$ intervals? In other words, how do $S_{i-1}$ and $S_i$ relate?

Well, either we select the $i^{th}$ interval or we don't. If $I_i$ is not added to the solution, then:

$$S_i = S_{i-1} \tag{1}$$

However, if we decide to add $I_i$ to our solution set, then we need to remove all the intervals in $S_{i-1}$ that intersect with $I_i$. How many possible subsets do we need to look at? All of them? There is an exponential number of them!

Notice that up to this point, we have not really defined what the **first** $i$ intervals are, since we didn't impose any particular order on our input. As we saw in class, the ordering we are choosing from is indeed the EFT ordering. Why is it EFT? Think of what would go wrong if we had an arbitrary ordering.

In order to add $I_i$ to our solution set, we need to remove all the conflicting intervals in $S_{i-1}$. This means we need to find a solution $S_j$ where $j < i$, such that:
(i) none of the intervals in $S_j$ intersect with $I_i$, and
(ii) all of the intervals $\{I_{j+1}, I_{j+2}, \dots, I_{i-1}\}$ intersect with $I_i$ (otherwise we can find a better sub-solution) and finally
(iii) $S_j$ has to be optimal for the first $j$ intervals.

The next question is how to compute $j$. Using the EFT ordering, we just need to find the largest index $j$ such that $j < i$, and $S_j$ satisfies the 3 conditions above. Therefore, we let $p[i]$ = largest index $j < i$ such that interval $I_j$ is compatible with $I_i$. So if we decide to add $I_i$ to our solution, we have:

$$\begin{aligned} S_i &= S_j + w_i && (2)\\ &= S_{p[i]} + w_i && (3) \end{aligned}$$

Combining (1) and (3), our recurrence relation is:

$$S_i = \max(S_{i-1}, S_{p[i]} + w_i) \tag{4}$$

Now our iterative bottom-top algorithm would look as follows:

**Algorithm 1** Weighted Interval Scheduling

**Input:** A set $\mathcal{I} = \{I_1, I_2, \dots, I_n\}$ of $n$ intervals where for all $i \in [n], I_i = (s_i, f_i, w_i)$, $s_i < f_i, w_i \geq 0$.
**Output:** A subset $S \subseteq \mathcal{I}$ of non-overlapping intervals that maximizes $\sum_{I_i \in S} w_i$

```
1: Define $S[0 \dots n]$ to be an array of size $n + 1$
2: $S[0] = 0$
3: Sort $\mathcal{I}$ by EFT
4: Compute $p[i]$ for all $i \in [n]$
5: for $i = 1 \dots n$ do
6:     $S[i] = \max(S_{i-1}, S[p[i]] + w_i)$
7: end for
8: return $S[n]$
```

**Proof of optimality**: To prove optimality, we just need to show that for $i \in [n], S_i$ contains the value of an optimal solution of the first $i$ intervals. We do so using induction.

*Proof.* For $i = 0, S_0 = 0$ is optimal since no interval has been processed. Suppose the claim holds for $S_j$ for all $j < i$.

Consider $S_i$: Either $I_i$ is added to the solution or it wasn't. If $I_i$ was not added to the solution then the optimal solution for the first $i$ intervals is just the same as the optimal solution for the first $i-1$ intervals, i.e. $S_i = S_{i-1}$.

Otherwise, suppose $I_i$ is added to the solution. Then all the intervals $I_{p[i]+1}, I_{p[i]+2}, \ldots, I_{i-1}$ conflict with $I_i$, and the the remaining intervals to choose from are amongst the first $p[i]$ intervals. Therefore any optimal solution that includes $I_i$ must be a subset of $\{I_1, I_2, \ldots, I_{p[i]}, I_i\}$. Since $I_i$ does not intersect with any interval in $\{I_1, \ldots, I_{p[i]}\}$, and $S_{p[i]}$ is optimal for the first $p[i]$ intervals (by induction hypothesis), we conclude that $S_i = S_{p[i]} + w_i$.

Since $S_i$ is the maximization of these two cases, the larger value out of these two scenarios is the value of $S_i$.

$\square$