CMSC 451: Lecture 10
Dynamic Programming: Weighted Interval Scheduling
Tuesday, Oct 3, 2017

Reading: Section 6.1 in KT.

Dynamic Programming: In this lecture we begin our coverage of an important algorithm design technique, called *dynamic programming* (or *DP* for short). The technique is among the most powerful for designing algorithms for optimization problems. Dynamic programming is a powerful technique for solving optimization problems that have certain well-defined clean structural properties. (The meaning of this will become clearer once we have seen a few examples.)

There is a superficial resemblance to divide-and-conquer, in the sense that it breaks problems down into smaller subproblems, which can be solved recursively. However, unlike divide-and-conquer problems, in which the subproblems are disjoint, in dynamic programming the subproblems typically overlap each other, and this renders straightforward recursive solutions inefficient.

Dynamic programming solutions rely on two important structural qualities, *optimal substructure* and *overlapping subproblems*.

Optimal substructure: This property (sometimes called the *principle of optimality*) states that for the global problem to be solved optimally, each subproblem should be solved optimally. While this might seem intuitively obvious, not all optimization problems satisfy this property. For example, it may be advantageous to solve one subproblem suboptimally in order to conserve resources so that another, more critical, subproblem can be solved optimally.

Overlapping Subproblems: While it may be possible subdivide a problem into subproblems in exponentially many different ways, these subproblems overlap each other in such a way that the number of distinct subproblems is reasonably small, ideally *polynomial* in the input size.

An important issue is how to generate the solutions to these subproblems. There are two complementary (but essentially equivalent) ways of viewing how a solution is constructed:

Top-Down: A solution to a DP problem is expressed recursively. This approach applies recursion directly to solve the problem. However, due to the overlapping nature of the subproblems, the same recursive call is often made many times. An approach, called *memoization*, records the results of recursive calls, so that subsequent calls to a previously solved subproblem are handled by table look-up.

Bottom-up: Although the problem is formulated recursively, the solution is built iteratively by combining the solutions to small subproblems to obtain the solution to larger subproblems. The results are stored in a table.

In the next few lectures, we will consider a number of examples, which will help make these concepts more concrete.

Weighted Interval Scheduling: Let us consider a variant of a problem that we have seen before, the Interval Scheduling Problem. Recall that in the original (unweighted) version we are given a set $S = \{1, \ldots, n\}$ of n *activity requests*, where each activity is expressed as an interval $[s_i, f_i]$ from a given start time s_i to a given finish time f_i. The objective is to compute any maximum sized subset of non-overlapping intervals (see Fig. 1(a)).

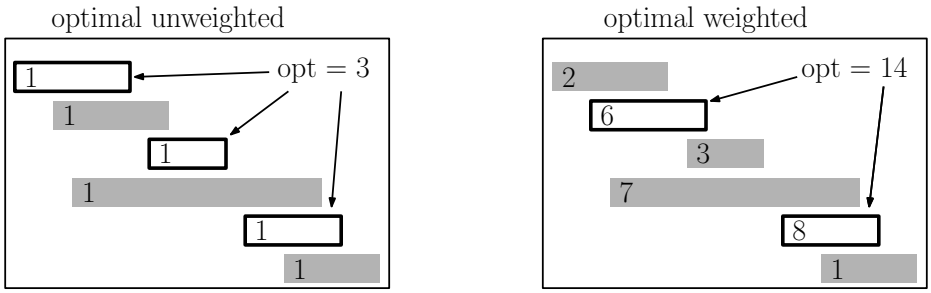

Fig. 1: Weighted and unweighted interval scheduling.

In *weighted interval scheduling*, we assume that in addition to the start and finish times, each request is associated with a numeric *weight* or *value*, call it v_i, and the objective is to find a set of non-overlapping requests such that sum of values of the scheduled requests is maximum (see Fig. 1(b)). The unweighted version can be thought of as a special case in which all weights are equal to 1. Although a greedy approach works fine for the unweighted problem, no greedy solution is known for the weighted version. We will demonstrate a method based on dynamic programming.

Recursive Formulation: Dynamic programming solutions are based on a decomposition of a problem into smaller subproblems. Let us consider how to do this for the weighted interval scheduling problem. As we did in the greedy algorithm, it will be convenient to sort the requests in nondecreasing order of finish time, so that $f_1 \leq \ldots \leq f_n$.

Here is the idea behind DP in a nutshell. Consider the *last* request $[s_n, f_n]$. There are two possibilities. If this request *is not* in the optimum schedule, then we can safely ignore it, and recursively compute the optimum solution of the first $n-1$ requests. Otherwise, this request *is* in the optimal solution. We will schedule this request (receiving the profit of v_n) and then we must eliminate all the requests whose intervals overlap this one. Because requests have been sorted by finish time, this involves finding the largest index p such that $f_p < s_n$. Thus, we solve the problem recursively on the first p requests.

But we don't know the optimum solution, so how can we select among these two options? The answer is that we will compute the cost of both of them recursively, and take the better of the two.

Let's now implement this idea. Recall that the requests are sorted by finish times. For the sake of generality, let's assume that we want to solve the problem on requests 1 through j, where $0 \leq j \leq n$. If $j = 0$, there is nothing to do. Otherwise, given any request j, define $p(j)$ to be the largest integer such that $f_{p(j)} < s_j$. Thus, request 1 through $p(j)$ do not overlap request j. If no such i exists, let $p(j) = 0$ (see Fig. 2).

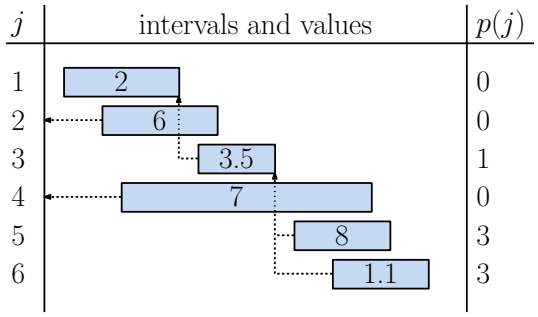

Fig. 2: Weighted interval scheduling input and p-values.

For now, let's just concentrate on computing the *optimum total value*. Later we will consider how to determine which *requests* produce this value. A natural idea would be to define a function that gives the optimum value for just the first i requests.

Definition: For $0 \leq j \leq n$, $\text{opt}(j)$ denotes the maximum possible value achievable if we consider just tasks $\{1, \ldots, j\}$ (assuming that tasks are given in order of finish time).

As a basis case, observe that if we have no tasks, then we have no value. Therefore, $\text{opt}(0) = 0$. If we can compute $\text{opt}(j)$ for each value of j, then clearly, the final desired result will be the maximum value using *all* the requests, that is, $\text{opt}(n)$.

Summarizing our earlier observations, in order to compute $\text{opt}(j)$ for an arbitrary j, we observe that there are two possibilities:

Request j is not in the optimal schedule: If j is not included in the schedule, then we should do the best we can with the remaining $j - 1$ requests. Therefore, $\text{opt}(j) = \text{opt}(j-1)$.

Request j is in the optimal schedule: If we add request j to the schedule, then we gain v_j units of value, but we are now limited as to which other requests we can take. We cannot take any of the requests following $p(j)$. Thus we have $\text{opt}(j) = v_j + \text{opt}(p(j))$.

How do we know which of the these two options to select? The danger in trying to be too smart (e.g., trying a greedy choice) is that the choice may not be correct. Instead, the best approach is often simple brute force:

DP Selection Principle:
When given a set of feasible options to choose from, try them *all* and take the *best*.

In this cases there are two options, which suggests the following recursive rule:

$$\text{opt}(j) \;=\; \max \left\{ \begin{array}{ll} \text{opt}(j-1) & (\text{request } j \text{ is not in opt}) \\ v_j + \text{opt}(p(j)) & (\text{request } j \text{ is in opt}) \end{array} \right\}.$$

We could express this in pseudocode as shown in the following code block:

Recursive Weighted Interval Scheduling

```
recursive-WIS(j) {
    if (j == 0) return 0
    else return max( recursive-WIS(j-1), v[j] + recursive-WIS(p[j]) )
}
```

I have left it as self-evident that this simple recursive procedure is correct. Indeed the only subtlety is the inductive observation that, in order to compute opt(j) optimally, the two subproblems that are used to make the final result opt$(j-1)$ and opt$(p(j))$ should also be computed optimally. This is an example of the principle of optimality, which in this case is clear.[1]

Memoized Formulation: The principal problem with this elegant and simple recursive procedure is that it has a *horrendous* running time. To make this concrete, let us suppose that $p(j) = j - 2$ for all j. Let $T(j)$ be the number of recursive function calls to opt(0) that result from a single call to opt(j). Clearly, $T(0) = 1$, $T(1) = T(0)+T(0)$, and for $j \geq 2$, $T(j) = T(j-1)+T(j-2)$. If you start expanding this recurrence, you will find that the resulting series is essentially a Fibonacci series:

j	0	1	2	3	4	5	6	7	8	...	20	30	50
$T(j)$	1	2	3	5	8	13	21	34	55	...	17,711	2,178,309	32,951,280,099

It is well known that the Fibonacci series $F(j)$ grows exponentially as a function of j.[2] This may seem ludicrous. (And it is!) Why should it take 32 billion recursive calls to fill in a table with just 50 entries? If you look at the recursion tree, the reason jumps out immediately (see Fig 3). The problem is that the same recursive calls are being generated over and over again. But there is no reason to make even a second call this, since they all return exactly the same value.

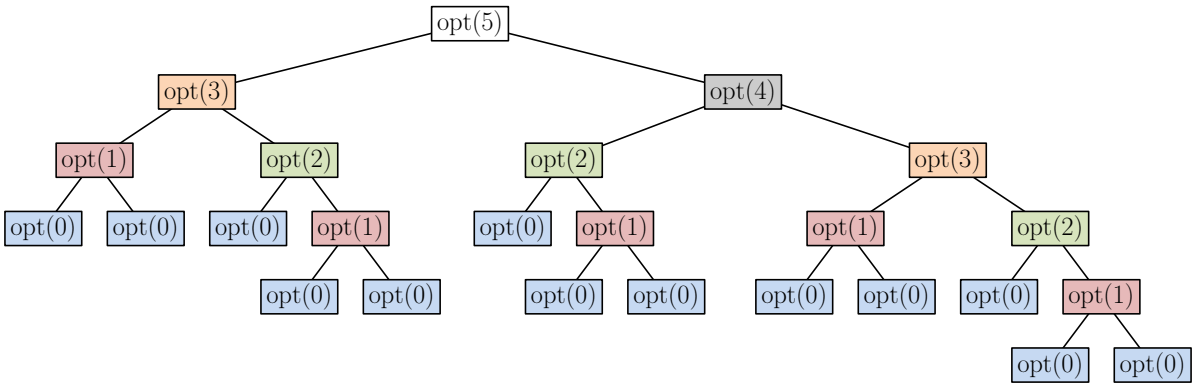

Fig. 3: The exponential nature of recursive-WIS.

[1] Why would you want to solve a subproblem suboptimally? Suppose, that you had the additional constraint that the final schedule can only contain 23 intervals. Now, it might be advantageous to solve a subproblem suboptimally, so that you have a few extra intervals left over to use at a later time.

[2] The nth Fibonacci number is roughly grows as $\Theta(\phi^n)$, where $\phi \approx 1.618$ is the celebrated Golden Ratio.

This suggests a smarter version of the algorithm. Once a value has been computed for recursive-WIS(j) we store the value in a global array $M[1..n]$, and all future attempts to compute this value will simply access the array, rather than making a recursive call. This technique is called *memoizing* (I guess because we are making a memo to ourselves of what the value is for future reference). You might imagine that we initialize all the $M[j]$ entries to -1 initially, and use this special value to determine whether an entry has already been computed.

We will add one additional piece of information, which will help in reconstructing the final schedule. Whenever a choice is made between two options, we'll save a *predecessor pointer*, pred[j], which reminds of which choice we made. (Its value is either $j-1$ or $p(j)$, depending on which recursive call was made). The resulting algorithm is presented in the following code block.

Memoized Weighted Interval Scheduling

```
memoized-WIS(j) {
    if (j == 0) return 0                          // basis case - no requests
    else if (M[j] has been computed) return M[j]
    else {
        leaveWeight = memoized-WIS(j-1)           // total weight if we leave j
        takeWeight = v[j] + memoized-WIS(p[j])    // total weight if we take j
        if ( leaveWeight > takeWeight ) {
            M[j] = leaveWeight                    // better to leave j
            pred[j] = j-1                         // previous is j-1
        } else {
            M[j] = takeWeight                     // better to take j
            pred[j] = p[j]                        // previous is p[j]
        }
        return M[j]                               // return final weight
    }
}
```

The memoized version runs in $O(n)$ time. To see this, observe that each invocation of memoized-WIS either returns in $O(1)$ time (with no recursive calls), or it computes one new entry of M. Since there are n entries in the table, the latter can occur at most n times.

Bottom-up Construction: (Optional) Yet another method for computing the values of the array, is to dispense with the recursion altogether, and simply fill the table up, one entry at a time. We need to be careful that this is done in such an order that each time we access the array, the entry being accessed is already defined. This is easy here, because we can just compute values in increasing order of j. As before, we include the computation of the predecessor values.

Do you think that you understand the algorithm now? If so, answer the following question. Would the algorithm be correct if, rather than sorting the requests by finish time, we had instead sorted them by start time? How about if we didn't sort them at all?

Computing the Final Schedule: So far we have seen how to compute the value of the optimal schedule, but how do we compute the schedule itself? This is a common problem that arises

Bottom-Up Weighted Interval Scheduling

```
bottom-up-WIS() {
    M[0] = 0
    for (i = 1 to n) {
        leaveWeight = M[j-1]                    // total weight if we leave j
        takeWeight = v[j] + M[p[j]]             // total weight if we take j
        if ( leaveWeight > takeWeight ) {
            M[j] = leaveWeight                  // better to leave j
            pred[j] = j-1                       // previous is j-1
        }
        else {
            M[j] = takeWeight                   // better to take j
            pred[j] = p[j]                      // previous is p[j]
        }
    }
}
```

in many DP problems, since most DP formulations focus on computing the numeric optimal value, without consideration of the object that gives rise to this value. The solution is to leave ourselves a few hints in order to reconstruct the final result.

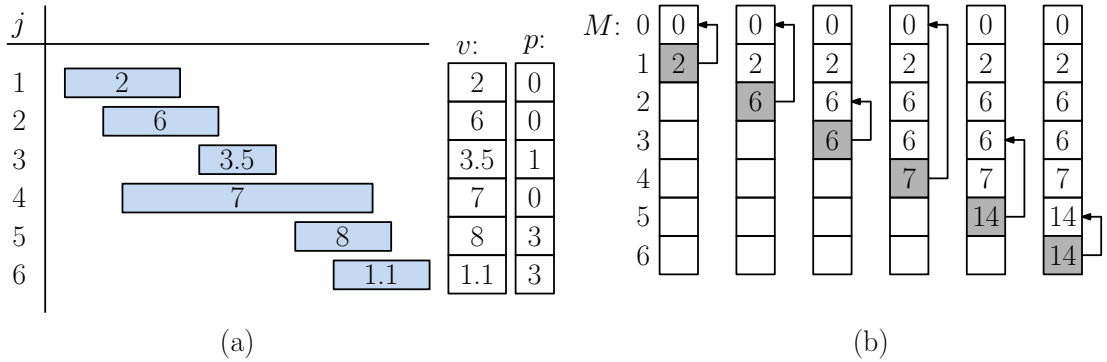

Fig. 4: (a) The input intervals and p values, (b) the bottom-up construction of the table and predecessor values, (c) the final predecessor values.

In bottom-up-WIS() we did exactly this. We know that value of $M[j]$ arose from two distinct possibilities, either (1) we didn't take j and just used the result of $M[j-1]$, or (2) we did take j, added its value v_j, and used $M[p(j)]$ to complete the result. To remind us of how we obtained the best choice for $M[j]$ was to store a predecessor pointer pred$[j]$.

In order to generate the final schedule, we start with $M[n]$ and work backwards. In general, if we arrive at $M[j]$, we check whether pred$[j] = p[j]$. If so, we can surmise that we did used the jth request, and we continue with pred$[j] = p[j]$. If not, then we know that we did not include request j in the schedule, and we then follow the predecessor link to continue with pred$[j] = j - 1$. The algorithm for generating the schedule is given in the code block below.

The computation of the final schedule is illustrated in Fig. 5 (where predecessor values are shown as arrows).

Computing Weighted Interval Scheduling Schedule

```
get-schedule() {
    j = n                                          // start with the last request
    sched = empty
    while (j > 0) {
        if (pred[j] == p[j]) {                     // take request j
            prepend j to the front of sched
        }
        j = pred[j]                                // continue with j's predecessor
    }
    return sched                                   // return the final schedule
}
```

- Since pred[6] $= 5 \neq p[6]$, we do *not* use request 6, and we continued with pred[6] $= 5$.
- Since pred[5] $= 3 = p[5]$, we *use* request 5, and we continue with pred[5] $= 3$.
- Since pred[3] $= 2 \neq p[6]$, we do *not* use request 3, and we continued with pred[3] $= 2$.
- Since pred[2] $= 0 = p[2]$, we *use* request 2, and we continue with pred[2] $= 0$, and terminate.

We obtain the final list $\langle 5, 2 \rangle$.

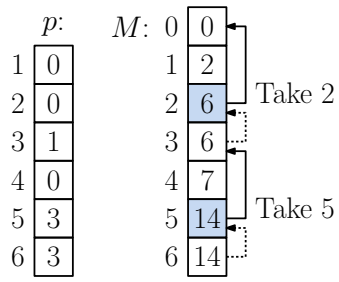

Fig. 5: Following the predecessor links to compute the final schedule.