Math 416

Worksheet 18. Dynamic Programming I

This is the final principle of algorithm design that we will study. It resembles divide-and-conquer in the sense that we will divide problems into subproblems, but the subproblems will often be *overlapping* unlike in D&C problems. And we will do a careful search through the solution space.

Weighted interval-scheduling The input is a list of n intervals $1, 2, \ldots, n$, each with a start time $s(i)$ and a finish time $f(i)$ and (this is the new part) a *weight* w_i.

Our goal is to find a set of nonoverlapping intervals S maximizing $\sum_{i \in S} w_i$.

Problem 1. Explain how the original interval-scheduling problem (for which the *earliest-finish-time-first* greedy solution worked) is a special case of this problem.

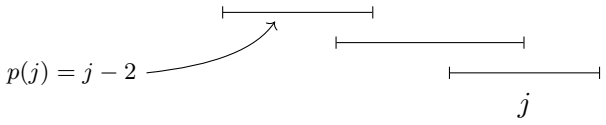
Problem 2. Give an example of a weighted interval-scheduling problem for which our interval-scheduling greedy algorithm (*earliest finish time first*) does not give the optimal solution.

In fact, no greedy algorithm is known to solve this problem. So we need a different method.

Suppose that the intervals are sorted by finish time:

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

Let $p(j)$ be the largest i such that i and j are disjoint intervals (i.e., $f(i) < s(j)$).



Problem 3. Suppose that O is an optimal solution to the problem. There are two cases. Explain the statements after the if-thens below.

if $n \in O$ **then:** the next interval to the left of n that can belong to O is $p(n)$; i.e., $p(n)+1, p(n)+2, \ldots, n-1 \notin O$; AND $O \backslash \{n\}$ is an optimal solution to the problem with intervals $1, 2, \ldots, p(n)$.

if $n \notin O$ **then:** O is an optimal solution for the intervals $1, 2, \ldots, n-1$.

Let $\text{opt}(j)$ be the value $\sum w_i$ of an optimal solution O_j to the problem with intervals $1, 2, \ldots, j$.

Problem 4. Use the previous problem to fill in the blanks:

$$\text{opt}(j) = \begin{cases} \boxed{} & \text{if } j \in O_j \\ \boxed{} & \text{if } j \notin O_j \end{cases}.$$

So:

$$\text{opt}(j) = \max(\boxed{}, \boxed{}), \qquad (\star)$$

and

$$j \text{ is in an optimal solution iff } \boxed{} \geq \boxed{}.$$

This suggests a recursive algorithm: compute the quantity in ($\star$) recursively. Doing this naively results in an algorithm that takes exponential time, though, since we're repeating the same computation over and over. To fix this, we use an array to store previously found solutions: this trick is called **memoization**.

Problem 5. Describe the tree of calls a naïve recursive algorithm would make to solve the problem in Figure 1. Explain why this suggests an exponential number of calls in the worst case.

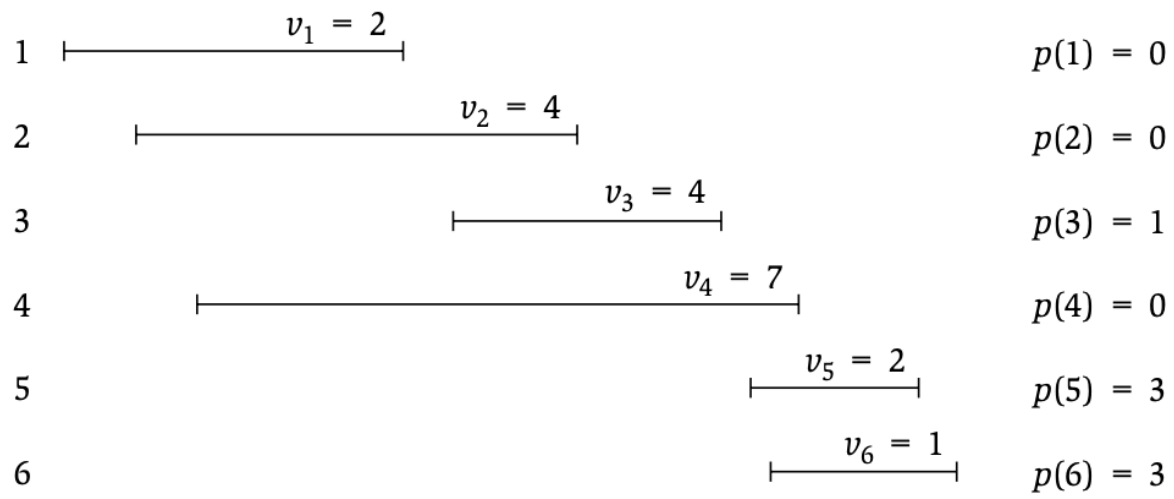


FIGURE 1. weighted interval scheduling example

Algorithm 1: weighted interval scheduling

```
compute-opt(n):
    Input: intervals 1, 2, ..., n, sorted by finish time, with weights w_i
    Output: array M[0 ··· n], M[j] = opt(j)
    set M[0] = 0 ;
    foreach j = 1, ..., n do
        M[j] = max(w_j + M[p(j)], M[j − 1]) ;
```

Problem 6. Now run the memoized algorithm below on the input in Figure 1. Record the array M at every stage of the algorithm.

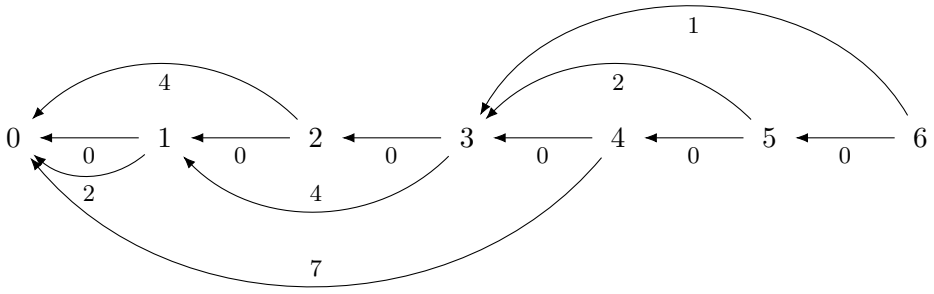
Problem 7. Explain how you would write a formal proof of correctness.
(*Hint:* By induction, using . . . ?)

Assuming that the intervals are presorted by finish time, this algorithm can be implemented to run in $O(n)$ time.

Problem 8. Make sure you and your group members understand why the algorithm can be implemented to run in linear time, assuming the tasks are pre-sorted by finish time and the values $p(j)$ are pre-computed.

Dynamic Programming principle. To solve problems, we solve subproblems. There is an ordering (implicit DAG) on the subproblems and a relation (e.g. ($\star$)) that can be used to solve a subproblem given solutions to previous (in the ordering) subproblems.

Problem 9. Explain a sense in which the DAG below represents the weighted-interval-scheduling problem from Figure 1. Then find a solution by finding a max-weight path from 6 to 0 in this DAG.



Optimal paths in weighted DAGs The previous discussion suggests that we just address directly the problem of finding max-weight or min-weight paths in weighted DAGs. (Remember that a DAG is Directed Acyclic Graph.) The trouble now is that we allow weights to be negative!

Example 1. In Figure 2 are pictured a weighted DAG and one of its linearizations.

FIGURE 2. weighted DAG example

Notice that the only way to get from S to C is through A or B. So

$$\text{dist}(C) = \min(\text{dist}(A) + 2, \text{dist}(B) + 3).$$

Computing distances left-to-right in the topological ordering, we always have the information we need to compute the distance using distances to previous vertices. So we can compute the distance from S to every vertex in a single pass!

Algorithm 2: smallest distance

```
smallest-dist(G, s):
    Input: weighted DAG G with a source vertex s and a topological ordering
    Output: an array dist containing the distance from s to every vertex in G
    set all dist(v) = ∞ ;
    set dist(s) = 0 ;
    foreach v ∈ V \ {s} considered in the given topological ordering do
        set dist(v) = min_{(u,v)∈E} (dist(u) + wt(u, v));
```

Problem 10. What changes need to be made to the `smallest-dist` algorithm to make it find the largest distance (i.e., max weight of a path) instead of the smallest?

Problem 11. Below are pictured a DAG and one of its topological orderings. Run our algorithm to find both a max-weight ('longest') and a min-weight ('shortest') path from S to E.