CSE347 Class 4

Jeremy Buhler

September 11, 2017

1 Weighted Scheduling is Different

Here's the problem we posed at the end of last time.

- Given a set P of projects p_i, each occupying half-open time interval $[s_i, f_i)$ ...
- *and* a weight w_i for each project p_i,
- Choose a subset $\Pi \subseteq P$ of projects for which
  - No two projects' intervals overlap ("conflict");
  - The *total weight* of projects in Π is maximized.
- The greedy algorithm from last time ignores weights, so it can't possibly work.
- Unfortunately, we don't know of *any* greedy algorithm that yields an optimal solution for this problem.

We need yet another new structural approach.

- We can't define a *single*, greedy choice that always leads us to an optimal solution.
- But maybe we can define a *set* of such choices, *at least one* of which is part of some optimal solution.
- If we come up with the best solution possible for each choice, the best among all these solutions must be globally optimal.
- *Example*: sort the projects P in *some* order $p_1 \ldots p_n$.
- Either p_n is part of some optimal solution to P, or it isn't. Let Π^* be this (unknown) optimal solution.
- Π^* consists of the result of our first choice (i.e., either p_n or nothing), combined with some subset Π' of the remaining projects.
- If p_n *is* part Π^*, then Π' must consist only of projects in P that do not conflict with P; that is,
$$\Pi' \subseteq P' = \{p \in P \mid p \cap p_n = \emptyset\}.$$
- If instead p_n is *not* part Π^*, then Π' is a subset of the remaining projects; that is,
$$\Pi' \subseteq P' = P - \{p_n\}.$$

- In either case, if we are trying to maximize the weight of Π^*, we should (recursively) find a solution Π' to P' of the greatest possible weight!
- Suppose this weight is W'.
- If we do use p_n, the total weight of Π^* is $W' + w_n$.
- If we don't use p_n, the total weight of Π^* is $W' + 0$.
- A globally optimal solution either uses p_n or not, so if we compute the best possible solutions for each choice, the better of them is globally optimal.

This sounds suspiciously like a recursive algorithm!

```
MaxWeight($P = \{p_1 \dots p_n\}$) $\triangleright$ returns a pair (solution, value)
    if $P = \emptyset$
        return $(\emptyset, 0)$
    $(\Pi_a, w_a) \leftarrow$ MaxWeight($P - \{p_n\}$)
    $(\Pi_b, w_b) \leftarrow$ MaxWeight($\{p \in P \mid p \cap p_n = \emptyset\}$)
    if $w_a + 0 > w_b + w_n$
        return $(\Pi_a, w_a)$
    else
        return $(\Pi_b \cup \{p_n\}, w_b + w_n)$
```

- To prove that this algorithm is correct, we need to prove three things.
    1. *The set of choices is complete.* One of the choices is part of some optimal solution to P.
    2. *The subproblem in each case is a smaller, unconstrained instance of the original problem*; that is, we can recursively solve the subproblem, and any solution to it may feasibly be combined with our first choice.
    3. For each initial choice, *The value of a solution is separable* into a sum (or product) of terms, one of which depends only on that choice, and one of which depends only on the remaining subproblem.
- Let's check these three points for our algorithm.
- First, an optimal solution must use p_n or not, so our two choices clearly form a complete set for this problem.
- Second, consider the (clearly smaller) set of projects left after the first choice. If we do not use p_n, then any feasible subset of the remaining P' is fine. If we do use p_n, then P' consists of only those projects that do not conflict with p_n, so any solution to P' can feasibly be combined with p_n.
- Third, the cost of the final solution is separable into a sum of two terms: one that depends only on the first choice (0 or w_n, respectively), and one that depends only on the remaining subproblem given that choice (w_a or w_b, respectively).

Why are these three properties the right ones to prove?

- **Claim**: MaxWeight returns an optimal solution to P.
- **Pf**: by induction on $|P|$.
- **Bas**: if $|P| = 0$, then the algorithm is correct by inspection (it returns the only possible solution, namely $\emptyset$).
- **Ind**: if $|P| > 0$, then by property #2, we may assume by the IH that MaxWeight returns optimal solutions Π_a and Π_b respectively to the two recursive subprobs, and each solution can be feasibly combined with its respective first choice.
- Now by property #1, there exists an optimal solution Π^* that makes one of the choices made by the algorithm – use p_n, or don't. Proceed by cases.
- Suppose first that Π^* *does* use p_n. By property #3, we can separate the weight of Π^* into a sum $w_n + w'$, where w' is the weight of the solution exclusive of p_n.
- Let Π_b^* be the solution returned by our algorithm when it does use p_n.
- Suppose Π_b^* is *not* optimal; then we have that

$$\begin{aligned} \text{weight}(\Pi_b^*) &< \text{weight}(\Pi^*) \\ w_b + w_n &< w' + w_n \\ w_b &< w'. \end{aligned}$$

- But this contradicts our assumption that Π_b is an optimal solution to its subproblem $\rightarrow\leftarrow$. Hence, Π_b^* must be optimal.
- Similarly, suppose that Π^* *does not* use p_n. By property #3, we can separate the weight of Π^* into a sum $0 + w'$.
- Let Π_a^* be the solution returned by our algorithm when it does not use p_n.
- Suppose Π_a^* is *not* optimal; then we have that

$$\begin{aligned} \text{weight}(Pi_a^*) &< \text{weight}(\Pi^*) \\ w_a + 0 &< w' + 0 \\ w_a &< w'. \end{aligned}$$

- But this contradicts our assumption that Π_a is an optimal solution to its subproblem $\rightarrow\leftarrow$. Hence, Π_a^* must be optimal.
- Since MaxWeight computes an optimal solution among all those making each possible first choice, and one of these choices is made by some *globally* optimal solution, the best solution over all first choices must be a global optimum. QED

Whew! That was exhausting (and exhaustive).

- In practice, you can prove the correctness of your recursive algorithm by checking the three properties, *and* arguing that in each case, you have correctly computed the value of a solution from the value of its parts.

- The rest of the proof is completely mechanical.
- These three properties have names:
    1. the “complete choice” property
    2. the “inductive structure” property
    3. the “optimal substructure” property
- A bit less formally, check that
    1. some optimal solution makes at least one of your choices;
    2. each choice yields a smaller version of the same problem whose solution can always be feasibly combined with that choice;
    3. you have correctly formulated the value of an optimal solution for each choice in terms of the subproblem’s value, and that formula is separable.

2 Improving Efficiency

We now have a correct recursive algorithm. What does it cost?

- Well, at the top level, we have two recursive calls.
- If the top level problem has size n, then the recursive calls are of size up to $n-1$.
- We do $O(1)$ work other than these calls at top level.
- Hence, the relevant worst-case recurrence is

$$T(n) = 2T(n-1) + O(1),$$

  which unfortunately implies $T(n) = \Theta(2^n)$.
- Oh dear. Was all our work for naught?

Key idea: *how many distinct subproblems occur across all our recursive calls?*

- Well, there are 2^n possible subsets of n projects.
- But the number of *distinct* problems we have to consider depends on how we order these projects.
- In particular, suppose we order $p_1 \ldots p_n$ in non-decreasing order by finishing time.
- If we use p_n, it conflicts with every project that ends after s_n.
- Hence, the remaining subproblem is the prefix $p_1 \ldots p_{c(n)}$ of the original list, where

$$c(n) = \max\{i \mid f_i \leq s_n\}.$$

- If we do *not* use p_n, then the remaining subproblem is the prefix $p_1 \ldots p_{n-1}$.
- Either way, *our subproblem is always a prefix of the original list of projects.*

- A list of n projects has only $n+1$ possible prefixes (including the empty one), so we solve at most $O(n)$ *distinct* subproblems!
- So by the pigeonhole principle, it must be that our $\Theta(2^n)$ recursive calls solve the same subproblems over and over again.

How can we exploit this redundancy to improve our running time?

- **Principle**: solve each possible subproblem only once!
- One way to do this is to *memoize* the algorithm.
- Create a table T with $n+1$ entries. Entry $T[i]$ will hold the pair (Π_i, w_i) – the solution to the subproblem $P_i = \{p_1 \ldots p_i\}$, along with its value.
- Run the recursive algorithm as normal, but …
- Whenever we are about to make a recursive call on subproblem P_i, check if $T[i]$ has been computed yet.
- If so, use it; if not, solve P_i recursively and save result to $T[i]$.
- This way, we fill in each entry of T at most once, and we make recursive calls only when an entry of T has not yet been filled.

This solution is workable, but it's actually kind of dumb.

- The recursive algorithm probes T in some complicated order.
- In fact, when do we have enough information to fill in $T[i]$?
- That is, when can we compute an optimal solution for $P_i = \{p_1 \ldots p_i\}$?
- Exactly when we have solved the two subproblems on which it depends, namely:
  - P_{i-1} (if we don't use p_i)
  - $P_{c(i)}$ (if we do use p_i)
- In other words, we can compute $T[i]$ once we know $T[i-1]$ and $T[c(i)]$.
- We trivially know $T[0]$, the "base case" solution to an empty set.
- Hence, *we can compute the solutions to all subproblems in one pass in increasing order of prefix length.*
- The last thing we compute, $T[n]$, is the solution to the whole problem!
- Borrowing the code to optimize over cases from the recursive procedure we wind up with the following algorithm:

```
MaxWeightDP($P$)                ▷ returns a pair (solution, value)
    sort $P$ by finishing time into $p_1 \ldots p_n$.
    $T[0] \leftarrow (\emptyset, 0)$
    for $i = 1..n$ do
        $(\Pi_a, w_a) \leftarrow T[i-1]$
        $(\Pi_b, w_b) \leftarrow T[c(i)]$
        if $(w_a + 0 > w_b + w_i)$
            $T[i] \leftarrow (\Pi_a, w_a)$
        else
            $T[i] \leftarrow (\Pi_b \cup \{p_i\}, w_b + w_i)$
    return $T[n]$
```

Now what does our algorithm cost?

- First, we have to sort the n projects by finishing time, in total time $O(n \log n)$.
- Second, we have to fill in $O(n)$ entries in T.
- For each entry $T[i]$, we need to look up two subproblems.
- $T[i-1]$ can be looked up in time $O(1)$ given i.
- What about $T[c(i)]$? We need to compute $c(i)$ first.
- We can do this naively in time $O(n)$ by walking backwards through the project list P_i to locate the last project that ends before p_i starts.
- But if we keep a sorted array of all the finishing times f_j, how fast can we find the largest j such that $f_j \leq s_i$?
- Using binary search, we can do it in time $O(\log n)$!
- Once we've looked up these two subproblems, we do $O(1)$ work to determine which one we "pull" from to compute $T[i]$.
- Finally, we need to write the new subsolution and its value into $T[i]$.
- Writing down each subsolution takes time $O(n)$, since it may contain up to n projects.
- Conclude that we need $O(n \log n)$ sorting time, plus $O(n)$ subproblem computations, each in time $O(n)$.
- Hence, the whole algorithm takes time $O(n^2)$.

Can we solve the problem even faster?

- The limiting factor is that it takes $O(n)$ time to write down each of the $O(n)$ subsolutions.
- But in fact, the solution in $T[i]$ simply combines a smaller solution and a local choice.
- We can avoid copying solutions around by *first* computing *only* the weights in T, *then* going back and reconstructing the solution that yields that weight.

- The code looks like this:

```
MaxWeightDPTB($P$)          ▷ returns a pair (solution, value)
    sort $P$ by finishing time into $p_1 \dots p_n$.
    $T[0] \leftarrow 0$
    for $i = 1..n$ do
        $w_a \leftarrow T[i-1]$
        $w_b \leftarrow T[c(i)]$
        if $(w_a + 0 > w_b + w_i)$
            $T[i] \leftarrow w_a$
        else
            $T[i] \leftarrow w_b + w_i$

    $\Pi \leftarrow \emptyset$
    $j \leftarrow n$
    while $j > 0$ do
        if $T[j] = T[j-1]$                  ▷ did not use $p_j$
            $j \leftarrow j - 1$
        else                                ▷ did use $p_j$
            $\Pi \leftarrow \Pi \cup \{p_j\}$
            $j \leftarrow c(j)$
    return $(\Pi, T[n])$
```

- The second loop is called a "traceback" – it traces a path back through the set of choices that led to the optimum weight.
- The length of the traceback is at most $O(n)$, since Π contains at most n projects.
- We can store Π as a linked list, so adding each project takes time $O(1)$.
- Hence, traceback is $O(n)$.
- The total cost of the algorithm is now only $O(n \log n)$.

3 General Philosophy

- The general approach we used here is called *dynamic programming.*
- It involves a few steps.
- **First**, develop a correct recursive algorithm by defining a set of initial choices, one of which must be consistent with optimality.
- Make sure your choice set satisfies the three correctness properties, so that you have a proof that it works.
- Moreover, your choices should imply only polynomially many *distinct* subproblems in the input size, *over the entire tree of recursive calls*, even though the naive recursive algorithm may take exponential time.

- **Second**, find a total ordering of the subproblems so that, when you need to compute the solution to a subproblem, you have already computed the solutions to its *dependencies* – the smaller subproblems that correspond to each possible first choice.
- Your recursive algorithm (sometimes called a "DP recurrence") should tell you how to compute the solution to a subproblem from its dependencies. *Justify this computation for each first choice as part of your correctness proof.*
- **Third**, write an iterative algorithm that computes and stores the *value* (or weight, or cost) of each subproblem in terms of the values of its dependencies.
- Be sure to *justify that your iteration order computes all dependencies before they are needed.*
- The cost of the iterative algorithm will be the number of distinct subproblems times the cost to compute each value from its dependencies, plus any overhead for setting up the ordering.
- **Finally**, write a traceback procedure to reconstruct an optimal solution from the table of subproblem values.
- At each step of traceback, determine which choice was made by your optimal solution, and add the effect of this choice to the solution you are building up.

Final note: you can omit the recursive implementation altogether and just use the three properties plus the dependency structure to form an inductive correctness proof for the DP algorithm directly.