CSE541 Class 2

Jeremy Buhler

January 16, 2019

1 A Classic Problem and a Greedy Approach

A classic problem for which one might want to apply a greedy algo is *knapsack*.

- Given: a knapsack of capacity M, and n items.
- Item i has weight $w_i > 0$, value $v_i > 0$.
- *Problem*: choose contents for the knapsack so that the total weight is at most M and total value is maximized.
- (To make this interesting, we assume that $\sum_i w_i > M$, so we cannot choose everything.)
- Many versions of this problem exist, but let's look at two.

First variant: *fractional knapsack*

- Can take any real-valued amount up to w_i of item i.
- **Examples**: gold dust, gasoline, cocaine ...
- (*Further reading*: `http://what-if.xkcd.com/108/`)
- Could also model return on activities, e.g. time spent coding vs time spent writing grants vs time spent reading papers.
- Suggestions? (**Wait**)
- *Intuition*: to maximize value, we want to take items with greatest "value density."
- Define $d_i = \frac{v_i}{w_i}$.
- Density measures "bang for buck" from taking a fixed amount of a given item.

OK, so let's design a (greedy) algorithm.

- Sort items in decreasing order of value density.
- Initial weight of knapsack is 0 (empty).
- For each item i in this order, add item to knapsack until it is used up or until total weight reaches M.
- Cost is trivially $O(n \log n)$. Is it correct?
- Let's formulate and prove our key properties!

2 Proof that Fractional Knapsack is Optimal

- **Greedy Choice**: Consider a knapsack instance P, and let item 1 be item of highest value density. Then there exists an optimal solution to P that uses as much of item 1 as possible (that is, $\min(w_1, M)$).
- **Pf**: suppose we have a solution Π that uses weight $w < \min(w_1, M)$ of item 1. Let $w' = \min(w_1, M) - w$.
- Π must contain at least weight w' of some other item(s), since it never pays to leave the knapsack partly empty.
- Construct Π^* from Π by removing w' worth of other items and replacing with w' worth of item 1.
- Because item 1 has max value density, Π^* has total value at least as big as Π. QED

One down, two to go.

- **Inductive Structure**: after making greedy first choice for P, we are left with a smaller problem instance P' with no external constraints.
- **Pf**: We are left with a knapsack of capacity $M' < M$ (possibly 0) and a collection of remaining items to fill it. Any feasible knapsack for P' may be combined with the remaining weight $M - M'$ of item 1 to form a feasible knapsack for P.

Two down, one to go.

- **Optimal Substructure**: optimal solution to P' combined with greedy choice yields optimal solution to P.
- **Pf**: suppose we find optimal solution Π' for P' and combine with greedy choice to get solution Π to P. Let $\hat{v} \leq v_1$ be value associated with initial greedy choice.
- Then $\text{value}(\Pi) = \text{value}(\Pi') + \hat{v}$.
- If Π is not optimal, let Π^* be an optimal solution that makes greedy choice, i.e. uses as much of item 1 as possible. Remainder of knapsack after this item is removed has value greater than $\text{value}(\Pi')$, which is impossible. QED
- **Note**: that last bit of argument gets repetitive to write. It is enough for your homework to recognize that

  $$\text{value}(\Pi) = \text{value}(\Pi') + \hat{v},$$

  that is, that the value of the full solution is *separable* into a term that depends only on the value of the subproblem's solution, plus a term that depends only on the value of the greedy choice.

3 And Now, a Classic Failure

Lest you get all excited and think that greed always works...

- *0-1 knapsack* problem does not permit items to be subdivided.
- **Example**: gold bar, painting, SD card full of purchased songs
- Each item still has weight w_i and value v_i.
- Goal is to maximize value of knapsack without going over weight M.

Fractional algo makes no sense in this context. What to do?

- Well, we can still assign value density d_i to each item.
- Intuitively, items of high value density are more attractive (diamond vs an equal-sized chunk of coal).
- *Suggestion*: sort items by decreasing value density as before, then choose items of highest density until next item would exceed total weight of M.
- Does this work? (**wait**)
- Counterexample with 3 items and $M = 5$:

- values $= 20, 30, 40$; weights $= 2, 3, 3.5$
- densities are $10, 10,\ 11.4$
- Greedy algo picks item of weight 3.5 first, then stops with value 40.
- Optimal solution would take other two items for total value 50.

What broke?

- There is *no* optimal solution that contains the greedy choice!
- Hence, greedy choice property fails for this problem.
- (In fact, it is NP-hard, but we don't know that yet!)

4 A New Problem: Weighted Telescope Scheduling

- Let's return to our telescope scheduling problem.
- This time, we'll assume that not all requests are treated equally.
- In particular, each project p_i now comes with a weight w_i indicating its importance.
- Our goal is now to find a feasible schedule Π for which the *total weight* $\sum_{p_i \in P} w_i$ 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 a structurally different approach.

- We can't identify 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 consistent with some optimal solution.
- If we come up with the best solution possible given *each* choice, the best among all these solutions must be optimal.
- *Example*: sort the projects P in *some* order $p_1 \dots p_n$.
- 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 $p_1 \dots p_{n-1}$.
- If p_n *is* part of 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'.
- A globally optimal solution either uses p_n or not, so if we compute the best possible solutions for each choice, the best among them is globally optimal.
- 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 $W' + 0$.

The above sounds suspiciously like a recursive algorithm!

```
MaxWeight(P = {p_1 ... p_n}) ▷ returns a pair (solution, value)
  if P = ∅
      return (∅, 0)
  (Π_a, w_a) ← MaxWeight(P − {p_n})
  (Π_b, w_b) ← MaxWeight({p ∈ P | p ∩ p_n = ∅})
  if w_a + 0 > w_b + w_n
      return (Π_a, w_a)
  else
      return (Π_b ∪ {p_n}, w_b + w_n)
```

5 Formally Proving Correctness

- Our proof strategy is very similar to what we did for greedy algorithms.
- For a greedy algorithm, we showed that a single first choice is *always* consistent with optimality and yields subproblems that satisfy the inductive structure and optimal substructure properties.
- Since we now make multiple first choices, we need to show that
  1. At least one of our first choices is consistent with optimality.
  2. *Each* possible first choice leaves us with a smaller instance of the original problem with no external constraints. (That is, we can feasibly combine any solution to that smaller problem with our first choice.)
  3. For *each* possible first choice, if we solve the resulting subproblem optimally to get some Π' and combine Π' with our first choice, there is no better way to solve the problem *given that first choice.*
- The first property is called "complete choice".
- The second and third are simply inductive structure and optimal substructure, repeated once for each possible first choice.

Let's check these three points for our algorithm.

- First, an optimal solution must either use p_n or not, so our two choices clearly form a complete choice 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, consider the weight of the final solution in each of the two cases, assuming that an optimal solution to the subproblem has weight W'.

- If we do not use p_n, the weight is $W' + 0$, which is separable and so satisfies optimal substructure.
- If we do use p_n, the weight is $W' + w_n$, which is separable and so satisfies optimal substructure.

As before, we can combine these three properties to prove correctness of the whole algorithm.

- **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 inductive structure, 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 complete choice, there exists an optimal solution Π^* that makes one of the choices made by the algorithm – use p_n, or don't.
- For each of these cases, we know by optimal substructure that combining the subsolution (Π_a or Π_b) with its respective first choice gives the best possible solution for that first choice.
- Conclude that the best among these combined solutions is globally optimal. QED

6 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 $p_1 \dots p_n$ 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 $p_1 \dots p_{c(n)}$ 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 $p_1 \dots p_{n-1}$.
- 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 $P_i = \{p_1 \dots p_i\}$, 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 $P_i = \{p_1 \dots p_i\}$?
- 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 ... p_n.
  T[0] ← (∅, 0)
  for i = 1..n do
      (Π_a, w_a) ← T[i − 1]
      (Π_b, w_b) ← T[c(i)]
      if (w_a + 0 > w_b + w_i)
          T[i] ← (Π_a, w_a)
      else
          T[i] ← (Π_b ∪ {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 $O(n)$ sub-solutions.
- 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 the optimal weight.
- The code looks like this:

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

  Π ← ∅
  j ← n
  while j > 0 do
      if T[j] = T[j − 1]                    ▷ did not use p_j
          j ← j − 1
      else                                  ▷ did use p_j
          Π ← Π ∪ {p_j}
          j ← c(j)
  return (Π, 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 $O(n \log n)$.

7 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, at least 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, any sequence of choices must lead to one of 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.
- (You might have to order your inputs to make this happen, as for prefixes in our example.)
- **Second**, find a total ordering of the distinct 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 preparing (e.g. sorting) the input.
- **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.