Dynamic programming

Abhiram Ranade

February 18, 2016

Weighted interval scheduling

Input: Integers $s_i, f_i, w_i, i = 1, \ldots, n$ which are are respectively start time, finish time, and weight of ith interval.

Output: Subset of disjoint intervals of largest total weight.

Example: Input: ((1,10,5), $(9, 12, 9), (11, 20, 5)$)
Output: Intervals (1,10), (11,20), total weight = 5+5=10.

Application: Each interval is a reservation request for some resource, which can be used at an instant for only for one activity. w_i = payment offered for the resource. Goal is to select requests so as to collect maximum payment.

Exercise: Greedy works for the above example, but does not work in general. Give an example to show why.

Brute force search?

C : Collection of all possible (sub)sets of activities
We want the max weight set of non-conflicting activities from C.
We will devise a recursive search procedure for this.

C_n = collection of all sets of activities including activity n,
$C_{\bar{n}}$ = collection of all sets of activities not including activity n.

Searching in C is equivalent to Searching in C_n, $C_{\bar{n}}$ and taking best.

We can recurse on $C_{\bar{n}}$.

How do we search C_n?

Best set containing activity n

Lemma: Let C' = collection of activities not containing activity n and activities conflicting with n. Let t = max weight non-conflicting subset of activities in C'. Then $s_n = \{n\} \cup t$ is the max weight subset of non-conflicting activities containing activity n.

Proof: Let $s'_n = \{n\} \cup t'$ have max weight among sets containing activity n.

t' does not contain activities conflicting with activity n.
Thus $t' \in C'$. Thus $w(t') \leq w(t)$.

$w(s'_n) = w_n + w(t')$
$\leq w_n + \mathrm{w(t)}$
$= w(s_n)$

Thus s_n is also optimal. □

t can be found recursively

Implementation

WIS(i){

Will return the weight of the best subset among activities 1..i

Assume activities are sorted by non-decreasing finishing time.

Arrays S,F,W are global

Initial call: WIS(n)

if i=0 return 0

j = largest integer s.t. F[j] $\leq$ S[i], 0 if no such integer.

BestWithi = W[i] + WIS(j)

BestWithouti = WIS(i-1)

Return max(BestWithi, BestWithouti)

}

Exercise: Try it on the following:

```
S = 1    2    3    4    5    6
F = 2.1  3.1  4.1  5.1  6.1  7.1
```

Draw out the recursion tree

Something interesting about the brute force search

Same calls made again and again!

Why not make each call once, store the result, use when needed?

Memoization

Let us store all solutions that we compute in a global array.

We will use a table $T[0..n]$ where in $T[i]$ we will store the maximum weight solution possible through activities $1..i$.
$T[0..n]$ will be initialized to `"empty"`

Before we make a call, we first check in T to see if the corresponding entry has been calculated, i.e. is $\neq$ `"empty"`

New function `OptW(i)` : returns the weight of the best solution for activities $1..i$.

Main program

```
Read in F, S, W.

Sort all arrays so that F is in non decreasing order.

For i=0..n T[i] = empty

OptW(n)

Return T[n]
```

Memoized solution

```
OptW(i){
  if i = 0 return 0

  if T[i-1] = empty then T[i-1] = OptW(i-1)

  k = largest s.t. F[k] <= S[i], 0 if no such.
  if T[k] = empty then T[k] = OptW(k)

  T[i] = max(T[i-1], W[i]+T[k])

  return T[i]
}
```

`k` can be computed by binary search for `S[i]` in $F[1..i-1]$
Time for non-recursive portion $= O(\log n)$
Number of recursive calls: n. Key point!

Total time : $O(n \log n)$. Initial sorting also takes $O(n \log n)$

A more direct approach

We want $T[i] =$ weight of best subset for activities $1..i$.

Instead of writing a recursive program that fills in T, why not fill the values directly?

Idea similar to computing Fibonacci numbers:
$F[i] = F[i-1] + F[i-2]$ can be used for recursion, or we can also fill in the values smallest to largest, given base cases
$F[0] = F[1] = 1$.

A more direct approach (continued)

Some observations:

- $T[i] = \max(T[i-1], w[i] + T[k[i]])$ where $k[i]$ = largest k s.t. $S[i] \geq F[k]$, 0 if no such k.
- We can use $T[0] = 0$ as a base case.

Now we can fill the values of $T[i]$ in increasing order of i.

Example: (Already sorted by F)
S=[1, 2, 5, 9], F=[4, 8, 10, 20], W=[3, 10, 6, 4]

k[1] = 0, k[2] = 0, k[3] = 1, k[4] = 2
T[0]=0, T[1] = max(T[0],w[1]+T[0]) = 3
T[2] = max(T[1], w[2]+T[0]) = 10
T[3] = max(T[2], w[3]+T[1]) = 10
T[4] = max(T[3], w[4]+T[2]) = 14

In general: n elements of T to be filled, each takes $O(1)$ time.
Total time = $O(n)$ excluding time to calculate $k[]$ and initial sort.

Finding the maximizing subset

Assume we are given the table T[1..n].
Can we tell if activity n should be selected, by looking at the table?

Key observation: Activity n should be selected if $T[n] = w[n] + T[k[n]]$.

If on the other hand, $T[n] = T[n-1]$ we know that we can get the best solution without activity n.

If activity i got selected, then we should not select activities $k[i]+1..i-1$, and we should next check if activity $k[i]$ should also be selected.

If activity i was not selected, then we should next check if activity $i-1$ should be selected.

And so on...

Time $= O(n)$

Remarks

- Many problems have the feature that recursive calls in brute force algorithms get repeated.
- In such cases, memoization is useful.
- After this, you can write down a recurrence involving the entries of the memoization table.
- The table can then be filled in a certain order such that all the values needed to fill a certain entry i are filled before entry i.
- The time taken can then be estimated as: number of entries to fill $\times$ time for filling a single entry given the required values from the table.
- Every brute force algorithm will not have repeated recursive calls.
  - What if the brute force algorithm for WIS considered activities in some random order?
  - What will the recursion look like?
- Typically, we will need to pay attention to the order of making decisions also.