

Optimization Problems Recap

Note Title

11/26/2007

ACTIVITY SCHEDULING (chapter 16)

Scheduling (potentially overlapping)

requests of different duration,

such that the following constraints

are satisfied:

- Ⓐ Requests must not overlap.
- Ⓑ Number of requests must be maximal.

Example:

1-2:00
1:30-2:30
2:00-3:00

Assumption:

We have all
information
available
to us.

fictitious
start request

fictitious
end request

$(-\infty, 1)$ $(1, 2)$ $(1.5, 2.5)$ $(2, 3)$ $(3, \infty)$

① BRUTE FORCE APPROACH

- Sort list, eg., by start time. $O(n)$ w/ Counting Sort or by finish time.
- Exhaustively enumerate all possible answers (valid + invalid) and get rid of invalid ones.

$$O(2^n \cdot n)$$

of lists \nearrow \nwarrow examine each end time to make sure no overlap with next start time.

$(-\infty, 1)$ $(1, 2)$ $(1.5, 2.5)$ $(2, 3)$ $(3, \infty)$ \times
 $(-\infty, 1)$ $(1, 2)$ $(2, 3)$ $(3, \infty)$ \checkmark
⋮

- Find list with max # of requests

$$O(2^n)$$

TOTAL : $O(2^n \cdot n)$

Can we do better? We want polynomial.

Can we apply D+Q?

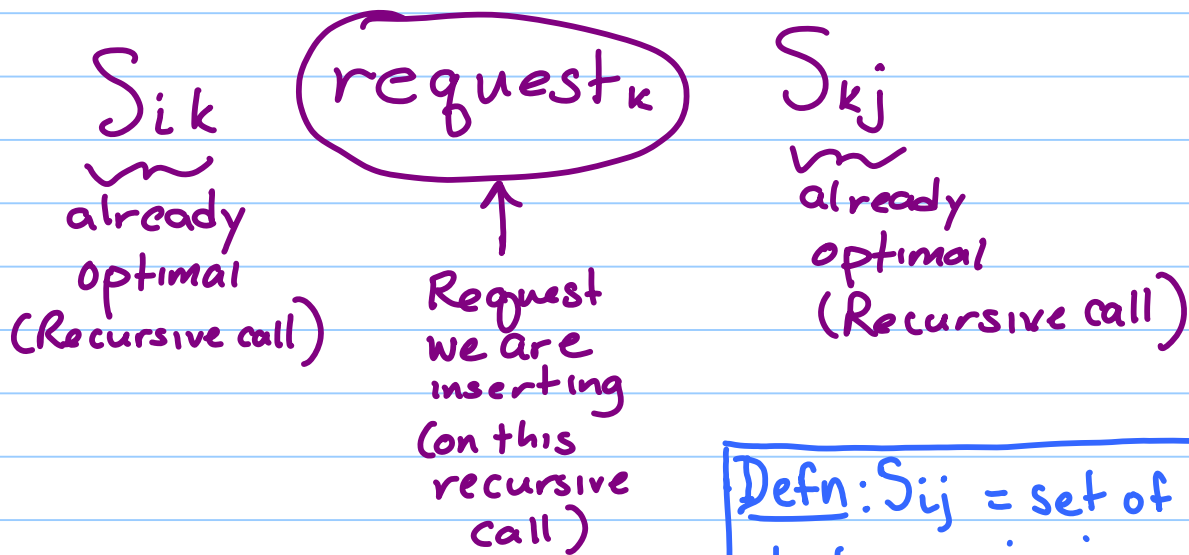


② DYNAMIC PROGRAMMING APPROACH

- Sort list by end time. (Break ties by sorting on start time)
- Add fictitious requests:

request₀, request_{n+1}

- Find largest non-conflicting subset of request₀, n+1 using divide and conquer.



Defn: S_{ij} = set of requests between i + j .
 $= \{r_k \in S \mid f_i \leq s_k < f_k \leq s_j\}$

Algorithm:

OPTIMAL-COUNT $[i, j]$

= MAX (OPTIMAL-COUNT $[i, k]$

+ 1

+ OPTIMAL-COUNT $[k, j]$)



Compute sum for all possible values of k and find max. Top-level recursive call:

OPTIMAL-COUNT $[\emptyset, n+1]$

Dynamic programming:

- Store optimal-count values from sub-problems and use these to solve bigger subproblems.

Will work because, for this particular problem, the overall solution is optimal if each sub-solution is optimal.

- $c[i, j]$ = max # requests from S_{ij} that are compatible with each other.
optimal count

- If $S_{ij} \neq \emptyset$, then request r_k exists s.t.:
$$c[i, j] = c[i, k] + 1 + c[k, j]$$

- Try all r_k 's:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j} (c[i, k] + 1 + c[k, j]) & \text{otherwise} \end{cases}$$



Initialize the matrix c to all zeros.
 Assume we have an array r of request records.

```

for (d=1 to n+1
  for i=0 to n-d+1
    j=i+d
    if (r[i].f<=r[j].s)
      for k=i+1 to j-1
        if (
          ((r[i].f<=r[k].s)
            &&
            (r[k].f<=r[j].s)
          )
          &&
          (c[i,k]+1+c[k,j]>c[i,j])
        )
        then c[i,j]= c[i,k]+1+c[k,j];
  
```

distance between requests

$d=1 : 0,1 \quad 1,2 \quad \dots \quad n,n+1$
 $d=2 : 0,2 \quad 1,3 \quad \dots \quad n-1,n+1$
 $d=3 : 0,3 \quad 1,4 \quad \dots \quad n-2,n+1$

Go through all r_k 's between r_i and r_j

Move distance d away from r_i to get to r_j

Is r_k compatible?

Does r_k get a better result?

Let's look at our example again:

$(-\infty, 1)$ $(1, 2)$ $(1.5, 2.5)$ $(2, 3)$ $(3, \infty)$
 r_0 r_1 r_2 r_3 r_4

$i \backslash j$	0	1	2	3	4
0	0	0	0	0	2
1		0	0	0	1
2			0	0	0
3				0	0
4					0

What does this chart tell you?

- The max number of requests that can be fulfilled.

Does it tell you how to fill them?

- No! Would need additional bookkeeping.
- Need to save information along the way.

- In the homework, you will need to determine how to return a schedule from the resulting chart.

Important: There could be 2 different ways to grant requests. You should focus on extracting one valid schedule.

(Reverse engineer the solution at the end.)

```

Initialize the matrix c to all zeros.
Assume we have an array r of request records.
for d=1 to n+1
  for i=0 to n-d+1
    j=i+d
    if (r[i].f<=r[j].s)
      for k=i+1 to j-1
        if (
          ((r[i].f<=r[k].s)
           &&
          (r[k].f<=r[j].s)
         )
          &&
          (c[i,k]+1+c[k,j]>c[i,j])
        )
          then c[i,j]= c[i,k]+1+c[k,j];

```

What is the runtime of this algorithm?

for d = 1 to n+1 $O(n)$

for i = 0 to n-d+1 $O(n)$

for k = i+1 to j-1

\uparrow
 i+d
 1 away
 2 away
 ⋮

} $O(n)$ feel

$O(n^3)$ worst case

$< O(n \cdot 2^n)$

Can we do better?

The DP solution is overkill

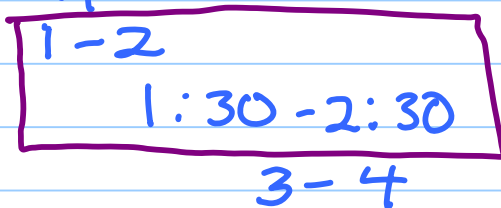
Can we do better than $O(n^3)$?

- Make decisions as data streams in.
- Never look back.

Greedy approach!

- Can't just make Y/N decision as data comes in. That's too greedy. (Adversary could mess things up.)
- So, do some work first: Sort!
(Sort by finish time.)
- $O(n)$ \Rightarrow linear sorting; our data falls in a particular range!

10-11



Which do we throw out?

Pick first! 2PM Gives us more time to schedule other things today! Don't choose 1:30-2:30 because we lose 30 min with no gain.

THIS IS WHY GREEDY WORKS!!

③ GREEDY APPROACH

// Look for "locally optimal" choice and take it.

- Sort list of requests by finish time. $O(n)$
- Take first request in sorted list and put it in the result list. $O(1)$
- Remove everyone who conflicts with that request $O(n)$
- Repeat on remaining requests until sorted list is empty. $O(n^2)$ seems like but it's not!
- Return the result list.

- When you eliminate a choice, you don't have to examine it again!!
- Amortized linear time.

Can this possibly be optimal?

The Greedy Solution is Optimal!

- By taking the first request, we only eliminate:

- Other requests that end at the same time as this one. (Sorting by finish time is key!)



This is fine because we could only have chosen one of all these overlapping requests anyway.

- Other requests that overlapped at some time period.



Again, this is fine for the same reason.

- In the homework, you will need to determine the runtime of this solution and compare it to the two previous solutions' runtimes.