EECS 336: Lecture 4: Introduction to Algorithms

Dynamic Programming (cont)

Reading: 6.4-6.8

Last Time:

- Dynamic Programming (a derivation)
- Weighted interval scheduling

Today:

- Dynamic Programming (a framework)
- Integer Knapsack
- Interval Pricing

From Last Time

Example: Weighted Interval Scheduling

input:

- n jobs $J = \{1, ..., n\}$
- s_i = start time of job i
- f_i = finish time of job i
- v_i = value of job i

compatibility constraint: Only one job can run at once.

output: Schedule $S \subseteq J$ if compatible jobs with maximum total value.

Recursive Memoized Algorithm

Algorithm: Weighted Interval Scheduling:

1. sort jobs by increasing start time.
2. initialize array next$[i]$.
3. initialize OPT$[i] = \varnothing$ for all i.
4. initialize OPT$[n + 1] = 0$.
5. compute OPT(1).

Subroutine: OPT(i)

1. if OPT$[i] \neq \varnothing$, return OPT$[i]$.
2. OPT$[i] \leftarrow \max(v_i + \text{OPT}[\text{next}[i]],\ \text{OPT}[i + 1])$.
3. return OPT$[i]$.

Unfinished Business

Iterative DPs

"fill in memoization table from bottom to top"

Algorithm: iterative weighted interval scheduling

1. OPT$[n + 1] = 0$
2. for $i = n$ down to 1:

   OPT$[i] = \max(v_i + \text{OPT}[\text{next}[i]], \text{OPT}[i + 1])$.

Finding Optimal Schedule

"traverse memoization table to find schedule"

Algorithm: schedule

1. $i = 1$
2. while $i < n$:

   if OPT$[i + 1] < v_i + \text{OPT}[\text{next}[i]]$:

   (a) schedule i.
   (b) $i \leftarrow \text{next}(i)$.

   else: $i \leftarrow i + 1$.

Key Ideas of Dynamic Programming

Subproblems must be:

1. succinct (only a polynomial number of them)
2. efficiently combinable.

3. depend on “smaller” subproblems (avoid infinite loops), e.g.,
    - process elements “once and for all”
    - “measure of progress/size.”

Seven Part Approach

I. identify subproblem in English

   OPT(i) = “optimal schedule of $\{i, ..., n\}$ (sorted by starting time)”

II. specify subproblem recurrence (argue correctness)

   OPT(i) = max(OPT($i+1$), v_i + OPT(next[i]))

III. solve the original problem from subproblems

   Optimal Interval Schedule = OPT(1)

IV. identify base case

   OPT($n+1$) = 0

V. write iterative DP.

VI. runtime analysis.

   $O(n)$ + initialization = $O(n \log n)$

VII. implement in your favorite language (Python!)

Dynamic Programming: Finding Subproblems

"find a first decision you can make which breaks problem into pieces that

- do not interact (across subproblems)
- can be described succinctly."

Example: Integer Knapsack

intput:

- n objects $S = \{1, ..., n\}$
- s_i = size of object i (integer)
- v_i = value of object i
- C = capacity of knapsack (integer)

output:

- subset $K \subseteq S$ of objects that
    (a) fit in knapsack together

    (i.e., $\sum_{i \in K} s_i \leq C$)

    (b) maximize total value

    (i.e., $\sum_{i \in K} v_i$)

Question: What is “first decision we can make” to separate into subproblems?

Answer: Is item 1 in the knapsack or not?

- if 1 in knapsack:

  value of knapsack is v_i+ optimal knapsack value on $S \backslash \{1\}$ with capacity $C - s_1$.

- if 1 not in knapsack:

  value of knapsack is optimal knapsack on $S \backslash \{1\}$ with capacity C.

Succinct description:

- remaining objects $\{j, ..., n\}$ represented by “j”
- remaining capacity represented by $D \in \{0, ..., C\}$.

Step I: identify subproblem in English

$\text{OPT}(j, D)$ = "value of optimal size D knapsack on $\{j, ..., n\}$"

Step II: write recurrence

$$\text{OPT}(j, D) = \max(\underbrace{v_j + \text{OPT}(j+1, D - s_j)}_{\text{if } s_j \leq D}, \text{OPT}(j+1, D))$$

Justification: either i is in or not (exhaustive.)

Step III: solve original problem

Value of Optimal Knapsack = $\text{OPT}(1, C)$

Step IV: base case

$\text{OPT}(n+1, D) = 0$ (for all D)

Step V: Iterative DP

Algorithm: knapsack

1. $\forall D, \text{OPT}[n+1, D] = 0$.
2. for $i = n$ down to 1,

   for $D = C$ down to 0,

   - if i firts (i.e., $s_i \leq D$)

     $\text{OPT}[i, D] = \max[\text{OPT}[j+1, D], v_j + \text{OPT}(j+1, D - s_j)]$

   - else

     $\text{OPT}[j, D] = \text{OPT}[j+1, D]$

3. return $\text{OPT}[1, C]$

Step VI: Runtime

$T(m, C) = O(\#$ of subprobs $\times$ cost per subprob$) = O(nC)$.

Note: not polynomial time.

Step VII: implementation

(see "guide")

Alternative Approach

"isolate previously made decisions"

Suppose:

- already processed jobs $\{1, ..., i\}$, and
- used capacity D.

Note: previous decisions succinctly summarized by i and D

Part I: subproblem in english

$\text{OPT}(i, D)$ = "value from remaining knapsack if

- alread processed jobs $\{1, ..., i\}$
- used capacity D."

Part II: recurrence

$\text{OPT}(i, D) = \max(v_i + \text{OPT}(i+1, D + s_i), \text{OPT}(i+1, D))$

(assuming $D + s_i \leq C$)

...