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

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.

Iterative DP

Algorithm: iterative weighted interval scheduling

1. initialize array next$[i]$

   sort jobs by increasing start time

   $\text{OPT}[n+1] = 0$

2. for $i = n$ down to 1:

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

3. return OPT[1]

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)

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

III. solve the original problem from subproblems

   Optimal Interval Schedule = OPT(1)

IV. identify base case

   $\text{OPT}(n+1) = 0$

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

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

Step II: write recurrence

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 = OPT$(1, C)$

Step IV: base case

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., $s_i \leq D$)

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

   - else

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

3. return OPT$[1, C]$

Step VI: Runtime

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

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

$$\text{OPT}(i, D) = \max(\text{OPT}(i + 1, D), \underbrace{v_i + \text{OPT}(i + 1, D + s_i)}_{\text{if } D + s_i \leq C})$$
