Michael A. Forbes
mforbes@illinois.edu
2024-01-25.1

cs473 Algorithms: Lecture 4 (2024-01-25)

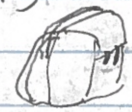
logistics – pset 0 due F17 [gradescope]
– pset 1 out F17

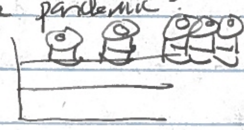
last lecture: dynamic programming: weighted interval scheduling
– brute force $O(n \cdot 2^n)$
– recursive $O(2^n)$
– memoized $O(n)$
  ↳ w/ soln
– iterative $O(n)$ [strongly] prefer iterative algo [easier to analyze] (differs prev. lec)

today – dynamic programming

reading – KT 6.2, 6.4

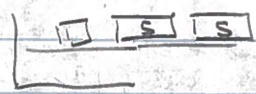
Q: buying toilet paper during the pandemic? [scarce due to hoarding]


[knapsack]


[toilet paper on shelf]
[bundled sizes]
[maximize TP in knapsack, wrt capacity]
[not harry potter]

Q: fill [greedily]?
↳ natural algorithmic paradigm where:
[local] optimal choices ⇒ [global] optimal choice.

eg – capacity 10 knapsack


← sorted increasing sizes


⟶ [one pass] [speedrun] ⇒ pick 1 6 || < 10 || = 5+5 [capacity] OPT [⇒ fails]

eg: capacity 10 [6] [5] [5] ← sorted [decreasing] sizes
[one pass] ⇒ pick 6 || < 10 || = 5+5 [capacity] OPT [fail]

Q: optimize [quality] of toilet paper? [1-ply vs 2-ply]

def: knapsack problem is given
– weights $w_1, \dots, w_n \in \mathbb{N}$ [non-neg], weight limit $W \in \mathbb{N}$
– values $v_1, \dots, v_n \in \mathbb{N}$

compute
$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \le W}} \sum_{i \in S} v_i$$
[max value] [weight constraint]

the subset sum problem is when $v_i = w_i$ [all TP same value] vs 2022 neg w_i, v_i

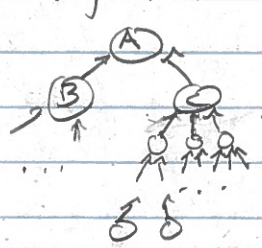
Q: [any] algo?

prop: knapsack solvable in $O(n \cdot 2^n)$ time [≥ fast recursive] [brute force, check – feasibility – value] greedy fails

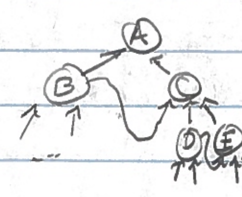
Q: [efficient] algo? dynamic programming?

idea (dynamic programming): compress recursion [tree] [naive] into recursion [directed acyclic graph] [memoized] [recursion + memoization]
↳ DAG
– identify [small] set of subproblems [new problems, related to orig]
– identify relations between subproblems

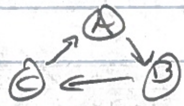
tree:


[# calls]
[base cases]
exp-many nodes

DAG:


[reuse solutions]
poly-many nodes

Michael A. Forbes
miforbes@illinois.edu
2024-01-25.2

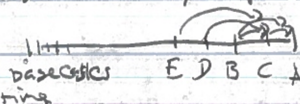
[no] cycles  ⟦dependency hell⟧ ⟹ directed [acyclic] graph

idea: solve recursion DAG by:

- memoization – – top down
  - implicitly create DAG ⟦explore DAG [without] knowing its full structure⟧ ⟦hard to analyze⟧

  eg 

- iterative: – bottom up
  - explicitly create DAG w/ topological sort ⟦all edges [forward]⟧ ⟦when reach subproblem all prereqs satisfied⟧

  eg: 

homework vs 'coursework' submissions: – design iterative DP ⟦easier to analyze⟧ – make subproblems [clear] ⟦easier for graders⟧

def: knapsack weights $w_1, \ldots, w_n, W$

values $v_1, \ldots, v_n$

compute $\max \sum_{i \in S} v_i$

$\sum_{i \in S} w_i \le W$

Q: dynamic programming? subproblems? relations between subproblems?

idea: mimic weighted interval scheduling

OPT(k) vs OPT_k

def: $OPT(k) = \max_{S \subseteq [k]} \sum_{i \in S} w_i$ ⟦first k items⟧, $\sum_{i \in S} w_i \le W$

prop: $\{S : S \subseteq [k-1], \sum_{i \in S} w_i \le W\} \subseteq \{S \subseteq [k] : \sum_{i \in S} w_i \le W\}$ ⟦feasible soln for k-1⟧ ⟦feasible for k⟧ ⟦$w_i \ge 0$⟧

cor: $OPT(k) \ge OPT(k-1)$ ⟦only more options⟧

cor: [if] exists optimal soln $S \subseteq [k]$ for $OPT(k)$ w/ $k \notin S$ ⟹ $OPT(k) = OPT(k-1)$ ⟦≈ weighted interval scheduling⟧

Q: if $k \in S$?

item k has weight w_k ⟹ leaves weight $W - w_k$ for rest of items
[not] one of our subproblems

idea: [expand] subproblems

def: $OPT(k, t) = \max_{S \subseteq [k]} \sum_{i \in S} v_i$ ⟦instance⟧ ⟦new⟧, $\sum_{i \in S} w_i \le t$

Michael A. Forbes
miforbes@illinois.edu
2024-01-25.3

finish 2022.

prop: $\{S \subseteq [k] : \sum_{i \in S} w_i \le t\} = \{S \subseteq [k-1] : \sum_{i \in S} w_i \le t\}$

feasible soln for OPT(k, t) — feasible soln for OPT(k-1, t) [-1 item, same weight]

$\cup \{S = \{k\} \cup T : T \subseteq [k-1]\ \sum_{i \in S} w_i \le t\}$

$= \{k\} \cup T$, for

$T \in \{T \subseteq [k-1] : \sum_{i \in T} w_i \le t - w_k\}$ — feasible soln for OPT(k-1, t-w_k) [-1 item, less weight]

[main] [no feasible soln to]

cor: if $w_k > t$, OPT(k, t) = OPT(k-1, t)

else OPT(k, t) = max{OPT(k-1, t), OPT(k-1, t-w_k) + v_k} [value of item k]

DP feasible set meets thm.

prop: knapsack solvable in $O(nW)$ time

pf: algo: [iterative]

- array M[0, ..., n][0, ..., W]
- for $0 \le t \le W$
  - M[0][t] = 0 [no items, no value] [base case]
- for $1 \le k \le n$
  - for $0 \le t \le W$
    - if $w_k > t$ M[k][t] = M[k-1][t] [is defined!]
    - else M[k][t] = max{M[k-1][t], M[k-1][t-w_k] + v_k}
- return M[n][W]

pseudo code.

correctness: clear [structural result]

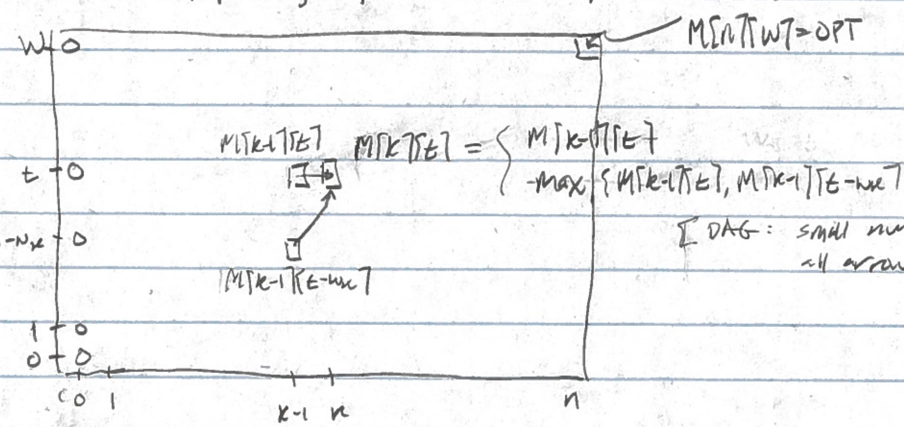
complexity: clear [iterative]

Q: find optimal solution?

prop: given filled M[·][·] [above], optimal soln computable in $O(n)$ time

give a real proof if there is time

sketch:



[DAG: small number of subproblems, all arrows point "forward"]

vs. pf

cor: if $w_k > t$: OPT(k, t) soln = OPT(k-1, t) soln

else: — " — if M[k-1][t] ≥ M[k-1][t-w_k] + v_k

OPT(k, t) soln = {k} ∪ OPT(k-1, t-w_k) soln else

algo: clear [implement this] [follow the arrows]

correctness: clear

complexity: n applications of

expand this

Michael A. Forbes
miforbes@illinois.edu
2024-01-25. 4

Q: is this actually efficient?

A: yes if $W \le n^{O(1)}$ [@ lecture 2]

true by conventions in this course

A: no if $W = 2^n$

↳ arithmetic still efficient here [reasonable to ask for algo]

def: algo on n integers $a_1, \ldots, a_n$ is pseudopolynomial if it runs in poly$(\sum |a_i|)$ time [unary vs binary]

rmk: poly time would be poly$(\sum O(\lg |a_i|))$

bit complexity of input

cor: knapsack has a pseudo polytime algo

thm: knapsack is NP-complete when W allowed to be exponential in n

⟹ strong evidence no polytime algo

today: dynamic programming

- abstract - trees [naive recursion] vs DAG [storing subproblem soln]
- memoization vs iterative
- knapsack - param subproblem w/ new variable

reading: KT 6.2, 6.4

next lecture: dynamic programming

logistics:
- pset 0 due F1/7
- pset 1 out F1/7