Knapsack and Dynamic Programming

6.1000 LECTURE 21

FALL 2025

Announcements

- Pset 5 checkoff due today
- Pset 6 due 12/1, Monday after Thanksgiving

- No office hours on 11/26, Wednesday before Thanksgiving
- No office hours on 12/10, last day of classes
- Instructor office hours on 12/11, Thursday before finals

Recall: Knapsack problems

- You have limited strength, so there is a maximum weight knapsack that you can carry
- You want to take more stuff than you can carry
- How do you choose what to take vs. leave behind?
  - Want to optimize "value" of things to take
- Two variants
  - Continuous or fractional knapsack problem — Straightforward
  - 0/1 knapsack problem — Much more interesting

Quanta are infinitesimal relative to available space

versus

Quanta are large relative to available space

Greedy solutions

- Repeatedly pick the best item among what's available
  - Once you pick, you have to commit, no backtracking
- Continuous knapsack
  - Pick as much of the most value-dense item as possible
- 0/1 knapsack
  - Must take an entire object or leave it behind
  - Possible greedy metrics: by value, by weight, by density
  - None are guaranteed to yield optimal solution

Enumeration of item combinations

- Optimal solution
  - Enumerate all possible item selections
  - For each, check whether it satisfies the capacity constraint
  - Among those that do, keep the one with the most value
- How to enumerate
  - Approach 1: recursively generate
    - generate all combinations for **`items[1:]`**
    - collect each combo both with and without **`item[0]`**
  - Approach 2: read off binary digits
    - let $\boldsymbol{n}$ be **`len(items)`**
    - generate binary representations of $\mathbf{0}$ through $\mathbf{2^n - 1}$
    - each is an $\boldsymbol{n}$-character string of $\mathbf{0}$'s and $\mathbf{1}$'s
    - each $\mathbf{1}$ selects the item at that index

Runtime performance of enumeration

- $\boldsymbol{2^n}$ combinations, grows exponentially in number of items
  - $2^{10} \sim 10^3$, $2^{20} \sim 10^6$, $2^{30} \sim 10^9$
  - Processor speeds $\sim 10^9$ Hz (cycles per second)
  - $\sim 10^3$ cycles to process each combination
- Solution times become user-unfriendly around $\boldsymbol{n \sim 25}$
  - Each increment in $\boldsymbol{n}$ will **double** the size of the solution space to consider
  - Not scalable
  - Are we doomed to rely on greedy solutions?

Pruning branches in decision tree

- While generating the combinations, keep track of remaining capacity as we go descend branches in the decision tree
- If taking the next object would cause our selection to go over capacity, do not consider that branch
- Testing result
  - Can solve up to $n \sim 35$ in less than a minute
  - Better, but still not practical
- In-class exercise
  - Decision-tree implementation makes list copies when slicing
  - Rewrite so that it recurses on ***index of next item*** to consider, rather than on list of remaining items

Avoid re-solving overlapping subproblems

- Consider items $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$, … with weights $\mathbf{2}, \mathbf{3}, \mathbf{2}, \mathbf{1}$, … and capacity $\mathbf{20}$
- After going down the branch $+\boldsymbol{A}, +\boldsymbol{B}, -\boldsymbol{C}$, need to solve a subproblem for items $\boldsymbol{D}, \boldsymbol{E}$, … with capacity $\mathbf{15}$
- After going down the branch $-\boldsymbol{A}, +\boldsymbol{B}, +\boldsymbol{C}$, need to solve the **exact same subproblem!**
- ***Idea (dynamic programming)***
  - Remember the result the first time
  - Recognize if we re-encounter, use saved result

Dynamic programming

- Broadly applicable algorithm strategy
  - Principles were invented by Richard Bellman in the 1950s
  - Name was chosen for "marketing"
    - "programming" in the sense of "creating a schedule/solution"
    - "dynamic" because it sounds cool
    - https://en.wikipedia.org/wiki/Dynamic_programming#History_of_the_name
- **Correctness requirement: optimal substructure**
  - optimal solutions involve optimal solutions to subproblems
- **Efficiency requirement: overlapping subproblems**
  - different subproblems rely on the same subproblems
  - how efficient depends on amount of subproblem overlap