Discussion 11: Dynamic Programming I

(CS101 Fall 2024)

CS101 Course Team

December 16, 2024

Recall: Bellman-Ford is dp

1. Let dp[x][k] be the shortest path from s to t with **exact** k edges.

Recall: Bellman-Ford is dp

1. Let dp[x][k] be the shortest path from s to t with **exact** k edges.
2. Let dp[x][k] be the shortest path from s to t with **no more than** k edges.(Homework 10 problem 6)

Recall: Bellman-Ford is dp

1. Let dp[x][k] be the shortest path from s to t with **exact** k edges.
2. Let dp[x][k] be the shortest path from s to t with **no more than** k edges.(Homework 10 problem 6)

What will happen if we use a scroll array?

Algorithmic paradigms

- **Greedy**: Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide and conquer**: Break up a problem into a few sub-problems, solve each sub-problem independently and recursively, and combine solutions to sub-problems to form a solution to the original problem.
- **Dynamic programming**: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
    - Very powerful and widely used technique in CS, info, and control theory.
    - Efficiently solves problems that otherwise seem intractable.
    - Name comes from the dynamic "schedule" of subproblems the algorithm produces.

Maximum Subarray

Given a sequence $\{a_0, a_1, ..., a_{n-1}\}$, calculate $\max\{\sum_{k=l}^{r} a_k | 0 \le l \le r < n\}$

Maximum Subarray

Given a sequence $\{a_0, a_1, ..., a_{n-1}\}$, calculate $\max\{\sum_{k=l}^{r} a_k | 0 \le l \le r < n\}$

- Let f_i be the maximum subarray ended with a_i. Final state will be $\max_{i=0}^{n-1} f_i$.
- If a_i can be joined with the previous subarray: $f_{i-1} + a_i$
- Otherwise, a_i will "be the header of subarray: a_i
- $$f_i = \max\{f_{i-1} + a_i, a_i\}$$

Maximum Subarray

Given a sequence $\{a_0, a_1, ..., a_{n-1}\}$, calculate $\max\{\sum_{k=l}^{r} a_k | 0 \leq l \leq r < n\}$

- Let f_i be the maximum subarray ended with a_i. Final state will be $\max_{i=0}^{n-1} f_i$.
- If a_i can be joined with the previous subarray: $f_{i-1} + a_i$
- Otherwise, a_i will "be the header of subarray: a_i
- $$f_i = \max\{f_{i-1} + a_i, a_i\}$$
- Very simple! It's like another formulate of recursion. $\Theta(n)$.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence in $\langle a_0, a_1, \ldots, a_i \rangle$. The final answer is f_{n-1}.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \dots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \dots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence in $\langle a_0, a_1, \dots, a_i \rangle$. The final answer is f_{n-1}.
- Compared to the previous subproblem, the additional information is a_i.
- Consider the impact of a_i: it may or may not be in the longest increasing subsequence.
  - If not, then the answer is f_{i-1}.
  - If it is, then whom does it follow? How do you know if it can follow?

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence ending with a_i. Then the final answer is $\max_{i=0}^{n-1} f_i$.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence ending with a_i. Then the final answer is $\max_{i=0}^{n-1} f_i$.
- a_i may follow some previous a_j as long as $a_j \leq a_i$.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence ending with a_i. Then the final answer is $\max_{i=0}^{n-1} f_i$.
- a_i may follow some previous a_j as long as $a_j \leq a_i$.
- $f_i = \max\{f_j + 1 \mid 0 \leq j < i, a_j \leq a_i\}$.

Longest Increasing Subsequence

Given a sequence $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, find the longest subsequence $a_{s_0}, a_{s_1}, \ldots, a_{s_{k-1}} (s_0 < s_1 < \cdots < s_{k-1})$ such that $a_{s_0} \leq a_{s_1} \leq \cdots \leq a_{s_{k-1}}$.

- Let f_i represent the length of the longest increasing subsequence ending with a_i. Then the final answer is $\max_{i=0}^{n-1} f_i$.
- a_i may follow some previous a_j as long as $a_j \leq a_i$.
- $f_i = \max\{f_j + 1 \mid 0 \leq j < i, a_j \leq a_i\}$.
- What's the time complexity?

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

- Let $f(i, j)$ represent the maximum total value of selecting some of the first i items and placing them into a knapsack with capacity j. Then the answer should be $f(n, W)$.

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

- Let $f(i,j)$ represent the maximum total value of selecting some of the first i items and placing them into a knapsack with capacity j. Then the answer should be $f(n, W)$.
- If we choose the i-th item, this item must satisfy $w_i \leq j$. at this moment we need to get a pack with $j - w_i$ capacity. That is

$$f(i-1, j-w_i) + v_i$$

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

- Let $f(i, j)$ represent the maximum total value of selecting some of the first i items and placing them into a knapsack with capacity j. Then the answer should be $f(n, W)$.
- If we choose the i-th item, this item must satisfy $w_i \leq j$. at this moment we need to get a pack with $j - w_i$ capacity. That is

$$f(i-1, j-w_i) + v_i$$

- Otherwise, we won't choose this item. That is

$$f(i-1, j)$$

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

- $$f(i,j) = \begin{cases} \max\{f(i-1,j-w_i)+v_i, f(i-1,j)\}, & \text{if } w_i \le j, \\ f(i-1,j), & \text{otherwise.} \end{cases}$$
- totally $\Theta(nW)$ state of f.
- Time complexity and space complexity if $\Theta(nW)$.
- Rotate the array: $f(i,\cdot)$ only depends on$f(i-1,\cdot)$. Then we can implement our algorithm in $O(W)$ space complexity.

0-1 Knapsack problem

There are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. The knapsack has a weight capacity of W. How do you choose the item packed into the knapsack to maximize the sum of values?

- $$f(i,j) = \begin{cases} \max\{f(i-1, j-w_i) + v_i, f(i-1,j)\}, & \text{if } w_i \le j, \\ f(i-1,j), & \text{otherwise.} \end{cases}$$
- totally $\Theta(nW)$ state of f.
- Time complexity and space complexity if $\Theta(nW)$.
- Rotate the array: $f(i,\cdot)$ only depends on$f(i-1,\cdot)$. Then we can implement our algorithm in $O(W)$ space complexity.
- W? Knapsack is not a polynomial time problem! We will learn that this is an NPC problem.

Idea: DP is a state transfer graph.

- Consider every (i, j) is a point: There are an edge from $(i-1, j-w_i)$ to (i, j) with edge weight v_i.
- The problem transferred to the longest-path problem.
- And mention that: this graph is a DAG! So do topological sort.
- Many DP problems can be transferred into a state transfer graph: every state can be formulated as a vertex, and the edge represents the transfer between statuses.

Complete Knapsack problem

Similar to the 0-1 knapsack problem: in the 0-1 backpack problem, every item can be selected only once. For the complete knapsack problem: it can be selected multiple times.

- Basic Idea: for item i, enumerate how many items are selected:

$$f(i,j) = \max_k \{f(i-1, j-kw_i) + kv_i\}.$$

Complete Knapsack problem

Similar to the 0-1 knapsack problem: in the 0-1 backpack problem, every item can be selected only once. For the complete knapsack problem: it can be selected multiple times.

- Basic Idea: for item i, enumerate how many items are selected:

$$f(i,j) = \max_k \{f(i-1, j-kw_i) + kv_i\}.$$

- As we increasingly enumerate over j: $f(i,j)$ only needs to transfer from $f(i, j-w_i)$!

$$f(i,j) = \max\{f(i-1,j), f(i,j-w_i) + v_i\}$$

- Why?

Complete Knapsack problem

Similar to the 0-1 knapsack problem: in the 0-1 backpack problem, every item can be selected only once. For the complete knapsack problem: it can be selected multiple times.

- Basic Idea: for item i, enumerate how many items are selected:

$$f(i,j) = \max_k \{f(i-1, j-kw_i) + kv_i\}.$$

- As we increasingly enumerate over j: $f(i,j)$ only needs to transfer from $f(i, j-w_i)$!

$$f(i,j) = \max\{f(i-1,j), f(i,j-w_i) + v_i\}$$

- Why?
- The reason is that, when we make this transition, $f_{i,j-w_i}$ has already been updated by $f_{i,j-2\times w_i}$. Thus, $f_{i,j-w_i}$ fully considers the optimal result after choosing the i-th item.
- In other words, we optimize the complexity of the knapsack problem by reusing the previous knapsack results through the property of local optimal substructure.

DP problems

Requirements

- No aftereffect: Decisions after won't be affected by decisions and state before.
- Optimal substructures: Subproblem optimal $\rightarrow$ Problem optimal.

Three Main Components of DP

- State (together with subproblems): Different (subsets of) problems.
- Transitions: The way to transfer between states.
- Initial states

Frame Title

- Structure of States(Subproblems).
  - Euclidean (1D,2D,...) OPT(i) (e.g. Weighted Interval Scheduling, Maximum Subarray), OPT(i,j) (e.g. Knapsack, Min Cost Refueling, LCS)
  - Tree/DAG (e.g. Critical Path, not required in CS101)
  - Subsets
- Structure of Transitions.
  - Linear Choice (e.g. Weighted Interval Scheduling, Knapsack, LCS)
  - Multiple Choices (e.g. House Coloring, Odd Numbers Coin Changing)
  - Interval Choices (e.g. Segmented Least Squares, Max Power)
- States and subproblems.
  - Exactly subproblem: (e.g. Exact Knapsack)
  - Intervals of subproblems: (e.g. Knapsack, LCS)