Dynamic Programming by Richard Bellman

Jill-Jênn Vie

Sep 6, 2024

First example: Fibonacci numbers

Naive

```python
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

First example: Fibonacci numbers

Naive

```python
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Memoization

```python
from functools import cache

@cache
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

First example: Fibonacci numbers

Naive

```python
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Memoization

```python
from functools import cache

@cache
def fibo(n):
    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Dynamic programming

```python
def fibo2(n):
    f = [0] * (n + 1)
    f[1] = 1
    for i in range(2, n + 1):
        f[i] = f[i - 2] + f[i - 1]
    return f[n]
```

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

States: (i first objects, capacity c)

Let us call $maxValue[i][c]$ the highest value one can obtain with first $i \in [1, n]$ objects and capacity $c \in [0, C]$. First, initialize. Then:

For the ith object:

- Either we take it, and go back to ($i - 1$, $c - c_i$ state) if exists

$$v_i + maxValue[i-1][c-c_i]$$

- Or we don't: go to $(i - 1, c)$ state

$$maxValue[i-1][c]$$

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

States: (i first objects, capacity c)

Let us call $maxValue[i][c]$ the highest value one can obtain with first $i \in [1, n]$ objects and capacity $c \in [0, C]$. First, initialize. Then:

For the ith object:

- Either we take it, and go back to ($i - 1$, $c - c_i$ state) if exists

$$v_i + maxValue[i-1][c-c_i]$$

- Or we don't: go to $(i - 1, c)$ state

$$maxValue[i-1][c]$$

Variants (besides coin change)

- Taking several times the same object instead of once
- Does this work with negative values? Negative capacities?

DP method

1. Try to identify states.
2. Find the recurrence relation.
3. If memoization: use `std::map`. If DP: initialize well.

Application to shortest paths

- Bellman-Ford: $d_k[v]$ = shortest length from source to v using at most k edges

$$d(v) = \min_u d(u) + w_{uv}$$

- Floyd-Warshall: $d_k[u][v]$ = shortest length between u and v using only nodes $< k$

$$d(u, v) = \min_w d(u, w) + d(w, v)$$

Problems last week

- Longest path in a DAG
- Ricochet Robots
- Ingredients

Gold mine problem (Bellman, 1952)

Two gold mines, Anaconda (amount x) and Bonanza (amount y).

- Anaconda: probability p_1 to collect $r_1 x$ $\quad$ $1 - p_1$ to break the machine forever
- Bonanza: probability q_1 to collect $r_2 y$ $\quad$ $1 - q_1$ to break the machine forever

Parameters of the problem: $p_1, q_1, r_1, r_2 \in [0, 1]$, $x, y \in \mathbb{R}^+$.

What is the optimal action, that maximizes the amount of extracted gold?

Gold mine problem (Bellman, 1952)

Two gold mines, Anaconda (amount x) and Bonanza (amount y).

- Anaconda: probability p_1 to collect $r_1 x$ — $1 - p_1$ to break the machine forever
- Bonanza: probability q_1 to collect $r_2 y$ — $1 - q_1$ to break the machine forever

Parameters of the problem: $p_1, q_1, r_1, r_2 \in [0, 1]$, $x, y \in \mathbb{R}^+$.

What is the optimal action, that maximizes the amount of extracted gold?

Let $f(x, y)$ be the maximum expected gold extracted.

Solution to the gold mine problem

$$f(x,y) = \max \left\{ \begin{array}{l} p_1(r_1 x + f((1-r_1)x, y) \\ q_1(r_2 y + f(x, (1-r_2)y) \end{array} \right\}.$$

Solution

Si $\dfrac{p_1 r_1}{1-p_1} x > \dfrac{q_1 r_2}{1-q_1} y$,

Choisir A

Sinon

Choisir B.

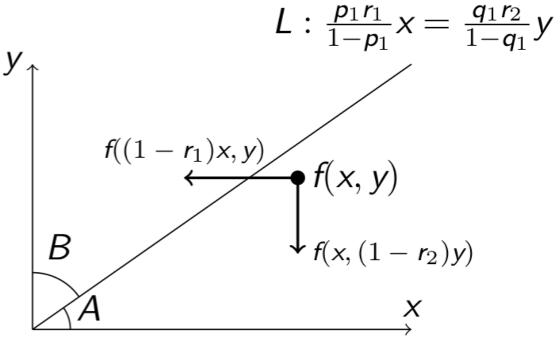


Plate-breaking problem

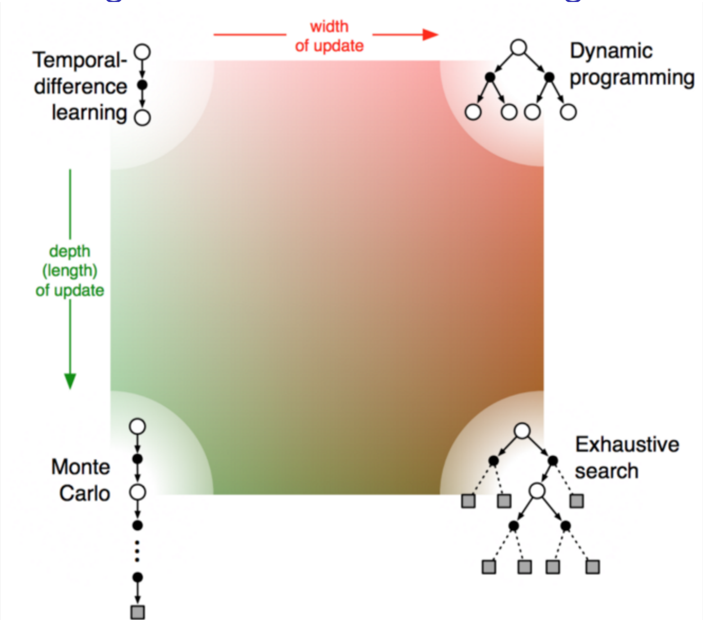
We have k plates and a building of n floors.

To identify the critical floor (i.e. highest floor where the plate does not break), we throw plates. If the plate breaks, we cannot reuse it; otherwise we can.

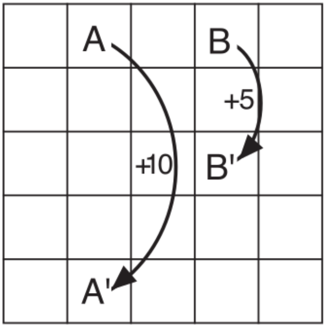
We want to find the critical floor in a minimum number of moves in the worst case.

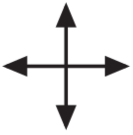
First: find a strategy for $k = 1$ or 2 or $k = \infty$. Then, for any k.

Dynamic programming led to reinforcement learning



Gridworld





3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

More optimizations

Name	Original Recurrence	Sufficient Condition of Applicability	Original Complexity	Optimized Complexity
Convex Hull Optimization1	$dp[i] = min_{j<i}\{dp[j] + b[j] \star a[i]\}$	$b[j] \geq b[j+1]$ ~~optionally~~ $a[i] \leq a[i+1]$	$O(n^2)$	$O(n)$
Convex Hull Optimization2	$dp[i][j] = min_{k<j}\{dp[i-1][k] + b[k] * a[j]\}$	$b[k] \geq b[k+1]$ ~~optionally~~ $a[j] \leq a[j+1]$	$O(kn^2)$	$O(kn)$
Divide and Conquer Optimization	$dp[i][j] = min_{k<j}\{dp[i-1][k] + C[k][j]\}$	$A[i][j] \leq A[i][j+1]$	$O(kn^2)$	$O(knlogn)$
Knuth Optimization	$dp[i][j] = min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$	$A[i, j-1] \leq A[i, j] \leq A[i+1, j]$	$O(n^3)$	$O(n^2)$

Richard Bellman (1920–1984)

- Man of the century
- Invented dynamic programming (1952) before programming was invented (1953)

Bellman's Principle of Optimality

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*