Dynamické programování

Samuel Štěpán

Prosinec 2023

1 Úvod

Dynamické programování je optimalizační technika založená na dekompozici komplexního optimalizačního problému na posloupnost jednodušších problémů takovým způsobem, že celkový potřebný čas na vyřešení je menší než v případě originálního problému. Jedná se spíše o obecný princip než o konkrétní optimalizační metodu.

Definice 1 (Sekvenční rozhodovací proces)**.** *Uvažujeme diskrétní dynamický systém modelovaný stavovou rovnicí*

$$\boldsymbol{x}_{t+1} = \boldsymbol{h}_t(\boldsymbol{x}_t, \boldsymbol{u}_t), \quad t = 0, 1, 2, \ldots, T,$$

kde $\boldsymbol{x}_t$ *je vektor stavových proměnných na v čase* t *a* $\boldsymbol{u}_t$ *je vektor řídících proměnných. Pro daný počáteční stav* $\boldsymbol{x}_0$ *chceme najít posloupnost řídících vektorů* $(\boldsymbol{u}_0^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_{T-1}^*)$ *takovou, že korespondující optimální trajektorie* $(\boldsymbol{x}_0^*, \boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_T^*)$ *minimalizuje účelovou funkci*

$$\sum_{t=1}^{T-1} f_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + F_T(\boldsymbol{x}_T, \tag{1}$$

kde $\sum_{t=1}^{T-1} f_t(\boldsymbol{x}_t, \boldsymbol{u}_t)$ *představuje cenu trajektorie a* $F_T(\boldsymbol{x}_T)$ *cenu koncového stavu.*

Definice 2 (Separabilita účelové funkce)**.** *Účelová funkce* (1) *je separabilní, pokud pro každé číslo* $r \in \mathbb{N}, r < T - 1$*, kontribuce posledních* r *stavů (tedy* $\sum_{t=T-r}^{T-1} f_t(\boldsymbol{x}_t, \boldsymbol{u}_t)$ *záleží pouze na současném stavu* $\boldsymbol{x}_{T-r}$ *a* r *řídících vektorů* $\boldsymbol{u}_{T-r}, \ldots, \boldsymbol{u}_{T-1}$*. Dále pro trajektorii platí podobná vlastnost. Dosažitelnost stavu* $\boldsymbol{x}_{t+1}$ *ze stavu* $\boldsymbol{x}_t$ *záleží pouze na řídícím vektoru* $\boldsymbol{u}_t$ *a ne na historii* $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{t-1}$*. Následkem separability dostáváme* ***princip optimality***

Definice 3 (Princip optimality)**.** *Optimální strategie* $(\boldsymbol{u}_0^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_{T-1}^*)$ *je taková, že pro jakýkoliv počáteční stav* $\boldsymbol{x}_0$ *a první řídící vektor* $\boldsymbol{u}_0^*$ *následující posloupnost řídících vektorů* $(\boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_{T-1}^*)$ *je optimální strategie* $(T-1)$*- stupňové úlohy s počátečním stavem* $\boldsymbol{x}_1 = \boldsymbol{h}_0(\boldsymbol{x}_0, \boldsymbol{u}_0^*)$*.*

Definice 4 (Bellmanova rovnice)**.** *Pokud je účelová funkce* (1) *separabilní, můžeme definovat rekurzivní funkcionální rovnici určující optimální strategii*

$$V_t(\boldsymbol{x}_t) = \min_{\boldsymbol{u}_t}\{f_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + V_{t+1}(\boldsymbol{h}_t(\boldsymbol{x}_t, \boldsymbol{u}_t))\}, \quad t = 0, 1, \ldots, T-1, \tag{2}$$

$$V_T(\boldsymbol{x}_T = F_T(\boldsymbol{x}_T \tag{3}$$

kde $F_T(\boldsymbol{x}_T)$ *představuje okrajovou podmínku, hodnotová funkce* $V_t(\boldsymbol{x}_t)$ *představuje celkovou cenu trajektorie určenou optimální strategií začínající ze stavu* $\boldsymbol{x}_t$*.*

2 Příklady

2.1 Fibonacciho posloupnost

Na Fibonacciho posloupnosti si ukážeme problém naivní rekurze, aplikaci memoizace a přístupu zdola nahoru. Fibonacciho n-té číslo se definuje jako

$$F_n = F_{n-1} + F_{n-2},$$
$$F_1 = F_2 = 1.$$

Tento problém můžeme snadno vyřešit v Pythonu pomocí rekurzivního algoritmu

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

Tento algoritmus je velmi neefektivní, časová náročnost je exponenciální. Tento algoritmus počítá některé hodnoty vícekrát (viz příklad v přiloženém souboru), proto se zavádí technika tzv. memoizace. Když se poprvé spočte hodnota F_k, pro $k \in \mathbb{N}, k < n$ tak se uloží a v případě dalšího volání funkce F_k se tato hodnota vrátí. Upravený algoritmus vypadá následovně

```
memo = {1:1, 2:1}
def fib(n):
    if(n in memo):
        return memo[n]
    else:
        f = fib(n-1) + fib(n-2)
        memo[n] = f
        return f
```

Tento algoritmus má již lineární časovou náročnost a je efektivní. Můžeme si povšimnout, že v tomto případě není rekurze nutná, můžeme začít z F_3 a dostat se až k F_n

```python
def fib(n):
    fib_d = {1:1, 2:1}
    for k in range(3,n+1):
        fib_d[k] = fib_d[k-1] + fib_d[k-2]
    return fib_d[n]
```

Tento algoritmus je ekvivalentní předchozímu a také má lineární časovou náročnost.

2.2 Nejkratší cesta

Hledáme nejkratší cestu v orientovaném grafu $G = (V, E, w, s, t)$, kde V je množina uzlů, E je množina hran, w je zobrazení $E \mapsto \mathbb{R}^+$, $s \in V$ je počáteční uzel a $t \in V, t \neq s$ je cílový uzel. Definujme funkci $\delta(u, v)$ jako hodnotu nejkratší cesty z uzlu u do uzlu v.
Dynamické programování se zakládá na hledání řešení jednotlivých podproblémů. U fibonacciho n-tého čísla byly jednotlivé podproblémy fibonacciho čísla $n-1$ a $n-2$. Pokud máme acyklický graf můžeme jako podproblémy nejkratší cesty z uzlu s do t definovat nejkratší cestu $\delta(u, t)$ pro všechny uzly $u \in V, (s, u) \in E$, tím získáme rekurzivní vztah

$$\delta(s,t) = \min_{u,(s,u)\in E} \{w(s,u) + \delta(u,t)\},$$
$$\delta(t,t) = 0.$$

Alternativní způsob je definovat nejkratší cestu $\delta(s, u)$ pro všechny uzly $u \in V, (u, t) \in E$, tím získáme rekurzivní vztah

$$\delta(s,t) = \min_{u,(u,t)\in E} \{w(u,t) + \delta(s,u)\}.$$
$$\delta(s,s) = 0.$$

Opět je potřeba použít memoizaci a pro sestavení nejkratší cesty je potřeba si ukládat $\text{argmin}_{u,(u,v)\in E}\{w(u, v) + \delta(s, u)\}$ pro každé v. Pokud do nějakého uzlu u nevedou žádné hrany tak volíme $\delta(s, u) = \infty$.
Pro cyklické orientované grafy se nejedná o stromovou strukturu a tento algoritmus by se mohl zacyklit, pro tyto případy je potřeba použít *Bellman-Ford* algoritmus.
Algoritmy a jejich testy lze nalézt v přiloženém souboru.

3 Závěr

Uvedli jsme 2 nejzákladnější příklady dynamického programování v diskrétním případě. U spojitých problémů (jako například *optimal consumption problem* je potřeba stavový prostor diskretizovat). Dynamické programování by se dalo popsat jako technika chytrého "brute force", typický příklad je problém balení batohu. Když bychom nepoužili dynamické programování tak musíme prozkoumat 2^n možných stavů, jelikož lze tento problém rozložit na jednotlivé podproblémy lze časovou náročnost snížit na pseudopolynomiální ($O(n * S)$, kde n

je počet předmětů a S je kapacita batohu).
Pro zájemce o toto téma doporučuji tyto online zdroje: `https://youtu.be/jTjRGe0wRvI?si=RTsrDI7WhSEKYH9w` (Dynamické programování pro začátečníky z pohledu informatiky), `https://www.youtube.com/watch?v=OQ5jsbhAv_M&list=PLZES21J5RvsHOeSW9Vrvo0EEc2juNe3tX` (Přednášky o dynamickém programování z MIT, můžete zde nalézt algoritmus na hraní Black Jacku, jak správně zobrazit text, jak nalézt nejdelší možný podřetězec, jak vyřešit plnění batohu, jak hrát nejlépe Tetris a Super Mario Brothers.