INFORMATIKA

Nalezení nejlepšího řešení

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

V nedávném článku [1] jsme si ukázali princip dynamického programování a jeho využití při řešení kombinatorických úloh typu *Kolika různými způsoby lze něco udělat?* Druhým velkým okruhem úloh, v nichž využijeme dynamické programování, jsou optimalizační úlohy typu *Jak nejlépe lze něco udělat?* Těm bude věnován dnešní článek.

Základní schéma řešení těchto úloh zůstane stejné, jako jsme si ho představili již v článku [1]. Nedokážeme sice přímo vyřešit úlohu pro velká vstupní data velikosti N, ale odvodíme způsob, jak lze toto řešení nalézt s využitím výsledků úloh téhož typu pro menší vstupní data. Vyřešíme tedy nejprve triviální úlohu pro vstup $N = 1$, pomocí jejího výsledku pak úlohu velikosti 2, s využitím výsledků těchto dvou úloh vyřešíme případ $N = 3$, atd. Takto postupujeme dále, až se dostaneme k řešení úlohy původně požadované velikosti.

Výsledky všech menších vyřešených úloh si budeme ukládat do tabulky, abychom je mohli využít v budoucnu, až budeme řešit úlohy větší. V nejjednodušším případě vystačíme s jednorozměrným polem t velikosti N, v němž každá uložená hodnota $t[i]$ udává výsledek úlohy pro vstupní data velikosti i. K obtížnějším úlohám s dvojrozměrnou tabulkou t se vrátíme někdy později v jiném článku.

Při určování počtu možností v článku [1] jsme počítali hodnotu $t[i]$ vždy jako součet některých vhodných hodnot $t[k]$ pro $k < i$. Nyní při řešení optimalizačních úloh bude změna spočívat pouze v tom, že při výpočtu $t[i]$ nebudeme sčítat, ale budeme vybírat maximum nebo minimum z vhodných hodnot $t[k]$ pro $k < i$. Předvedeme si to na řešení dvou konkrétních příkladů.

Podobně jako v [1] začneme u jednoduchého grafového problému. Máme dán orientovaný acyklický graf s N vrcholy, jehož vrcholy jsou očíslovány od 1 do N v topologickém uspořádání. Připomeňme si, že každý orientovaný graf bez cyklů lze topologicky uspořádat, tzn. očíslovat jeho vrcholy navzájem různými čísly od 1 do N takovým způsobem, aby každá hrana vedla z vrcholu s menším číslem do vrcholu s větším číslem. Algoritmem nalezení topologického uspořádání se zde zabývat nebudeme a předpokládáme, že zadaný graf je již topologicky uspořádán. Naším úkolem je určit délku nejdelší cesty z vrcholu 1 do vrcholu N.

Úlohu je možné řešit prohledáváním zadaného grafu do hloubky. Prohledávání začneme ve startovním vrcholu 1 a průběžně si budeme pamatovat, v jaké vzdálenosti od něj se momentálně nacházíme. To lze snadno implementovat rekurzivní funkcí, která bude v jednom parametru dostávat číslo zpracovávaného vrcholu a ve druhém parametru jeho vzdálenost od startovního vrcholu 1. Při každém příchodu do cílového vrcholu N porovnáme aktuální délku cesty s dosud nalezeným maximem a hodnotu maxima případně zlepšíme (tzn. zvýšíme, protože hledáme nejdelší cestu).

V následující ukázkové implementaci předpokládáme, že zadaný graf je reprezentován maticí sousednosti g. Hodnota $g[x][y]$ je typu boolean a udává, zda v grafu existuje orientovaná hrana z vrcholu x do vrcholu y. Vzhledem k topologickému uspořádání grafu nás zajímají pouze takové hodnoty $g[x][y]$, v nichž $x < y$. Jiné hrany v topologicky uspořádaném grafu totiž nemohou existovat.

```
def maxdelka(g, N):
    maximum = 0

    def pruchod(v, delka):
        nonlocal maximum
        if v == N:
            maximum = max(delka, maximum)
        else:
            for k in range(v+1, N+1):
                if g[v][k]: pruchod(k, delka+1)

    pruchod(1, 0)
    return maximum
```

Uvedené řešení sice dává správný výsledek, ale podstatnou nevýhodou je jeho neefektivita. Počet různých cest z vrcholu 1 do vrcholu N v ori-

entovaném grafu může být až exponenciální vzhledem k počtu vrcholů N. Popsaný postup řešení postupně prochází každou z těchto cest, takže jeho asymptotická časová složitost je v nejhorším případě exponenciální vzhledem k N.

Ukážeme si podstatně lepší řešení založené na metodě dynamického programování. Pro jednotlivé vrcholy $i = 1, 2, \ldots, N$ budeme postupně počítat délku nejdelší cesty z vrcholu 1 do vrcholu i. Tyto délky si budeme ukládat pro další využití do pole t jako hodnoty $t[i]$. Je zřejmé, že $t[1] = 0$. Ukážeme, jak spočítáme $t[i]$ pro $i > 1$, když už známe hodnoty $t[1]$, $t[2]$, …, $t[i-1]$. Uvažujme jednu libovolnou cestu vedoucí z vrcholu 1 do vrcholu i. Jejím předposledním vrcholem je nějaký vrchol k. Podle topologického uspořádání je $k < i$, takže při výpočtu $t[i]$ již známe hodnotu $t[k]$ neboli délku nejdelší cesty z vrcholu 1 do vrcholu k. Pak tedy nejdelší cesta z vrcholu 1 do vrcholu i vedoucí přes vrchol k má délku $t[k]+1$. Při výpočtu $t[i]$ proto stačí uvažovat všechny vrcholy k takové, že v grafu existuje hrana z vrcholu k do vrcholu i. Hodnotu $t[i]$ určíme jako maximum z hodnot $t[k] + 1$ pro všechny takové vrcholy k.

Uvedeným postupem spočítáme po řadě hodnoty $t[2]$, $t[3]$, …, $t[N]$. Poslední z nich $t[N]$ je hledaným výsledkem naší úlohy. Výpočet každé hodnoty $t[i]$ má časovou složitost $\mathrm{O}(N)$ a těchto hodnot je $N - 1$, takže asymptotická časová složitost celého řešení je $\mathrm{O}(N^2)$. Prostorová složitost je $\mathrm{O}(N)$, neboť si musíme pamatovat všech N počítaných hodnot $t[i]$ pro případ, že je budeme ještě potřebovat. Některé z nich si možná ukládáme zbytečně a už je nepoužijeme, ale to dopředu nevíme.

Popsaný algoritmus můžeme zapsat jako funkci v Pythonu. Na první pohled se nabízí toto jednoduché řešení:

```
def maxdelka(g, N):
    t = [0] * (N+1)
    for i in range(2, N+1):
        for k in range(1, i):
            if g[k][i]: t[i] = max(t[i], t[k]+1)
    return t[N]
```

Uvedená funkce vypadá sice dobře a pro některé vstupy dává správné výsledky, má ale jednu velmi podstatnou vadu: obecně nefunguje správně. Představte si třeba graf se čtyřmi vrcholy a s hranami $(1, 4)$, $(2, 3)$, $(3, 4)$. Jediná cesta z vrcholu 1 do vrcholu 4 má délku 1, ale naše funkce vrátí chybný výsledek 2. Počítá totiž délku nejdelší cesty vedoucí do vrcholu N, ale ne nutně ze startovního vrcholu 1. Když si uvědomíme tento problém,

oprava funkce je již docela snadná:

```
def maxdelka(g, N):
    t = [None] * (N+1)
    t[1] = 0
    for i in range(2, N+1):
        for k in range(1, i):
            if g[k][i] and t[k] != None:
                if t[i] == None or t[i] < t[k]+1:
                    t[i] = t[k]+1
    return t[N]
```

Přirozeným rozšířením naší úlohy je požadavek určit nejen délku nejdelší cesty v grafu z vrcholu 1 do vrcholu N, ale nalézt i tuto cestu a vypsat ji jako posloupnost čísel procházených vrcholů. To lze řešit při využití dynamického programování dvěma základními způsoby.

První možností je uložit si pro každé $i = 1, 2, \ldots, N$ zároveň s hodnotou $t[i]$ také celou nejdelší cestu vedoucí z vrcholu 1 do vrcholu i. Hodnotu $t[i]$ jsme získali jako $t[k] + 1$ díky nějakému konkrétnímu vrcholu k a nejdelší cestu vedoucí do vrcholu i dostaneme tak, že dříve nalezenou cestu do vrcholu k nyní jednoduše prodloužíme o vrchol i. Nevýhodou tohoto řešení je kvadratická prostorová složitost vzhledem k N. Postupně si ukládáme N cest, z nichž každá má délku $\mathrm{O}(N)$.

Výhodnější je proto druhá možnost, jak lze k řešení přistoupit. Pro každé $i = 1, 2, \ldots, N$ si zároveň s hodnotou $t[i]$ uložíme pouze jedno číslo: číslo toho vrcholu k, z něhož jsme přišli do vrcholu i po nejdelší cestě (tedy takové k, pro které $t[i] = t[k] + 1$). Toto číslo si uložíme do druhého jednorozměrného pole p. Bude tedy platit $p[i] = k$, údaj $p[i]$ představuje předchůdce vrcholu i na optimální cestě vedoucí z vrcholu 1 do vrcholu i. Po spočítání všech dvojic hodnot $t[i]$, $p[i]$ bude $t[N]$ udávat délku nejdelší cesty a z uložených hodnot $p[i]$ nyní dodatečně sestrojíme odzadu trasu nejdelší cesty. Tato cesta končí vrcholem N, do něj jsme přišli z vrcholu $p[N]$, tomu předchází vrchol $p[p[N]]$ atd., dokud nedojdeme ke startovnímu vrcholu 1.

Asymptotická časová složitost algoritmu zůstává $\mathrm{O}(N^2)$ stejně jako u původní úlohy, v níž jsme hledali pouze délku nejdelší cesty. Ukládání hodnot $p[i]$ tuto složitost nijak neovlivní a závěrečná zpětná rekonstrukce nejdelší cesty má časovou složitost pouze $\mathrm{O}(N)$, neboť každá cesta v grafu má délku nejvýše N. Prostorová složitost zůstává $\mathrm{O}(N)$, místo jednoho pole zde používáme dvě stejně velká pole velikosti N.

```
def maxcesta(g, N):
    t = [None] * (N+1)
    p = [None] * (N+1)
    t[1] = 0
    for i in range(2, N+1):
        for k in range(1, i):
            if g[k][i] and t[k] != None:
                if t[i] == None or t[i] < t[k]+1:
                    t[i] = t[k]+1
                    p[i] = k

    cesta = [N]
    v = N
    while v != 1:
        v = p[v]
        cesta.append(v)
    return list(reversed(cesta))
```

Náš druhý ukázkový příklad je ze zcela jiné oblasti, týká se plánování a rozvrhování. Postup jeho řešení metodou dynamického programování bude ovšem velmi podobný jako u předchozí grafové úlohy. Máme zadánu posloupnost N po sobě jdoucích úkolů ($N > 1$), které máme vykonat. Úkoly označíme čísly 1, 2, ..., N. Pro každý z úkolů známe čas $c[i]$, jak dlouho nám práce na i-tém úkolu bude trvat. Úkoly musíme provádět postupně v uvedeném pořadí, nemůžeme pracovat na více úkolech zároveň. Některé ze zadaných úkolů smíme vynechat, ale nikdy nesmíme vynechat žádné dva po sobě jdoucí úkoly. Máme zjistit, jak nejrychleji dokážeme za těchto podmínek zadané úkoly splnit.

Situaci ilustruje následující příklad: pro $N = 10$ a posloupnost časů 3 4 8 2 1 3 1 1 6 3 je správným výsledkem 12. Vykonáme úkoly s pořadovými čísly 2, 4, 5, 7, 8, 10, jejich celková délka je

$$4 + 2 + 1 + 1 + 1 + 3 = 12.$$

Nižšího výsledného času potřebného na splnění vybraných úkolů nelze dosáhnout.

Na této úloze si ukážeme, že i řešení dynamickým programováním můžeme někdy navrhnout více různými způsoby. V každém případě si budeme postupně počítat a ukládat vhodné hodnoty $t[i]$ pro $i = 1, 2, \ldots, N$. Důležité ovšem je nejdříve jasně rozhodnout, co tyto hodnoty budou přesně

znamenat. V tom se právě mohou různé způsoby řešení lišit. Předvedeme si dvě takové možnosti.

Předpokládejme nejprve, že $t[i]$ určuje minimální čas, v němž lze splnit podle stanovených pravidel posloupnost úkolů 1, 2, …, i, pokud i-tý zadaný úkol je tím posledním, který jsme vykonali. Zřejmě platí $t[1] = c[1]$, $t[2] = c[2]$. Potřebujeme nalézt vztah, jak spočítat hodnotu $t[i]$ pro $i > 2$. Protože i-tý úkol určitě vykonáme, započítáme si čas $c[i]$. Předcházet mu může splněný úkol $i - 1$ nebo $i - 2$ a pro ně již známe optimální řešení $t[i-1]$, resp. $t[i-2]$. Pro výpočet hodnoty $t[i]$ vezmeme menší z nich, neboť hledáme minimální možný čas. Platí tedy $t[i] = c[i] + \min(t[i-1], t[i-2])$. Takto spočítáme postupně všechny hodnoty $t[3]$, …, $t[N]$. Řešením celé úlohy pak bude $\min(t[N], t[N-1])$, neboť posledním splněným úkolem může být buď celkově poslední, nebo předposlední úkol ze zadané posloupnosti.

```
def mincas(c):      # c = seznam N časů jednotlivých úkolů
    N = len(c)
    c = [None] + c # úkoly číslujeme od 1 do N
    t = [None] * (N+1)
    t[1], t[2] = c[1], c[2]
    for i in range(3, N+1):
        t[i] = c[i] + min(t[i-1], t[i-2])
    return min(t[N], t[N-1])
```

Pokud by nás navíc zajímalo, které úkoly máme vykonat, abychom dosáhli minimálního možného času, upravíme toto řešení obdobným způsobem, jako jsme to provedli v úvodní úloze o nejdelší cestě v grafu. Při každém výpočtu hodnoty $t[i]$ si uložíme do $p[i]$ informaci, který úkol předcházel i-tému úkolu v optimálním řešení. Z hodnot $p[i]$ pak na závěr zrekonstruujeme odzadu posloupnost vykonaných úkolů.

```
def ukoly(c):       # c = seznam N časů jednotlivých úkolů
    N = len(c)
    c = [None] + c # úkoly číslujeme od 1 do N
    t = [None] * (N+1)
    p = [None] * (N+1)
    t[1], t[2] = c[1], c[2]
    for i in range(3, N+1):
        t[i] = c[i] + min(t[i-1], t[i-2])
        p[i] = i-1 if t[i-1] < t[i-2] else i-2
```

```
    v = N if t[N] < t[N-1] else N-1  # poslední splněný úkol
    vyber = [v]
    while p[v] != None:
        v = p[v]
        vyber.append(v)
    return list(reversed(vyber))
```

V našem druhém řešení téže úlohy o posloupnosti úkolů stanovíme, že hodnota $t[i]$ bude určovat minimální čas, v němž lze splnit podle stanovených podmínek posloupnost úkolů 1, 2, …, i. Poslední i-tý úkol přitom můžeme, ale také nemusíme vykonat. Právě toto je jediný rozdíl od předchozího řešení. Celý další postup bude velmi podobný, ale všechny vzorce tentokrát vycházejí trochu odlišně. V tomto případě bude platit $t[1] = 0$, $t[2] = \min(c[1], c[2])$. Za hodnotu $t[i]$ pro $i > 2$ vezmeme výhodnější ze dvou možností podle toho, zda v pořadí i-tý úkol vykonáme, nebo ne. Pokud ano, započítáme čas $c[i]$ a optimálně splníme předchozích $i-1$ úkolů. Pokud nebudeme provádět úkol číslo i, musíme nutně vykonat úkol $i-1$, takže započteme čas $c[i-1]$ a optimálně splníme předchozích $i-2$ úkolů. Platí tedy

$$t[i] = \min(c[i] + t[i-1], c[i-1] + t[i-2]).$$

Řešením celé úlohy je v tomto případě hodnota $t[N]$.

```
def mincas(c):     # c = seznam N časů jednotlivých úkolů
    N = len(c)
    c = [None] + c # úkoly číslujeme od 1 do N
    t = [None] * (N+1)
    t[1], t[2] = 0, min(c[1], c[2])
    for i in range(3, N+1):
        t[i] = min(c[i] + t[i-1], c[i-1] + t[i-2])
    return t[N]
```

Pokud při tomto způsobu řešení chceme nalézt seznam skutečně vykonaných úkolů, budeme si při každém výpočtu hodnoty $t[i]$ ukládat do $p[i]$ informaci, který úkol jsme se rozhodli vykonat v i-tém kroku výpočtu (zda úkol i, nebo úkol $i-1$). Z těchto hodnot pak opět dokážeme zrekonstruovat odzadu výslednou posloupnost splněných úkolů.

```
def ukoly(c):      # c = seznam N časů jednotlivých úkolů
    N = len(c)
    c = [None] + c # úkoly číslujeme od 1 do N
    t = [None] * (N+1)
    p = [None] * (N+1)
    t[1], t[2] = 0, min(c[1], c[2])
    p[2] = 1 if t[2] == c[1] else 2
    for i in range(3, N+1):
        t[i] = min(c[i] + t[i-1], c[i-1] + t[i-2])
        p[i] = i if t[i] == c[i] + t[i-1] else i-1

    vyber = []
    v = N
    while v > 1:
        vyber.append(p[v])
        v = p[v] - 1
    return list(reversed(vyber))
```

Obě popsané varianty řešení druhé úlohy jsou zcela rovnocenné z hlediska správnosti i z hlediska efektivity. Záleží tedy jenom na vás, který pohled na dynamické programování vám při řešení nějaké konkrétní úlohy bude připadat přirozený a názornější. Oba algoritmy mají asymptotickou časovou složitost $\mathrm{O}(N)$, protože počítají postupně N hodnot $t[i]$ (případně také N hodnot $p[i]$) a každou z nich získají v konstantním čase. Rovněž zpětná rekonstrukce posloupnosti úloh má vždy časovou složitost $\mathrm{O}(N)$. Prostorová složitost všech uvedených řešení je $\mathrm{O}(N)$, pracuje se v nich se seznamy t, případně p, délky N.

Jistě jste si všimli, že v obou výše uvedených variantách řešení naší druhé úlohy jsme pro výpočet $t[i]$ potřebovali pouze dvě bezprostředně předcházející hodnoty $t[i-1]$, $t[i-2]$. Pokud tedy budeme určovat jenom minimální čas, seznam t délky N ani nepotřebujeme ukládat a místo něj vystačíme se dvěma proměnnými. Řešení tak může mít dokonce konstantní prostorovou složitost. Při určování posloupnosti splněných úkolů se ovšem nevyhneme ukládání všech hodnot do seznamu p a prostorová složitost v takovém případě proto bude lineární vzhledem k N.

Literatura

[1] *Töpfer, P.*: Zjištění počtu možností. Matematika–fyzika–informatika, roč. 34 (2025), č. 3, s. 219–224.