Dynamické programování

1 Základní princip

Princip dynamického programování je založen na podobné myšlence jako rozděl a panuj. Vstupní instanci vždy rozdělíme na několik menších instancí, z jejichž řešení sestavíme řešení původní instance. Rozdíl je v efektivitě s jakou to v dynamickém programování provedeme. Zvýšení efektivity lze shrnout do následujících bodů.

1. *Omezíme opakovaný výpočet pro stejné podinstance*, ke kterému dochází u přístupu rozděl a panuj tak, že si výsledky pamatujeme (například v tabulce).
2. *Podinstance, které se v průběhu výpočtu vyskytnou, vhodně uspořádáme.* Označme jako $I_i \lesssim I_j$ situaci, kdy k výpočtu výsledku podinstance I_j potřebujeme znát řešení podinstance I_i. Dynamické programování můžeme použít pouze tehdy, pokud v množině všech podinstancí uspořádané podle $\lesssim$ nejsou cykly. To znamená, že neexistuje žádná instance, k jejímuž výpočtu potřebujeme znát výsledek jí samotné. Situaci lze visualizovat pomocí orientovaného grafu. Za vrcholy grafu vezmeme všechny podinstance, z I_i vede hrana do I_j právě když $I_i \lesssim I_j$. Pak lze dynamické programování použít, právě když nejsou v tomto grafu cykly. Graf bez cyklů totiž lze linearizovat. To znamená, že nalezneme takové lineární uspořádání $\leq$ podinstancí, že pro každé dvě podinstance platí, že pokud $I_i \lesssim I_j$, pak i $I_i \leq I_j$. Graf odpovídající $\leq$ je pak řetězec. Situace je demonstrována na obrázku 1.
3. *Podinstance řešíme od nejmenší* podle jejich lineárního uspořádání. Předchozí bod zajišťuje, že vždy známe řešení všech podinstancí, které jsou k sestavení řešení pro aktuální podinstanci potřeba.
4. *Ze vstupní instance jsme schopni najít nejmenší podinstance*, přesněji pro takové, které jsou podle $\lesssim$ minimální (v grafu do nich nevede žádná hrana). Toto je potřeba, abychom mohli výpočet spustit. Pro další podinstance to není nutné (i když je to občas možné), lze je hledat až v průběhu výpočtu.

Princip demonstrujeme na problému výpočtu n-tého Fibonacciho čísla. Připomeňme si, že Fibonacciho čísla je definována následujícím rekurentním vztahem

$$F(n) = \begin{cases} F(n-1) + F(n-2) & n \geq 2 \\ n & n \in \{0, 1\} \end{cases} \tag{1}$$

Nabízí se použít algoritmus rozděl a panuj, který následuje

Algoritmus 1 n-té Fibonacciho číslo pomocí rozděl a panuj

```
1: procedure FibDQ(n)
2:     if n = 0 or n = 1 then
3:         return n
4:     end if
5:     return FibDQ(n − 1) + FibDQ(n − 2)
6: end procedure
```

Složitost FibDQ můžeme vyjádřit jako rekurenci

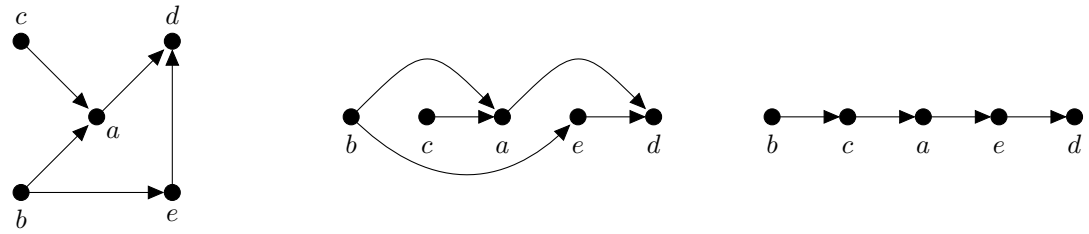
$$T(n) = T(n-1) + T(n-2) + O(n),$$

odhadem jejíhož řešení je $O(2^n)$ (lze snadno vidět metodou stromu). Algoritmus má tedy velmi nevýhodnou složitost. Pokud si visualizujeme strom rekurzivního volání, zjistíme, že problém je v opakovaném volání procedury pro stejný argument, viz. obrázek 2 (a). Problém odstraníme tak, že si pro každé číslo, pro které jednou zavoláme FibDQ, zapamatujeme výsledek, který vrátil. Při opakovaném rekurzivním volání pro tuto podinstanci zapamatovanou hodnotu použijeme. To přesně odpovídá prvnímu principu dynamického programování.

Složitost FibTab je lineární. Lze snadno vidět, že k rekurzivnímu volání na řádku 12 dojde pro každé číslo (a tedy pro každou podinstanci) maximálně jedenkrát. Ve zbylých případech je vrácena v tabulce zapamatovaná hodnota. Strom rekurzivních volání pak vypadá jako na obrázku 2 (b).

Algoritmus 2 n-té Fibbonaciho číslo s pamatováním si mezivýsledků

```
1: procedure PrepareTable(n)
2:     t[0] ← 0
3:     t[1] ← 1
4:     for i ← 2 to n do
5:         t[i] ← −1
6:     end for
7:     return t
8: end procedure
9:
10: procedure FibHelp(n, t)
11:     if t[n] = −1 then
12:         t[n] ←FibTab(n − 1) + FibTab(n − 2)
13:     end if
14:     return t[n]
15: end procedure
16:
17: procedure FibTab(n)
18:     return FibHelp(n, PrepareTable(n))
19: end procedure
```



(a) Orientovaný graf bez cyklů (b) Lineární uspořádání vrcholů (c) Odpovídající lineární tvar

Obrázek 1: Linearizovaná podoba grafu

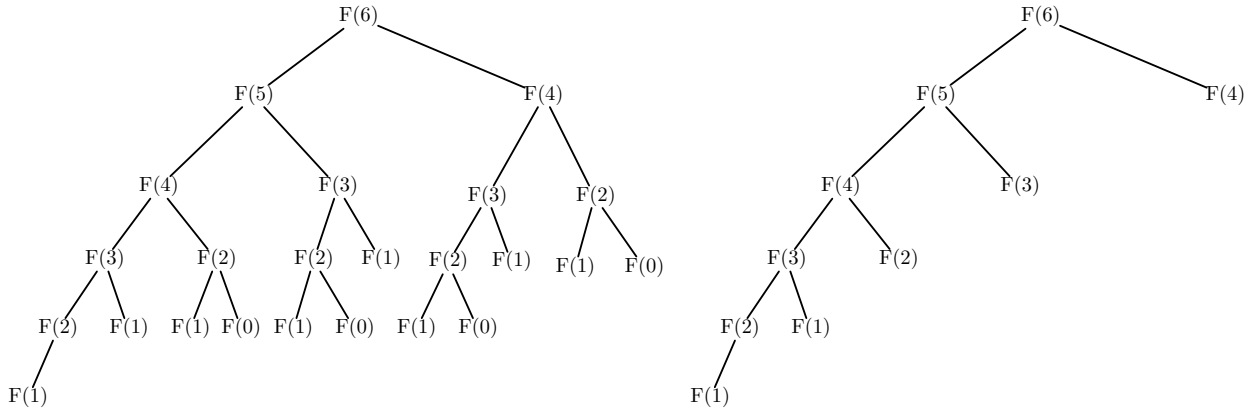
Složitost algoritmu můžeme ještě zlepšit. Zlepšíme ji sice jenom o faktor 2, který z pohledu asymptotické notace nezanedbatelný, nicméně dostaneme mnohem hezčí algoritmus. Také zmenšíme paměťovou složitost z lineární (potřebujeme celou tabulku) na konstantní. Klíčem je všimnout si, že na problém výpočtu Fibonacciho čísla lze aplikovat i zbylé principy dynamického programování. Závislosti mezi jednotlivými podinstancemi totiž neobsahují cyklus. Lze je tedy linearizovat, viz obrázek 3. Navíc umíme pro jakékoliv n sestavit podinstance, kterými výpočet zahájíme: jsou to právě čísla 0 a 1. Pro vypočtení n-tého Fibonacciho čísla tedy stačí procházet čísla od 0 do n a pamatovat si vždy 2 poslední výsledky.

2 Příklady algoritmů

Při návrhu algoritmu dynamického programování je dobré vzít v úvahu následující

1. Je výhodné, pokud má vstupní instance vhodné vnitřní uspořádání, na jejímž základě lze generovat podinstance. Je-li toto uspořádání lineární, lze podinstance generovat například jako prefixy nebo jako intervaly. Příkladem instancí s vnitřním lineárním uspořádáním jsou řetězce, posloupnosti čísel apod. Je-li vnitřní uspořádání instance komplikovanější (např. graf), jsou komplikovanější i podinstance (např. podgraf).

2. Je dobré uvažovat také o ceně řešení podinstancí, nikoliv pouze o řešeních samotných. Stejně tak o tom, jak zkombinovat cenu řešení podinstancí do ceny řešení aktuální instance (místo pouze kombinování řešení podinstancí do řešení aktuální instance). Algoritmus typicky nejdříve počítá cenu řešení aktuální instance z cen řešení podinstancí a vypočte tak cenu řešení původní instance. Vlastní řešení pak lze dohledat ze způsobu generování podinstancí (předchozí bod) a vztahu mezi jejich cenami.



(a) FibDQ (b) FibTab

Obrázek 2: Rekurzivní strom volání výpočtu Fibbonaciho čísla

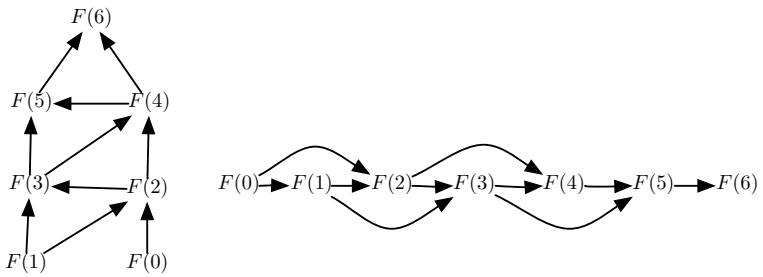
Algoritmus 3 n-té Fibonacciho číslo pomocí dynamického programování

```
1: procedure FibIdeal(n)
2:     if n = 0 then
3:         return n
4:     end if
5:     a ← 0
6:     b ← 1
7:     for n ← 2 to n − 1 do
8:         c ← a + b
9:         a ← b
10:        b ← c
11:    end for
12: end procedure
```

3. Počet podinstancí je klíčem k určení složitosti algoritmu. S pouze pár výjimkami platí, že čím méně podinstancí tím lépe.

2.1 Úloha batohu

Úloha batohu (KNAPSACK), jejíž definice následuje, je optimalizačním problémem. Neformálně můžeme popsat KNAPSACK například následovně. Zloděj se vloupal do banky a chce si odnést zlatý písek. Má k dispozici batoh, který má omezený objem. Zlatý písek je v bance uskladněn v pytlících, které mohou mít různý objem. Každý z pytlíku může zloděj buď přesypat celý do batohu, nebo jej nesmí vůbec otevírat. Samozřejmě, cílem zloděje je odnést si v batohu co největší objem zlatého písku. Formálně je úloha batohu určena následovně:



Obrázek 3: Závislosti mezi podinstancemi pro vypočtení šestého Fibonacciho čísla.

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$sol(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leq b\}$
Cena řešení:	$cost(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Čísla w_i odpovídají objemům jednotlivých pytlů s pískem, b je kapacita batohu. Pro instanci $I = (b, w_1, \ldots, w_5)$, kde $b = 29, w_1 = 3, w_2 = 6, w_3 = 8, w_4 = 7, w_5 = 12$, existuje jediné optimální řešení $C = \{1, 2, 3, 5\}$ s cenou $cost(C, I) = 29$. Lze snadno ověřit, že všechna ostatní přípustná řešení mají menší cenu.

Uvažujme vstupní instanci $I = (b, w_1, \ldots, w_n)$. Abychom mohli použít dynamické programování, musíme najít vhodné podinstance. Protože složitost celého algoritmu závisí na počtu podinstancí, začneme hledat tak, aby počet podinstancí byl co nejmenší. Uspořádanou n-tici $w_1, \ldots, w_n$ můžeme považovat za vnitřní lineární uspořádání instance I. Podinstance budeme generovat jako prefixy. Označme si tedy jako $I(i, c)$ takovou podinstanci problému, ve které uvažujeme pouze prvních i členů $w_1, \ldots, w_n$ spolu s kapacitou c (přirozeně, $c \leq b$). Podívejme se, jakým způsobem můžeme z instance $I(i, c)$ vygenerovat podinstance. Protože generujeme prefixy, jsou jenom dvě možnosti

- pokud i (vzpomeňme si, že řešení jsou množiny indexů hodnot) patří do řešení instance $I(i, c)$, pak má podinstance tvar $I(i-1, c-w_i)$. Zařazením i do řešení jsme totiž snížili kapacitu, která zbývá pro zařazení prvků $1, \ldots, i-1$.
- pokud i nepatří do řešení instance $I(i, c)$, pak má podinstance tvar $I(i-1, c)$.

Z předchozího jde vidět, že podinstance lze uspořádat bez cyklů. Platí totiž, že $I(i-1, c-w_i) \lesssim I(i, c)$ a současně $I(i-1, c) \lesssim I(i, c)$. Minimální prvky vzhledem k tomuto uspořádání jsou vždycky

- $I(0, d)$ pro nějakou kapacitu d, což odpovídá prázdnému prefixu.
- $I(i, 0)$ pro nějaké i, což odpovídá prázdné kapacitě (a nemá cenu řešit, které prvky z $w_1, \ldots, w_{i-1}$ dám do řešení, protože kapacita je prázdná)

Nyní se podívejme na to, jak z cen řešení instancí $I(i-1, c)$ a $I(i-1, c-w_i)$ dostaneme cenu instance $I(i, c)$. Protože úloha batohu je maximalizační problém, chceme přirozeně získat řešení s co nejlepší cenou. Odtud

$$cost(I(i, c)) = \max(cost(I(i-1, c)), cost(I(i-1, c-w_i)) + w_i). \tag{2}$$

Dále pro minimální prvky platí, že $cost(I(0, d)) = 0$ (nemám prvky, které bych do řešení zařadil) a $cost(I(i, 0)) = 0$ (prázdná kapacita a nemůžu do řešení zařadit žádné prvky).

Nyní si už můžeme napsat algoritmus, s pomocí kterého spočítáme cenu řešení $I(n, b)$. Zbývá si říci, jak nalézt skutečné řešení, nikoliv jeho cenu. Toho lze dosáhnout „zpětným inženýrstvím“ vztahu (2). Vezmeme cenu $I(n, b)$, pokud $b - w_n < 0$, pak n do řešení nepatří a posuneme se k podinstanci $I(n-1, b)$; v opačném případě se podíváme na ceny $x = cost(I(n-1, b-w_n))$ a $y = cost(I(n-1, b))$. Pokud platí, že $x + w_n > y$, pak w_n do řešení patří, pokud $x + w_n < y$, pak w_n do řešení nepatří. Jsou-li si x a y rovny, můžeme zvolit jak chceme. Postup poté opakujeme pro tu podinstanci, jejíž cena byla větší. Algoritmus si uvádíme jako Algoritmus 5.

Na řádcích 2 až 16 vytváří KnapsackDP tabulku o rozměrech $n \cdot b$. Pro vyplnění každého políčka je potřeba konstantní počet operací. Složitost nalezení řešení na řádcích je 17 až 24 je lineární. Můžeme tedy konstatovat, že složitost algoritmu je $O(n \cdot b)$. Složitost algoritmu tedy není závislá jenom na velikosti instance měřené jako počet čísel $w_1, \ldots, w_n$, ale i na velikosti největšího čísla se v instanci vyskytujícího. Toto je zajímavá vlastnost. Pokud bychom například vynásobili všechna čísla v instanci, kterou jsme si uvedli jako příklad, milionem, KnapsackDP by počítal řešení milionkrát déle. Intuitivně bychom však řekli, že obě dvě instance jsou stejně komplikované.

Algoritmus 4 Úloha batohu pomocí dynamického programování

```
1: procedure KnapsackDP(w_1, ..., w_n)
2:     for i ← 0 to b do
3:         M[0, i] ← 0
4:     end for
5:     for i ← 0 to n do
6:         M[i, 0] ← 0
7:     end for
8:     for i ← 1 to n do
9:         for c ← 1 to b do
10:            if c < w_i then
11:                M[i, c] ← M[i − 1, c]
12:            else
13:                M[i, c] ← max(M[i − 1, c], M[i − 1, c − w_i] + w_i)
14:            end if
15:        end for
16:    end for
17:    S ← ∅
18:    c ← b
19:    for i ← n to 0 do
20:        if c − w_i ≥ 0 and M[i − 1, c] < M[i − 1, c − w_i] + w_i then
21:            S ← S ∪ {i}
22:            c ← c − w_i
23:        end if
24:    end for
25:    return S
26: end procedure
```

2.2 Nejkratší cesty v grafu

Definice 1. Nechť $G = (V, E)$ je graf. *Ohodnocení hran* grafu G je zobrazení $c : E \to \mathbb{Q}$ přiřazující hranám grafu jejich racionální hodnotu, $c(e)$ je pak ohodnocení hrany $e \in E$. Dvojici (G, c) říkáme hranově ohodnocený graf.

Definice 2. Cesta z uzlu s do uzlu t v grafu G je posloupnost vrcholů a hran $P = u_0, e_0, \ldots, e_{n-1}, u_n$ taková, že $e_i = \{u_{i-1}, u_i\}$, $u_0 = s$, $u_n = t$, a každý uzel $u \in P$ se vyskytuje v cestě právě jednou.

Definice 3. Cena $c(P)$ cesty P v ohodnoceném grafu $(G = (V, E), c)$ je suma ohodnocení všech hran vyskytujících se v cestě.

Definice 4. Nejkratší cesta z uzlu s do uzlu t v grafu G je taková cesta P, pro kterou platí, že $c(P) \leq c(R)$ pro každou cestu R z uzlu s do uzlu t. Cenu nejkratší cesty z s do t značíme jako $dist(s, t)$.

Problém nalezení nejkratších cest je dán následovně.

Nalezení nejkratší cesty	
Instance:	$(G = (V, E), c), s, t \in V$
Přípustná řešení:	$sol(G, s, t) = \{P \mid P$ je cesta z uzlu s do uzlu $t\}$
Cena řešení:	$cost(P, G, s, t) = c(P)$
Cíl:	minimum

Tedy, je-li dán spojitý ohodnocený graf a dva jeho uzly, chceme najít nejkratší cestu mezi těmito uzly. Ukážeme si algoritmus založený na dynamickém programování známý pod jménem *Floyd-Warshallův algoritmus*, který dokáže spočítat vzdálenosti mezi všemi dvojicemi uzlů.

Označme si vrcholy grafu jako $V = \{1, 2, \ldots, n\}$ a označme si jako $I(i, j, k)$ nejkratší cestu[1] mezi uzly i a j (pozor, tyto uzly nemusí být různé), která mimo je obsahuje pouze uzly menší než k (taková cesta nemusí existovat, pak předpokládáme, že je prázdná). Označme si cenu této cesty jako $dist(i, j, k)$. Cena prázdné cesty je ∞. Cestu $I(i, j, k + 1)$ můžeme obdržet z $I(i, j, k)$ podle následující úvahy:

- Pokud je $I(i, j, k)$ různá od $I(i, j, k + 1)$ (tj. je kratší), pak musí vést přes uzel $k + 1$. V tomto případě se tedy $I(i, j, k)$ rovná spojení cest $I(i, k + 1, k)$ a $I(k + 1, j, k)$. Toto je ekvivalentní podmínce

$$dist(i, j, k + 1) = dist(i, k + 1, k) + dist(k + 1, j, k) < dist(i, j, k). \tag{3}$$

- Pokud (3) neplatí, tak $I(i, j, k + 1) = I(i, j, k)$.

Pro každé k počítáme $I(i, j, k)$ pro všechny dvojice uzlů i, j. Je vidět, že odpovídající podinstance[2] lze uspořádat podle třetí komponenty. Minimální prvky jsou pak podinstance odpovídající $I(i, j, 0)$, což je buď cesta tvořena hranou mezi uzly i a j, v tomto případě je $dist(i, j, 0) = c(\{i, j\})$, nebo hrana mezi i a j neexistuje, pak nastavíme $dist(i, j, 0) = \infty$.

Algoritmus uvádíme jako Algoritmus 5.

Algoritmus 5 Nejkratší cesty v grafu

```
1:  procedure FLOYD-WARSHALL(G = (V, E), c)
2:      for i ← 1 to n do:
3:          for j ← 1 to n do:
4:              D[i, j] = ∞
5:              P[i, j] = ∞
6:          end for
7:      end for
8:      for {i, j} ∈ E do:
9:          D[i, j] = c({i, j})
10:     end for
11:     for k ← 1 to n do:
12:         for i ← 1 to n do:
13:             for j ← 1 to n do:
14:                 x ← D[i, k] + D[k, j]
15:                 if x < D[i, j] then
16:                     D[i, j] ← x
17:                     P[i, j] ← k
18:                 end if
19:             end for
20:         end for
21:     end for
22:     return ⟨D, P⟩
23: end procedure
24:
25: procedure GETPATH(P,i,j)
26:     mid ← P[i, j]
27:     if mid = ∞ then                          ▷ i a j jsou spojeny jednou hranou
28:         return []
29:     else
30:         return GETPATH(P, i, mid) + [mid]+GETPATH(P, mid, j)   ▷ + je zřetezení seznamů
31:     end if
32: end procedure
```

[1] Pozorný čtenář si jistě všimne, že technicky se v průběhu algoritmu může jednat o tah. Pokud je ohodnocovací funkce rozumná (viz dále) vrací algoritmus nakonec cestu.

[2] Podinstance, která odpovídá cestě $I(i, j, k)$ je podgraf obsahující pouze uzly i, j, a (případně) některé z uzlů $1, \ldots, k$ spolu s příslušnými hranami (které maji oba uzly v podgrafu) z původního grafu.

Algoritmus Floyd-Warshall pro vstupní graf s n uzly vrátí matici D o velikosti $n \times n$, kde je na pozici (i, j) cena nejkratší cesty mezi uzly i a j. Dále vrací matici P o rozměrech $n \times n$, která na pozici (i, j) obsahuje uzel, přes který nejkratší cesta z i do j vede. Tuto matici poté využívá procedura GetPath, která s její pomocí cestu sestaví. Lze snadno vidět, že algoritmus má kubickou složitost (3 vnořené cykly na řádcích 11 až 21).

Algoritmus nefunguje pro grafy, které obsahují tzv. záporné cykly. Záporný cyklus je takový cyklus, jehož suma hran je menší než 0. Opakováním tohoto cyklu můžeme získat potenciálně nekonečný tah. Záporný cyklus lze detekovat tak, že zkontrolujeme diagonálu matice D. Pokud se na ní nachází záporné číslo, leží uzel, který danému místu v matici odpovídá, na záporném cyklu.

Reference

[1] Donald Knuth. The Art of Computer Programming, Volume I. Addison-Wesley, 1997

[2] Donald Knuth. Selected papers on analysis of algorithms. Center for the study of language an information, 2000

[3] Cormen et. al. Introduction to algorithms. The MIT press. 2008.

[4] Donald Knuth Concrete mathematics: A foundation for computer science. Addison-Wesley professional, 1994

[5] John Kleinberg, Éva Tardes. Algorithm design. Addison-Wesley, 2005.

[6] U. Vazirani et. al. Algorithms. McGraw-Hill, 2006.

[7] Steve Skiena. The algorithm design manual. Springer, 2008.

[8] Juraj Hromkovič. Algorithmics for hard problems. Springer, 2010.

[9] Oded Goldreich. Computational complexity: a conceptual perspective. Cambridge university press, 2008.