Algoritmická matematika 3

Dynamické programování

Petr Osička

DATA ANALYSIS AND MODELING LAB
Univerzita Palackého v Olomouci

Zimní semestr 2013

Základní idea

Princip lze odvodit přechodem od rozděl a panuj

- Omezíme opakovaný výpočet pro stejné podinstance
- Podinstance, které se v průběhu výpočtu vyskytnou, vhodně uspořádáme
- Podinstance řešíme od nejmenší
- Ze vstupní instance jsme schopni najít nejmenší podinstance

Jak poznat, že dp je vhodnější než rp?

- opakující se podinstance u rp (Fibonnaciho číslo vs Quicksort)
- to, že se řešení podinstancí se překrývají (obsahují společnou část) je znamením toho, že dp je vhodnější.

Příklad

Princip demonstujeme na problému výpočtu n-tého Fibonacciho čísla.

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

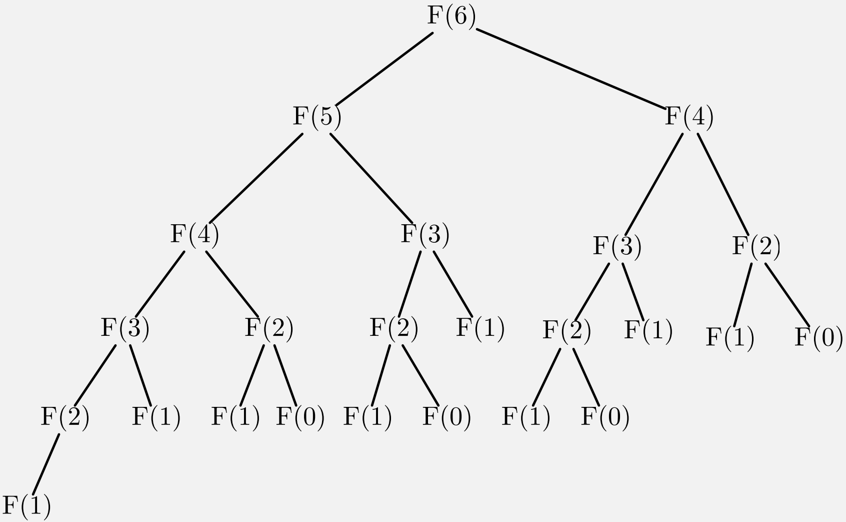
Princip rozděl a panuj vede na nasledujici algoritmus

```
procedure FibDQ(n)
    if n = 0 or n = 1 then
        return n
    end if
    return FibDQ(n − 1) + FibDQ(n − 2)
end procedure
```

Složitost je dána rekurencí: $T(n) = T(n-1) + T(n-2) + O(n)$ odhadem jejíhož řešení je $O(2^n)$.

Příklad (pokračování)

Strom rekurze:



Příklad (pokracovani)

Efektivitu můžem zvýšit zapamatováním si výsledků pro stejné podinstance

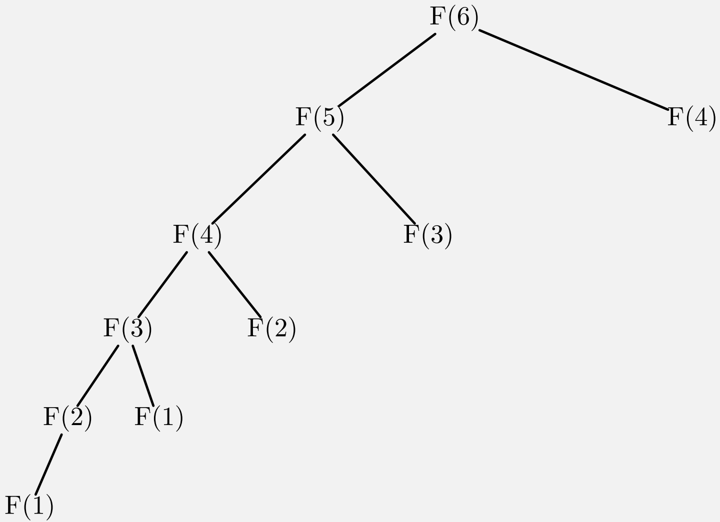
```
procedure PrepareTable(n)
    t[0] ← 0
    t[1] ← 1
    for i ← 2 to n do
        t[i] ← −1
    end for
    return t
end procedure
```

```
procedure FibHelp(n, t)
    if t[n] = −1 then
        t[n] ← FibHelp(n−1, t) + FibHelp(n−2, t)
    end if
    return t[n]
end procedure

procedure FibTab(n)
    return FibHelp(n, PrepareTable(n))
end procedure
```

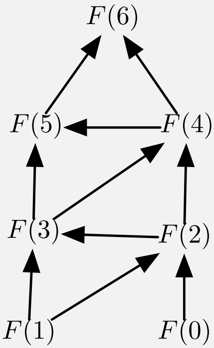
Složitost FibTab je lineární. Pamětová složitost je také lineární.

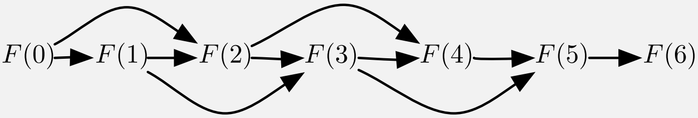
Příklad (pokračování)



Příklad

Všimněme si uspořádání podinstancí





Příklad

Můžeme počítat řešení podinstancí od nejmenší

```
procedure FibIdeal(n)
    if n = 0 then
        return n
    end if
    a ← 0
    b ← 1
    for n ← 2 to n − 1 do
        c ← a + b
        a ← b
        b ← c
    end for
    return b
end procedure
```

Časová složitost je stále lineární, paměťová složitost je konstantní

Dynamické programování

Tipy pro návrh

1. Je výhodné, pokud má vstupní instance **vhodné vnitřní uspořádání**, na jejímž základě lze generovat podinstance. (prefixy, intervaly, podgrafy …)
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. 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.
3. **Počet podinstancí** je klíčem k určení složítosti algoritmu. S pouze pár výjimkami platí, že čím méně podinstancí tím lépe.

Úloha batohu

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

Příklad

Pro instanci $I = (b, w_1, \dots, w_5)$, 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.

Úloha batohu

Uspořádání instance a tvar podinstancí

- Uspořádanou n-tici $w_1, \dots, w_n$ můžeme považovat za vnitřní lineární uspořádání instance I.
- Podinstance budeme generovat jako prefixy. Označme $I(i, c)$ podinstanci problému, ve které uvažujeme pouze prvních i členů $w_1, \dots, w_n$ spolu s kapacitou c

Vztah mezi podinstancemi

Pro $I(i, c)$ máme dvě instance, které jsou „o 1 menší“

- pokud i 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 $1, \dots, i-1$. Přirozeně tedy platí, že $w_i \leq c$, jinak by i nemohlo patřit do řešení.
- pokud i nepatří do řešení instance $I(i, c)$, pak má podinstance tvar $I(i-1, c)$.

Úloha batohu

Uspořádání podinstancí

Podinstance lze uspořádat bez cyklů. Platí totiž, že

$$I(i-1, c-w_i) \lesssim I(i,c) \text{ a současně } I(i-1,c) \lesssim I(i,c)$$

Minimální prvky

- $I(0,d)$ pro nějakou kapacitu d, což odpovídá práznému 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)

Ceny řešení

- $cost(I(0,d)) = 0$ a $cost(I(i,0)) = 0$
- pokud $w_i \geq c$, tak $cost(I(i,c)) = \max(cost(I(i-1,c)), cost(I(i-1,c-w_i)) + w_i)$
- jinak $cost(I(i,c)) = cost(I(i-1,c))$

procedure KnapsackDPTable($w_1, \ldots, w_n, b$)
    for $i \leftarrow 0$ **to** b **do**
        $M[0, i] \leftarrow 0$
    end for
    for $i \leftarrow 0$ **to** n **do**
        $M[i, 0] \leftarrow 0$
    end for
    for $i \leftarrow 1$ **to** n **do**
        for $c \leftarrow 1$ **to** b **do**
            if $c < w_i$ **then**
                $M[i, c] \leftarrow M[i-1, c]$
            else
                $M[i, c] \leftarrow \max(M[i-1, c], M[i-1, c-w_i] + w_i)$
            end if
        end for
    end for
    return M
end procedure

Úloha batohu

Jak nalézt v M řešení?

- „zpětným inženýrstvím“ výpočtu ceny řešení
- 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

$$x = cost(I(n-1, b - w_n)) \text{ a } y = cost(I(n-1, b)).$$

  - pokud $x + w_n > y$, pak w_n do řešení patří
  - pokud $x + w_n < y$ pak w_n do řešení nepatří
  - pokud $x = y$ můžeme zvolit jak chceme.
  - Posuneme se na odpovídající podinstanci (podle toho, jestli jsme vybrali x nebo y.
- Pokračujeme, dokud se nedostame na nejmenší podinstanci.

Úloha batohu

procedure KnapsackDPSolution($w_1, \ldots, w_n, b, M$)
    $S \leftarrow \emptyset$
    $c \leftarrow b$
    for $i \leftarrow n$ **to** 0 **do**
        if $c - w_i \geq 0$ **and** $M[i-1, c] < M[i-1, c-w_i] + w_i$ **then**
            $S \leftarrow S \cup \{i\}$
            $c \leftarrow c - w_i$
        end if
    end for
    return S
end procedure

Úloha batohu

Složitost:

- vytváříme tabulku o velikosti $n \cdot b$.
- pro vyplnění jednoho políčka je potřeba konstantní počet operací
- Složitost nalezení řešení je lineární.
- Složitost je $\Theta(n \cdot b)$.

Poznámka:

- Složitost algoritmu 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.
- Pokud bychom například vynásobili všechna čísla v instanci, kterou jsme si uvedli jako příklad, milionem, algoritmus by počítal řešení milionkrát déle.

Nejkratší cesty v grafu

Definice

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

Cesta z uzlu s do uzlu t v grafu G je posloupnost vrcholů a hran $P = u_0, e_0, \dots, e_{n-1}, u_n$ 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.

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

Definice

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 $dist(s, t)$.

Nejkratší cesty v grafu

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

- existuje řada algoritmů (Dijkstrův, ...)
- ukážeme si **Floyd-Warshallův algoritmus**

Nejkratší cesty v grafu

Označme si vrcholy grafu jako $V = \{1, 2, \ldots, n\}$.

Označme jako $I(i, j, k)$ nejkratší cestu mezi uzly i a j, která mimo je obsahuje pouze uzly menší než k. Označme si cenu této cesty jako $dist(i, j, k)$.

Poznámky

- $I(i, j, k)$ může být prázdná (tj. neexistuje), pak je její cena rovna ∞.
- i a j nemusí být různé, pak se délka cesty rovná 0.
- $I(i, j, k)$ se v průběhu algoritmu může změnit na o tah. Pokud je ohodnocovací funkce rozumná (bez záporných cyklů) vrací algoritmus nakonec cestu.

Jak dostaneme cestu $I(i,j,k+1)$ z $I(i,j,k)$?

- 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

$$dist(i,j,k+1) = dist(i,k+1,k) + dist(k+1,j,k) < dist(i,j,k).$$

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

Podinstance

- Pro každé k počítáme $I(i,j,k)$ pro všechny dvojice uzlů i,j. Odpovídající podinstance je podgraf obsahující pouze uzly i, j, a 1, ..., k spolu s příslušnými hranami. Lze je 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 $dist(i,j,0) = \infty$.

Floyd-Warshallův algoritmus

procedure FWPath($G = (V, E), c$)
    for $i \leftarrow 1$ **to** n **do:**
        for $j \leftarrow 1$ **to** n **do:**
            $D[i, j] = \infty$
            $P[i, j] = \infty$
        end for
    end for
    for $\{i, j\} \in E$ **do:**
        $D[i, j] = c(\{i, j\})$
    end for
    for $k \leftarrow 1$ **to** n **do:**
        for $i \leftarrow 1$ **to** n **do:**
            for $j \leftarrow 1$ **to** n **do:**
                $x \leftarrow D[i, k] + D[k, j]$
                if $x < D[i, j]$ **then**
                    $D[i, j] \leftarrow x$
                    $P[i, j] \leftarrow k$
                end if
            end for
        end for
    end for
    return $\langle D, P \rangle$
end procedure

Floyd-Warshallův algoritmus

```
procedure GetPath(P,i,j)
    mid ← P[i, j]
    if mid = ∞ then
        return []
    else
        return GetPath(P, i, mid) · [mid]· GetPath(P, mid, j)
    end if
end procedure
```

Nejkratší cesty v grafu

Algoritmus FWPath **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.
- 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.

Poznámky

- kubická složitost
- 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.