

# 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 vizualizovat 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

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

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 vizualizujeme 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.

**Algoritmus 2**  $n$ -té Fibonacciho číslo s pamatováním si mezivýsledků

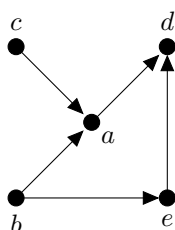
---

```

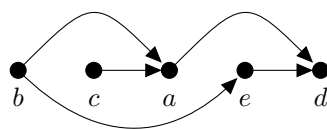
1: procedure PREPARETABLE( $n$ )
2:    $t[0] \leftarrow 0$ 
3:    $t[1] \leftarrow 1$ 
4:   for  $i \leftarrow 2$  to  $n$  do
5:      $t[i] \leftarrow -1$ 
6:   end for
7:   return  $t$ 
8: end procedure
9:
10: procedure FIBHELP( $n, t$ )
11:   if  $t[n] = -1$  then
12:      $t[n] \leftarrow \text{FIBTAB}(n-1) + \text{FIBTAB}(n-2)$ 
13:   end if
14:   return  $t[n]$ 
15: end procedure
16:
17: procedure FIBTAB( $n$ )
18:   return FIBHELP( $n, \text{PREPARETABLE}(n)$ )
19: end procedure

```

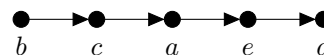
---



(a) Orientovaný graf bez cyklů



(b) Lineární uspořádání vrcholů



(c) Odpovídající lineární tvar

Obrázek 1: Linearizovaná podoba grafu

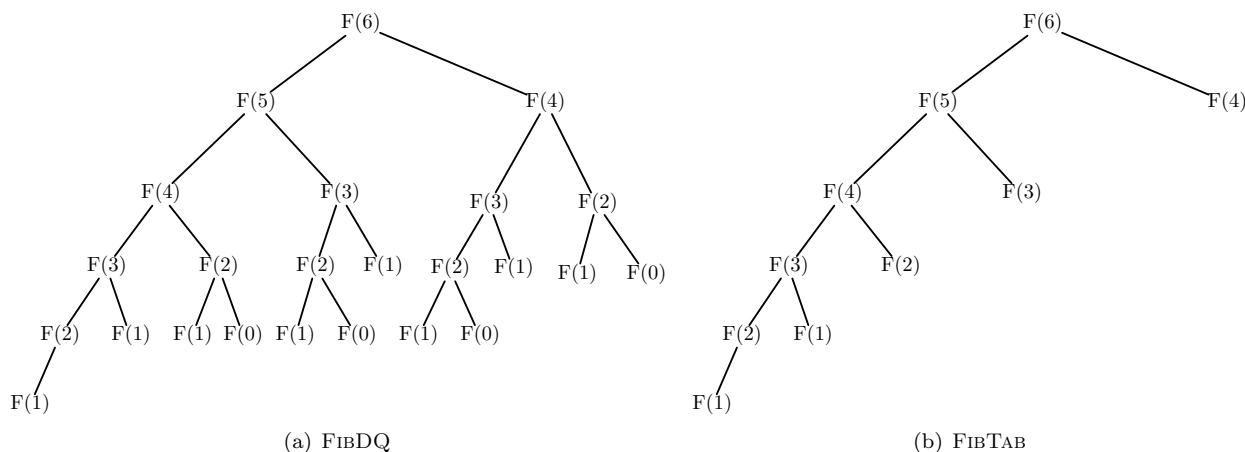
Složitost FIBTAB je lineární. Lze snadno vidět, že k rekurzivnímu volání na řádce 12 dojde pro každé číslo (a tedy pro každou podinstanci) maximálně jedenkrát. Ve zbylých případech je vrácena v tabulce zapamatovaná hodnota. Strom rekurzivních volání pak vypadá jako na obrázku 2 (b).

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 ???. 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,



Obrázek 2: Rekurzivní strom volání výpočtu Fibonacciho 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:    $a \leftarrow 0$ 
6:    $b \leftarrow 1$ 
7:   for  $n \leftarrow 2$  to  $n - 1$  do
8:      $c \leftarrow a + b$ 
9:      $a \leftarrow b$ 
10:     $b \leftarrow c$ 
11:  end for
12: end procedure

```

---

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.

- 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. Formálně je úloha batohu určena následovně:

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

Čísla  $w_i$  odpovídají objemům jednotlivých pytlů s pískem,  $b$  je kapacita batohu. 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.

Uvažujme vstupní instanci  $I = (b, w_1, \dots, w_n)$ . Uspořádanou  $n$ -tici  $w_1, \dots, w_n$  můžeme považovat za vnitřní lineární uspořádání instance  $I$ . Jednotlivé podinstance, kterými při výpočtu projdeme, budeme brát jako prefixy této posloupnosti doplněné o kapacitu. Označme si tedy jako  $I(i, c)$  takovou podinstanci instance  $I$ , ve které uvažujeme pouze prvních  $i$  členů  $w_1, \dots, w_n$  spolu s kapacitou  $c$  (kde  $c \leq b$ ). Dalším krokem je uvědomit si, jaké existuje uspořádání podinstancemi tohoto tvaru. Musíme se zamyslet nad tím, optimální řešení kterých podinstancí nám nejvíce pomůže při výpočtu optimálního řešení pro  $I(i, c)$ . Přirozeným kandidátem jsou podinstance odpovídající prefixu o 1 kratšímu, tedy  $I(i-1, \_)$ , které se liší pouze o  $i$ -tou položku. Potom

- pokud by  $i$  patřilo do optimálního řešení instance  $I(i, c)$ , pak toto řešení můžeme dostat pouhým přidáním  $i$  k optimálnímu řešení instance  $I(i-1, c-w_i)$ . Protože uvažujeme přidání  $i$  do řešení, musíme přirozeně snížit kapacitu na  $c-w_i$ , aby se položka  $i$  do batohu vešla.
- pokud by  $i$  nepatřilo do optimálního řešení instance  $I(i, c)$ , pak je toto řešení stejné jako optimální řešení podinstance  $I(i-1, c)$ .

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 (tedy instanci, kde nemáme žádné položky pro vkládání do batohu)
- $I(i, 0)$  pro nějaké  $i$ , což odpovídá prázdné kapacitě (a nemá cenu řešit, které prvky z  $w_1, \dots, w_{i-1}$  dám do řešení, protože kapacita je prázdná)

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

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

kde ve výraz  $\text{cost}(I(i-1, c-w_i)) + w_i$  odpovídá situaci, kdy  $i$  patří do optimálního řešení (a musíme tedy k ceně řešení přičíst jeho váhu) a výraz  $\text{cost}(I(i-1, c))$  odpovídá situaci, kdy  $i$  do řešení nepatří. Dále pro minimální prvky platí, že  $\text{cost}(I(0, d)) = 0$  (nemám prvky, které bych do řešení zařadil) a  $\text{cost}(I(i, 0)) = 0$  (prázdná kapacita a nemůžu do řešení zařadit žádné prvky).

Z předchozího lze vidět, že nám v podstatě stačí počítat jenom ceny řešení podinstancí, a pokud si zapamatujeme, který z výrazů v (2) byl větší, můžeme toto řešení posléze najít. Nyní si už můžeme napsat algoritmus, s pomocí kterého spočítáme cenu řešení  $I(n, b)$ . Algoritmus si vytváří tabulku, jejíž řádky odpovídají jednotlivým položkám, které chceme vložit do batohu, uvažujeme i jeden řádek odpovídající žádné položce. Sloupce tabulky odpovídají číslům  $0, 1, \dots, b$ , tedy všem číslům potenciálně se vyskytujícím jako kapacita některé podinstance. Na místo v tabulce na řádku odpovídajícím položce  $i$  a číslu  $c$  spočítáme cenu optimálního řešení podinstance  $I(i, c)$ . To můžeme tak, že první řádek a první sloupec vyplníme nulami (viz minimální podinstance a jejich ceny diskutované v předchozím odstavci). Poté můžeme po řádcích doplňovat další místa tabulky podle vztahu (2). Nakonec na místě odpovídajícím  $I(n, b)$  je cena řešení vstupní instance  $I(n, b)$ .

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  $x = \text{cost}(I(n-1, b-w_n))$  a  $y = \text{cost}(I(n-1, b))$ . 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 4.

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  $w_1, \dots, w_n$ , 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( $w_1, \dots, w_n$ )
2:   for  $i \leftarrow 0$  to  $b$  do
3:      $M[0, i] \leftarrow 0$ 
4:   end for
5:   for  $i \leftarrow 0$  to  $n$  do
6:      $M[i, 0] \leftarrow 0$ 
7:   end for
8:   for  $i \leftarrow 1$  to  $n$  do
9:     for  $c \leftarrow 1$  to  $b$  do
10:      if  $c < w_i$  then
11:         $M[i, c] \leftarrow M[i - 1, c]$ 
12:      else
13:         $M[i, c] \leftarrow \max(M[i - 1, c], M[i - 1, c - w_i] + w_i)$ 
14:      end if
15:    end for
16:  end for
17:   $S \leftarrow \emptyset$ 
18:   $c \leftarrow b$ 
19:  for  $i \leftarrow n$  to  $0$  do
20:    if  $c - w_i \geq 0$  and  $M[i - 1, c] < M[i - 1, c - w_i] + w_i$  then
21:       $S \leftarrow S \cup \{i\}$ 
22:       $c \leftarrow c - w_i$ 
23:    end if
24:  end for
25:  return  $S$ 
26: end procedure

```

---

## 2.2 NEJKRATŠÍ CESTY V GRAFU

**Definice 1.** Necht  $G = (V, E)$  je graf. *Ohodnocení hran* grafu  $G$  je zobrazení  $c : E \rightarrow \mathbb{Q}$  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  $P = u_0, e_0, \dots, e_{n-1}, u_n$  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.

**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:	$sol(G, s, t) = \{P \mid P \text{ je cesta z uzlu } s \text{ do uzlu } t\}$
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  $V = \{1, 2, \dots, n\}$  a označme si jako  $I(i, j, k)$  nejkratší cestu<sup>1</sup> 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  $\infty$ . 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

$$dist(i, j, k + 1) = dist(i, k + 1, k) + dist(k + 1, j, k) < dist(i, j, k). \quad (3)$$

- 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<sup>2</sup> lze uspořádat podle třetí komponenty. Minimální prvky jsou pak podinstance odpovídající  $I(i, j, 0)$ , což je buď cesta tvořená 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 4.

---

**Algoritmus 5** Nejkratší cesty v grafu

---

```

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

```

---

<sup>1</sup>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.

<sup>2</sup>Podinstance, která odpovídá cestě  $I(i, j, k)$  je podgraf obsahující pouze uzly  $i, j$ , a (případně) některé z uzlů  $1, \dots, k$  spolu s příslušnými hranami (které mají 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 and 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 GOLDBREICH. Computational complexity: a conceptual perspective. Cambridge university press, 2008.