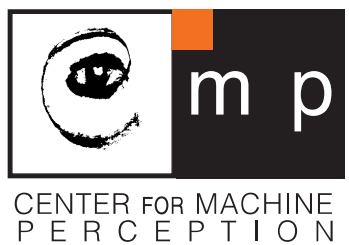
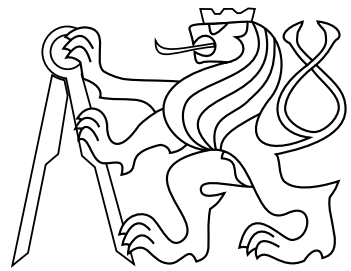
Dynamické programování

Jan Kybic

http://cmp.felk.cvut.cz/~kybic
kybic@fel.cvut.cz

2016





Memoizace

Dynamické programování

Memoizace
(Memoization/caching)

- Pro dlouhotrvající funkce $f(x)$
- Jednou vypočtené $f(x)$ si zapamatujeme
- Vyměníme prostor za čas *(time-space trade-off)*
- Funkce musí být čistá *(pure)*

Memoizace — implementace

Pro funkce jednoho argumentu. Vrátí funkci, která se chová jako f, jen jednou vypočítané hodnoty nejsou počítány znovu.

```
def memoize(f):
  memo={}
  def g(x):
    if x not in memo:
      memo[x]=r=f(x)
      return r
    return memo[x]
  return g
```

Memoizace — použití

```
def is_prime(p):
  if p<2:
    return False
  n=2
  while n*n<=p:
    if p % n == 0:
      return False
    n+=1
  return True

g=memoize(is_prime)
```

Soubor `memoize.py`.

Memoizace se zapomínáním

Last recently used (LRU)

- Nechceme si pamatovat vše
- Zapomeneme hodnotu použitou před nejdelší dobou *(LRU — last recently used)*

Memoizace se zapomínáním

Last recently used (LRU)

- Nechceme si pamatovat vše
- Zapomeneme hodnotu použitou před nejdelší dobou *(LRU — last recently used)*

V Pythonu (varianta bez zapomínání)

```
h=functools.lru_cache(maxsize=None)(is_prime)
```

Memoizace — měření času

Otestuje čísla $1, 2, \ldots 10^5$ na prvočíselnost.

```
>>> memoize.try_memoize_primes()
Bez memoizace:        1.41s
S memoizací poprvé:   1.51s
S memoizací podruhé:  0.06s
S lru_cache poprvé:   1.75s
S lru_cache podruhé:  0.23s
```

Soubor `memoize.py`.

Fibonacciho čísla

$$F_N = \begin{cases} 1 & \text{if } N < 2 \\ F_{N-1} + F_{N-2} & \text{if } N \geq 2 \end{cases}$$

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$

Rekurzivní implementace:

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

Fibonacciho čísla

$$F_N = \begin{cases} 1 & \text{if } N < 2 \\ F_{N-1} + F_{N-2} & \text{if } N \geq 2 \end{cases}$$

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$

Rekurzivní implementace:

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

Časová složitost je $\propto F_N \propto \phi^N$, kde $\phi \approx 1.618$.

Fibonacciho čísla

$$F_N = \begin{cases} 1 & \text{if } N < 2 \\ F_{N-1} + F_{N-2} & \text{if } N \geq 2 \end{cases}$$

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$

Rekurzivní implementace:

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

Časová složitost je $\propto F_N \propto \phi^N$, kde $\phi \approx 1.618$.

Memoizace odstraní rekurzivní volání.

Soubor `memoize.py`.

Fibonacciho čísla a memoizace

```
def fibonacci2(n):
  if n <= 2:
    return 1
  else:
    return fibonacci3(n-1) + fibonacci3(n-2)

fibonacci3=memoize(fibonacci2)
```

Fibonacciho čísla a memoizace

```
def fibonacci2(n):
  if n <= 2:
    return 1
  else:
    return fibonacci3(n-1) + fibonacci3(n-2)

fibonacci3=memoize(fibonacci2)
```

`fibonacci=memoize(fibonacci)` funguje také.

Fibonacciho čísla a memoizace

```
def fibonacci2(n):
  if n <= 2:
    return 1
  else:
    return fibonacci3(n-1) + fibonacci3(n-2)

fibonacci3=memoize(fibonacci2)
```

`fibonacci=memoize(fibonacci)` funguje také.

Python má tzv. *dekorátory (decorators)*, např. `@memoize`.

Fibonacciho čísla iterativně

specializovaná memoizace — pro F_N budeme potřebovat $F_1, F_2, \ldots, F_{N-1}$

```
def fibonacci4(n):
  p=[ 1 for i in range(n+1) ]
  for i in range(3,n+1):
    p[i]=p[i-1]+p[i-2]
  return p[n]
```

Fibonacciho čísla iterativně

specializovaná memoizace — pro F_N budeme potřebovat $F_1, F_2, \ldots, F_{N-1}$

```
def fibonacci4(n):
  p=[ 1 for i in range(n+1) ]
  for i in range(3,n+1):
    p[i]=p[i-1]+p[i-2]
  return p[n]
```

Stačí si pamatovat dvě poslední hodnoty.

Soubor `memoize.py`.

Experimentální vyhodnocení časové složitosti

	N	25	30	40	400	4000	40000
`fibonacci`	rekurzivní	0.06	0.62	75.44	*S*	*S*	*S*
`fibonacci3`	memoizace	0.00	0.00	0.00	0.00	*R*	*R*
`fibonacci4`	iterativní	0.00	0.00	0.00	0.00	0.00	0.16

Čas v sekundách.
S — neskončí v rozumném čase
R — překročená maximální hloubka rekurze

Memoizace vyrobila z rekurzivního algoritmu nerekurzivní.

Memoizace

Dynamické programování

Dynamické programování
Dynamic programming

- Problém rozdělíme na několik menších.
- Menší problémy vyřešíme systematicky ‘všechny’ a výsledky uložíme.
- Explicitní memoizace.
- Pro řešení většího problému využíváme dříve předpočítaných řešení.

Dynamické programování
Dynamic programming

- Problém rozdělíme na několik menších.
- Menší problémy vyřešíme systematicky ‘všechny’ a výsledky uložíme.
- Explicitní memoizace.
- Pro řešení většího problému využíváme dříve předpočítaných řešení.

Příklad: Fibonacciho posloupnost `fibonacci4`

Dynamické programování
Dynamic programming

- Problém rozdělíme na několik menších.
- Menší problémy vyřešíme systematicky ‘všechny’ a výsledky uložíme.
- Explicitní memoizace.
- Pro řešení většího problému využíváme dříve předpočítaných řešení.

Příklad: Fibonacciho posloupnost `fibonacci4`

‘programování’ = reformulace problému

Bellmanův princip optimality

- Ve stavovém prostoru hledejme cestu mezi stavy s a t.
- Nechť stav r leží na *optimální* cestě:
  $p_1 = s, \ldots, p_M = r, \ldots, p_N = t$.
- Pak i podcesty $p_1 = s, \ldots, p_M = r$ a $p_M = r, \ldots, p_N = t$ jsou optimální.

Dynamické programování a stavový prostor

- Nechť kritérium optimality je součet cen akcí a/nebo stavů, které minimalizujeme.

$$C = \sum_{i=1}^{N} C(p_i) + \sum_{i=1}^{N-1} C(a_i), \quad \text{kde} \quad p_{i+1} = f(p_i, a_i)$$

- Nechť délka řešení je maximálně N kroků.
- Nechť v každém kroku existuje množina $\mathcal{R}_i$ stavů, kudy musí optimální cesta procházet.
- Nechť se do stavů $\mathcal{R}_{i+1}$ z $\mathcal{R}_i$ mohu dostat maximálně k způsoby.

Dynamické programování a stavový prostor (2)

Problém lze řešit dynamickým programováním takto:

- V každém kroku i si pamatuji, jaká je nejlepší cena $D_i(r)$ dosažení každého stavu r z $\mathcal{R}_i$, optimálního předchůdce a akci.
- Při výpočtu ceny stavů $u \in \mathcal{R}_{i+1}$ v dalším kroku vyberu nejlepší z možných akcí

$$D_{i+1}(u) = \min_{a \in A} D_i(r) + C(a) + C(u) \quad \text{kde} \quad u = f(r, a)$$

- Po dosažení cílového stavu zpětně rekonstruuji optimální cestu (je-li nutno).

Složitost tohoto algoritmu je $O(kN)$.

Dynamické programování a stavový prostor (2)

Problém lze řešit dynamickým programováním takto:

- V každém kroku i si pamatuji, jaká je nejlepší cena $D_i(r)$ dosažení každého stavu r z $\mathcal{R}_i$, optimálního předchůdce a akci.
- Při výpočtu ceny stavů $u \in \mathcal{R}_{i+1}$ v dalším kroku vyberu nejlepší z možných akcí

$$D_{i+1}(u) = \min_{a \in A} D_i(r) + C(a) + C(u) \quad \text{kde} \quad u = f(r, a)$$

- Po dosažení cílového stavu zpětně rekonstruuji optimální cestu (je-li nutno).

Složitost tohoto algoritmu je $O(kN)$.

Existuje mnoho variant tohoto základního postupu.

Příklad: Mince

- Řešili jsme, jak vypsat *všechny* způsoby, jak zaplatit x Kč mincemi v hodnotách $h = \{50, 20, 10, 5, 2, 1\}$ Kč.
- Nyní najděme *nejmenší* počet mincí, kterými lze zaplatit x Kč.

Příklad: Mince

- Řešili jsme, jak vypsat *všechny* způsoby, jak zaplatit x Kč mincemi v hodnotách $h = \{50, 20, 10, 5, 2, 1\}$ Kč.
- Nyní najděme *nejmenší* počet mincí, kterými lze zaplatit x Kč.
- Hladový *(greedy)* algoritmus nefunguje pro všechny h.

Mince — rekurzivně

Jak zaplatit částku x pomocí co nejmenšího počtu mincí o hodnotách h (pole). Vrátí pole počtů m_i mincí h_i.
Myšlenka: Rekurzivně zkus zaplatit $x - h_i$ a vyber nejlepší možnost.

Mince — rekurzivně

```
def zaplat_nejlepe(h,x):
  def zaplat_internal(x):
    if x in h:      # existuje mince s hodnotou 'x'?
      return [ int(x==v) for v in h ]
    else:
      nejlepsi=None              # nejlepší pole počtů 'm'
      for i in range(len(h)):    # pro všechny mince
        if h[i]<=x:              # které lze použít
          m=zaplat_internal(x-h[i])
          m[i]+=1
          if nejlepsi is None or sum(m)<sum(nejlepsi):
            nejlepsi=m.copy()
      return nejlepsi
  return zaplat_internal(x)
```

Funguje, ale má exponenciální složitost, použitelné jen asi do $x = 35$ (40 s).

Soubor `dynamic.py`.

Mince — memoizace

```
def zaplat_nejlepe2(h,x):
  def zaplat_internal(x):  # existuje mince s hodnotou 'x'?
    if x in h:
      return [ int(x==v) for v in h ]
    else:
      nejlepsi=None
      for i in range(len(h)):
        if h[i]<=x:
          m=zaplat_internal(x-h[i])
          m[i]+=1
          if nejlepsi is None or sum(m)<sum(nejlepsi):
            nejlepsi=m.copy()
      return nejlepsi
  zaplat_internal=memoize.memoize(zaplat_internal)
  return zaplat_internal(x)
```

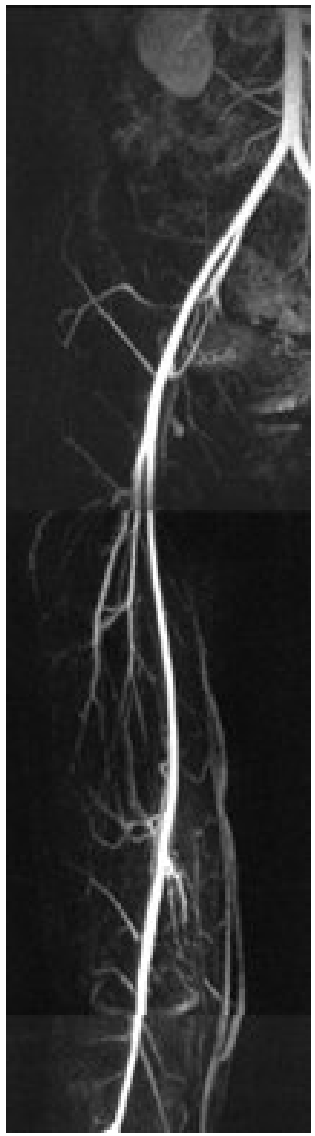
Funguje mnohem rychleji, použitelné asi do $x = 24500$ (0.42 s), pak dojde k překročení hloubky zásobníku.

Mince — dynamické programování

```
def zaplat_nejlepe3(h,x):
  reseni=[ [] ] * (x+1)       # nej. řešení pro zapl. 'x'
  reseni[0]=[0] * len(h)
  for y in range(1,x+1): # jak zaplatit y=1..x Kč
    nejlepsi_i=None       # kterou minci je nejlépe použít
    nejlepsi_pocet=None   # kolik mincí bude pak potřeba
    for i in range(len(h)):    # zkus všechny mince
      if h[i]<=y:              # lze ji použít?
        pocet=sum(reseni[y-h[i]])+1 # tolik mincí je potřeba
        if nejlepsi_i is None or pocet<nejlepsi_pocet:
          nejlepsi_i=i
          nejlepsi_pocet=pocet
    if nejlepsi_i is None:     # pro jistotu
      return None
    reseni[y]=reseni[y-h[nejlepsi_i]].copy()
    reseni[y][nejlepsi_i]+=1
  return reseni[x]
```

Nejrychlejší řešení, složitost zhruba lineární, $O(x)$ (0.02 s pro $x = 2450$, 0.24 s pro $x = 24500$, 2.7 s pro $x = 245000$)

Příklad: hledání lineárních struktur v obraze



- Rentgenový (CT) obraz dolních končetin.
- Najděte nejjasnější spojnici první a poslední řádky obrazu.
- Maximalizujte:

$$J = \sum_{i=0}^{n_y} f(i, p_i)$$

$$\text{aby} \quad |p_i - p_{i-1}| <= 1$$

- $f(y, x)$ = jas pixelu na souřadnicích (y, x)

Knihovna numpy

Numeric Python

```
>>> import numpy as np
>>> a=np.zeros((3,4),dtype=np.uint8)
>>> a[1,2]=3
>>> a
array([[0, 0, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 0]], dtype=uint8)
```

Typ `numpy.array`

- Vícerozměrná pole, vícerozměrné indexování.
- Definovatelný a homogenní typ elementů.
- Efektivnější využití paměti než nativní typ `list`.
- Interoperabilita s jinými programovacími jazyky.
- Nepodporuje dynamickou změnu rozměrů, vyhledávání atp.
- Podporuje řadu matematických vektorových a maticových operací (à la Matlab)

Načtení a zobrazení obrázku

```
import scipy                # Scientific Python
import scipy.misc           # pro čtení obrázků
import pylab                # pro zobrazování a grafy
import numpy as np          # numeric arrays

def test_vessels():
  # načti obrázek do numpy.array pole
  f=scipy.misc.imread('figs/limb_vessels2.jpg')
  pylab.figure(1)                          # první okno na obrazovce
  pylab.imshow(f,cmap=pylab.cm.gray) # ukaž šedotónový obrázek
  p=find_vertical(f)                       # najdi spojnici, vrať x souřad.
  pylab.figure(2)                          # druhé okno na obrazovce
  pylab.imshow(f,cmap=pylab.cm.gray) # ukaž obrázek znovu
  pylab.plot(p,range(len(p)),'r.')   # nakresli spojnici červeně
  pylab.show()                             # počkej, až uživatel okna zavře
```

Možností načítání a zobrazování obrázků je více.

Soubor `vessels.py`.

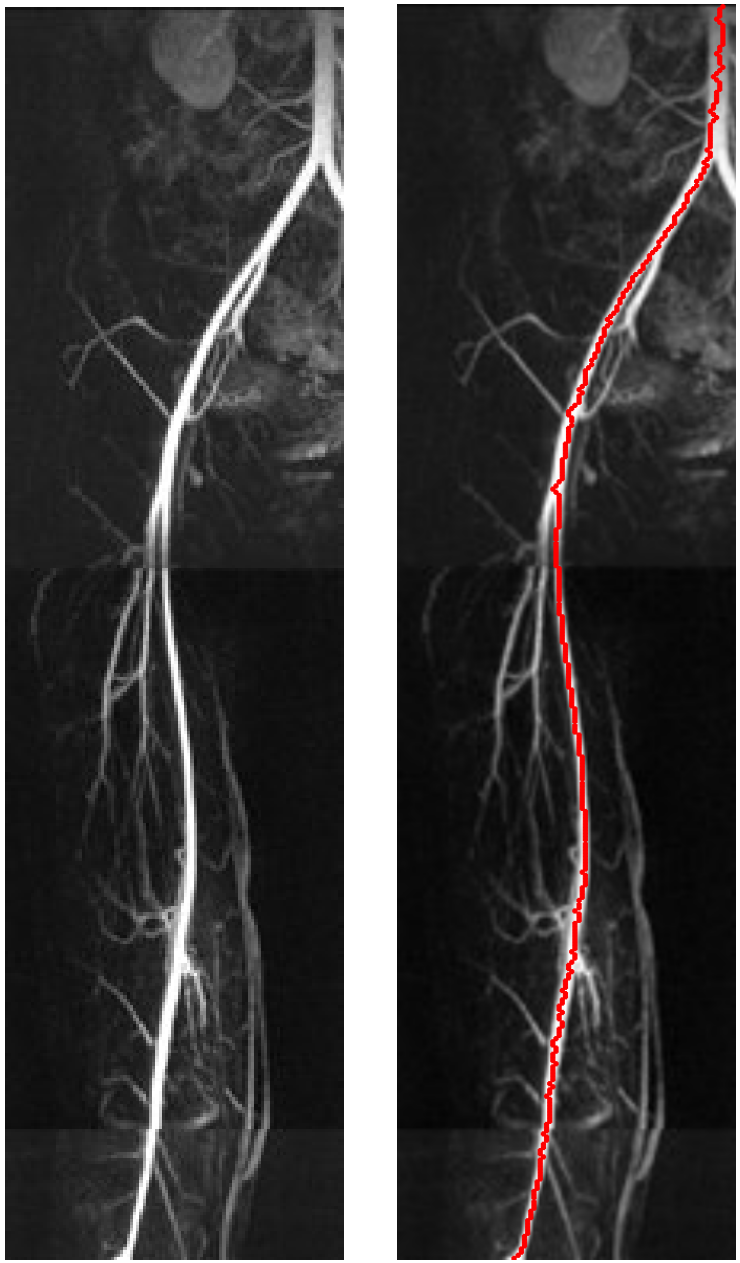
Hledání spojnice a dynamické programování

```
def find_vertical(f):
  ny,nx=f.shape          # velikost obrázku
  c=np.zeros(nx)         # ceny sloupců pro aktuální řadu
  cnew=np.zeros(nx)      # ceny sloupců pro novou řadu
  delta=np.zeros((ny,nx),dtype=np.int8) # posun p[i]-p[i-1]

  for iy in range(ny):       # první průchod shora dolů
    for ix in range(nx):     # pro všechny sloupce
      cnew[ix],delta[iy,ix]=max(  # nejlepší cesta do (iy,ix)
        (c[ix+d]+f[iy,ix],d) for d in (-1,0,1)
          if ix+d<nx and ix-d>=0 )
    cnew,c=c,cnew            # vyměň odkazy na pracovní pole

  p=np.zeros(ny,dtype=np.int)      # výsledná cesta
  p[ny-1]=np.argmax(c)             # začni v nejlepším sloupci
  for iy in range(ny-1,0,-1):      # zpětný průchod
    p[iy-1]=p[iy]+delta[iy,p[iy]]
  return p
```

Výsledek



Náměty na domácí práci

- Implementujte iterativní výpočet Fibonacciho čísel na základě dvou předchozích hodnot.
- Jak závisí zrychlení na velikosti LRU paměti v uvedených příkladech memoizace?
- Jaká je složitost algoritmu nalezení minimálního počtu mincí pro zaplacení dané částky vzhledem k počtu druhů mincí?
- Řešte úlohu minimálního počtu mincí pro zaplacení dané částky, pokud je počet mincí dané hodnoty omezen.
- Řešte úlohu batohu (dynamickým programováním): Je dáno N položek s váhou m_i a hodnotou c_i. Kolik kterých položek mám vybrat, aby váha nepřekročila M a hodnota byla maximální?
- Upravte řešení problému mincí dynamickým programováním, aby nebylo potřeba ukládat vždy celé řešení, ale jen naposledy přidanou minci.
- Upravte hledání jasné cesty v obrazu tak, aby cesta nemusela být pouze svislá.