Prohledávání stavového prostoru do hloubky

= depth-first search (DFS)
backtracking
prohledávání s návratem
zpětné prohledávání

- začínáme v zadaném výchozím stavu systému
- postupně zkoušíme všechny varianty pokračování, dokud
  nenajdeme řešení úlohy (hledáme-li jedno libovolné)
  nebo dokud
  neprojdeme všechny možnosti (máme-li nalézt všechna řešení)
- je-li zvolená cesta neúspěšná, vrátíme se z ní zpět a zkoušíme jinou
- to se opakuje v každém kroku, kde je více možných pokračování
  $\rightarrow$ průchod stromem všech možných cest výpočtu do hloubky,
  v každém stavu procházíme všechna možná pokračování
  (procházíme seznam možných cest)

Stejný postup jsme použili již dříve při procházení stromu do hloubky.

Na stejném principu jsou založeny také algoritmy rekurzivního generování.

Stejná je proto také **implementace algoritmu**:

- buď *rekurzivní procedura*, která zpracuje aktuální stav a volá se rekurzivně na stavy, do nichž z něj vede přechod

- nebo *zásobník* na uložení všech stavů, které jsme již navštívili, ale ještě jsme je nezpracovali (tzn. nezkoumali jsme přechody, které z nich vedou) + cyklus

    dokud nedojdeme do cílového stavu (hledáme-li jedno řešení)

nebo

    dokud se zásobník nevyprázdní (hledáme-li všechna řešení)

Příklad: **Proskákání šachovnice koněm**

- je dána výchozí pozice šachového koně
- úkol: proskákat s ním postupně všechna pole na šachovnici, žádné přitom nenavštívit dvakrát

- v každé pozici zkoušíme postupně provést koněm tah na všechna dosud nenavštívená sousední pole
- pro každý takový tah hledáme rekurzivně řešení v pozici vzniklé tímto tahem

Paměťová složitost

Výška stromu představujícího všechny možné cesty výpočtu:

kořen = výchozí situace

list stromu = buď řešení, nebo „slepá ulička“

Obvykle bývá rozumně velká, paměťově zvládnutelná.

Časová složitost

Počet uzlů ve stromu představujícím všechny možné cesty výpočtu (příp. v lepším případě jen jeho části, pokud neprocházíme celý, ale jen do nalezení prvního řešení).

Obvykle exponenciální, tedy metoda je použitelná jen pro velmi malé úlohy.

Snaha alespoň trochu zlepšit časovou složitost
→ *ořezávání, heuristiky*.

Urychlení prohledávání do hloubky

Ořezávání

Během procházení stavového stromu do hloubky vyhodnocujeme průběžně situaci v každém uzlu a je-li neperspektivní (tzn. nemůže vést k nalezení řešení), příslušný podstrom vůbec neprocházíme (odřízneme ho ze stromu).

Se stejným principem ořezávání jsme se již setkali při řešení některých úloh na rekurzivní generování.

U některých úloh může ořezávání ušetřit velké množství práce.

Příklad: **Osm dam na šachovnici**

- úkol: rozmístit na šachovnici 8 šachových dam tak,
  aby se žádné dvě navzájem neohrožovaly

- bez ořezávání (hloupý postup):
  zkoušet všechny výběry 8 polí z 64

- základní ořezávání:
  na každém řádku právě jedna dáma,
  po umístění všech osmi dam otestovat kolize

- lepší ořezávání:
  hned při umisťování každé dámy testovat kolize
  s dámami umístěnými na předchozích řádcích

```
n = 8
a = [[False]*n for _ in range(n)]

def vypis():
    for i in range(n):
        for j in range(n):
            if a[i][j]:
                print('*', end=' ')
            else:
                print('.', end=' ')
        print(end='\n')
    print(end='\n')
```

```
def kolize(r, s):
    """umístění dámy na pozici [r,s] způsobí kolizi"""
    for i in range(r):
        if a[i][s]:
            return True      # kolize svisle nahoru

    i = r-1; j = s-1
    while i >= 0 and j >= 0:
        if a[i][j]:
            return True      # kolize šikmo vlevo nahoru
        i-=1; j-=1

    i = r-1; j = s+1
    while i >= 0 and j < n:
        if a[i][j]:
            return True      # kolize šikmo vpravo nahoru
        i-=1; j+=1

    return False             # bez kolizí
```

```
def dama(r):
    """zkouší umístit dámu na řádek r"""
    for j in range(n):
        a[r][j] = True
        if not kolize(r, j):
            if r < n-1:
                dama(r+1)
            else:
                vypis()
        a[r][j] = False


dama(0)
```

Příklad: **Magické čtverce**

- úkol: nalézt všechny magické čtverce *N* x *N*
- každé z čísel od 1 do N^2 právě jednou
- stejné součty ve všech řádcích a sloupcích
  (příp. i na obou diagonálách)

pro *N* = 3:

8	1	6
3	5	7
4	9	2

Hloupé řešení: zkoušíme všechny permutace čísel od 1 do N^2

Ořezávání:
- vždy hned po vyplnění jednoho řádku otestovat,
  zda je perspektivní (tzn. zda má správný součet)
- když není, neztrácet zbytečně čas zkoušením všech permutací
  zbývajících čísel na zbývajících pozicích ve čtverci

Určení součtu čísel v každé řadě magického čtverce řádku N:
- součet všech čísel od 1 do N^2 je roven $N^2.(N^2 + 1) / 2$
- o tento součet musí rovným dílem podělit N řad,
  tedy každá z nich má součet $N.(N^2 + 1) / 2$

Např. pro $N = 3$ vyjde součet 15.

Heuristika

Odhad, kde je asi větší šance na nalezení řešení
→ použije se na určení pořadí, v němž se budou procházet varianty možných pokračování.

Hledáme-li jedno libovolné řešení, zvyšujeme dobrou heuristikou pravděpodobnost, že ho najdeme brzy (neboť pořadí listů ve stromu se změní tak, že listy představující úspěšné řešení se nakupí více vlevo). V důsledku toho projdeme třeba jen velmi malou část stromu → velká časová úspora.

Heuristika nic nezaručuje, proto se nemusí dokazovat její správnost, lze ji použít na základě intuitivního odhadu programátora. Není-li dobrá, výpočet nezrychlí, ale o možnost nalézt řešení jejím použitím nepřijdeme.

Příklad: **Proskákání šachovnice koněm**

Existuje velmi účinná heuristika:

v dané pozici provádět dříve vždy ty skoky, které vedou do pozic s menším počtem dalších pokračování.

Obrovské zrychlení výpočtu o několik řádů.

Spojení ořezávání a heuristik

- pokud máme nalézt nejlepší řešení úlohy podle nějakého zadaného kritéria

Vhodnou heuristikou se zajistí, aby se co nejdříve našlo nějaké hodně dobré řešení.
Díky tomu se následně zvýší účinnost ořezávání
(ořezávají se všechny pokusy, u nichž je zřejmé,
že povedou k nalezení horšího řešení).

Příklad: ***Nejkratší cesta mezi dvěma městy***

- neznáme předem mapu ani vzájemné vzdálenosti měst
- v každém městě si pamatujeme, jak nejlépe jsme tam přišli
- přicházíme-li do města delší cestou, nemá smysl pokračovat
- je-li průběžná délka větší než již nalezená cesta do cíle, nemá smysl pokračovat.

Prohledávání stavového prostoru do šířky

= breadth-first search (BFS)
algoritmus „vlny“

- budeme souběžně zkoušet všechny možné varianty pokračování výpočtu, dokud nenajdeme řešení úlohy
  → průchod stromem všech možných cest výpočtu do šířky, po vrstvách
  (v každé vrstvě zkoušíme všechny možnosti zleva doprava)

- je nutné pamatovat si všechny stavy v jedné vrstvě stromu
  → při větší hloubce to může být paměťově velmi náročné (exponenciálně), až nepoužitelné

- použijeme, máme-li nalézt jedno (nejkratší) řešení
  → vždy najde nejkratší možné řešení co do počtu kroků

- výhodné, existují-li v rozsáhlém prohledávacím stromu nějaká řešení vzdálená od startu jen na málo kroků – prohledá se jen několik horních vrstev stromu (významná časová úspora oproti prohledávání do hloubky)

- nevhodné při úloze nalézt všechna řešení – musí se projít celý strom, a to je při stejné časové složitosti výhodnější provádět do hloubky (paměťově mnohem méně náročné)

- v některých úlohách je nutné prohledávat do hloubky (prohledávání do šířky by se paměťově nezvládlo), v některých je nutné prohledávat do šířky (prohledávání do hloubky by se časově nezvládlo), v některých je to jedno (obě varianty prohledávání jsou paměťově i časově stejně náročné)

Implementace prohledávání stavového prostoru do hloubky a do šířky

- již známe z prohledávání binárního stromu,
  zde použijeme v principu stejný postup

- prohledávání *do hloubky* se zpravidla realizuje rekurzivní funkcí,
  totéž lze naprogramovat pomocí **zásobníku** a cyklu

vlož do zásobníku výchozí stav
dokud (není zásobník prázdný) a (není nalezeno řešení) opakuj
    vyjmi ze zásobníku vrchní stav a zpracuj ho
    vlož do zásobníku všechna jeho možná pokračování
        (pouze dosud nenavštívené stavy!)

- prohledávání *do šířky* se realizuje podobně pomocí **fronty** a cyklu

- do fronty ukládáme navštívené stavy,
  které je třeba ještě zpracovat

- musíme evidovat všechny již navštívené stavy
  a nezařazovat je do fronty opakovaně

vlož do fronty výchozí stav
dokud (není fronta prázdná) a (není nalezeno řešení) opakuj
    vyjmi z fronty první stav a zpracuj ho
    vlož do fronty všechna jeho možná pokračování
        (pouze dosud nenavštívené stavy!)

Příklady:

Proskákání šachovnice koněm *(bylo)*
- každé možné řešení úlohy má délku 63 kroků,
  prohledávání do šířky je paměťově nereálné

Nejkratší cesta koněm na šachovnici
- dána výchozí a cílová pozice
- úkol: dojít šachovým koněm co nejmenším počtem tahů
  z výchozí do cílové pozice
- prohledávání do hloubky je velmi nevhodné – backtracking
  do hloubky až 63
  (přitom vždy existuje cesta délky maximálně 6)

Postup řešení:

- postupujeme do šířky: sousední políčka k výchozímu označíme 1,
  jejich sousedy označíme 2, jejich sousedy 3, atd.,
  dokud nedojdeme na cílové políčko
  (číslo = délka nejkratší cesty ze startu)

  – *algoritmus „vlny“*

- k nalezení vlastní cesty je nutný ještě *„zpětný chod“*
  (cíl $\rightarrow$ sousední políčko s hodnotou o 1 menší $\rightarrow$ … $\rightarrow$ start)

Možnosti implementace:

1. Políčka šachovnice z předchozího kroku vlny (tzn. políčka s číslem o 1 menším) nevyhledávat vždy procházením celé šachovnice, ale ukládat si jejich souřadnice do fronty
→ je potřeba paměť navíc na uložení fronty, ale urychlí to výpočet ve fázi algoritmu „vlny“

2. U každého políčka na šachovnici si pamatovat nejen minimální počet kroků, které na něj vedou ze startu, ale i souřadnice předchozího políčka
→ snadné nalezení předchůdce při zpětném chodu (ale zase je to paměťově i časově náročnější při provádění „vlny“)

Obě uvedená vylepšení implementace použijeme později také při hledání nejkratší cesty v grafu.

```
n = 8                  # rozměr šachovnice
s = [[-1]*(n+1) for _ in range(n+1)
                       # vzdálenost pole od startu
pred = [[None]*(n+1) for _ in range(n+1)]
                       # předchůdce na nejkratší cestě
tah = ((1,2), (2,1), (2,-1), (1,-2),
       (-1,-2), (-2,-1), (-2,1), (-1,2))
                       # přírůstky souřadnic při tahu koněm

start1, start2 =
    [int(_) for _ in input('Startovní pole: ').split()]
cil1, cil2 =
    [int(_) for _ in input('Cílové pole: ').split()]
s[start1][start2] = 0
fronta = [(start1, start2)]
                       # fronta řídí průchod do šířky
```

```
while s[cil1][cil2] == -1: # dokud nejsme v cíli
    i1, i2 = fronta.pop(0) # vyzvedneme z fronty (i1,i2)
    krok = s[i1][i2] + 1
    for smer in range(len(tah)):   # zkusíme všemi směry
        j1 = i1 + tah[smer][0]
        j2 = i2 + tah[smer][1]
        if j1 >= 1 and j1 <= n and j2 >= 1 and j2 <= n:
            if s[j1][j2] == -1:    # pole nenavštíveno
                s[j1][j2] = krok   # půjdeme na (j1,j2)
                pred[j1][j2] = (i1, i2)
                fronta.append((j1, j2))
                                   # vložíme do fronty
```

```
print('Nejkratší cesta v pořadí od cíle ke startu:')
print((cil1, cil2), end=' ')
i1, i2 = cil1, cil2
while (i1, i2) != (start1, start2):
    # rekonstrukce cesty - zpětný chod
    j1, j2 = pred[i1][i2]
    i1, i2 = j1, j2
    print((i1, i2), end=' ')
print()
```

Příklad: **Procházení grafu**

Úkol:
- v obyčejném neorientovaném grafu projít ze zvoleného výchozího vrcholu všechny dostupné vrcholy (např. pro určení komponenty souvislosti)

Postup řešení:

- označujeme všechny již navštívené vrcholy, abychom někam nešli dvakrát (abychom nezačali v grafu chodit v cyklu)

- můžeme zcela rovnocenně použít průchod do hloubky nebo do šířky, liší se pořadím návštěvy jednotlivých vrcholů, ale oba najdou stejné řešení se stejnou časovou i paměťovou složitostí

Možnosti realizace:

1. rekurzivní funkce realizující průchod do hloubky
- volá sama sebe pro všechny dosud nenavštívené sousední vrcholy

2. zásobník nebo fronta pro řízení průchodu do hloubky resp. do šířky
- představují seznam „dluhů“ = čísla těch vrcholů, které ještě musíme zpracovat (do nichž jsme při prohledávání už přišli, ale z nichž jsme ještě nezkoušeli jít dál)

3. seznam „dluhů“ dokonce nemusí být realizován ani jako zásobník nebo fronta, můžeme z něj vybírat vrchol ke zpracování libovolně (pak průchod grafem neprobíhá do hloubky ani do šířky)