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
$\rightarrow$ 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.

Algoritmus minimaxu

- hra dvou hráčů s úplnou informací
- bílý a černý se pravidelně střídají na tahu
- cíl algoritmu: pro aktuální pozici zvolit nejvýhodnější tah

- **strom hry** (stavový prostor):
  kořen = aktuální pozice
  počet synů = počet možných tahů
  liché hladiny – na tahu je bílý, sudé – na tahu je černý
  list = konec hry, některý z hráčů vyhrál (příp. remíza)

„Malé“ hry (piškvorky na hodně omezené ploše, odebírání zápalek)

$\rightarrow$ lze postavit celý strom hry, listy ohodnotit podle výsledku hry
(1 = vyhrál bílý, -1 = vyhrál černý, příp. 0 = remíza).

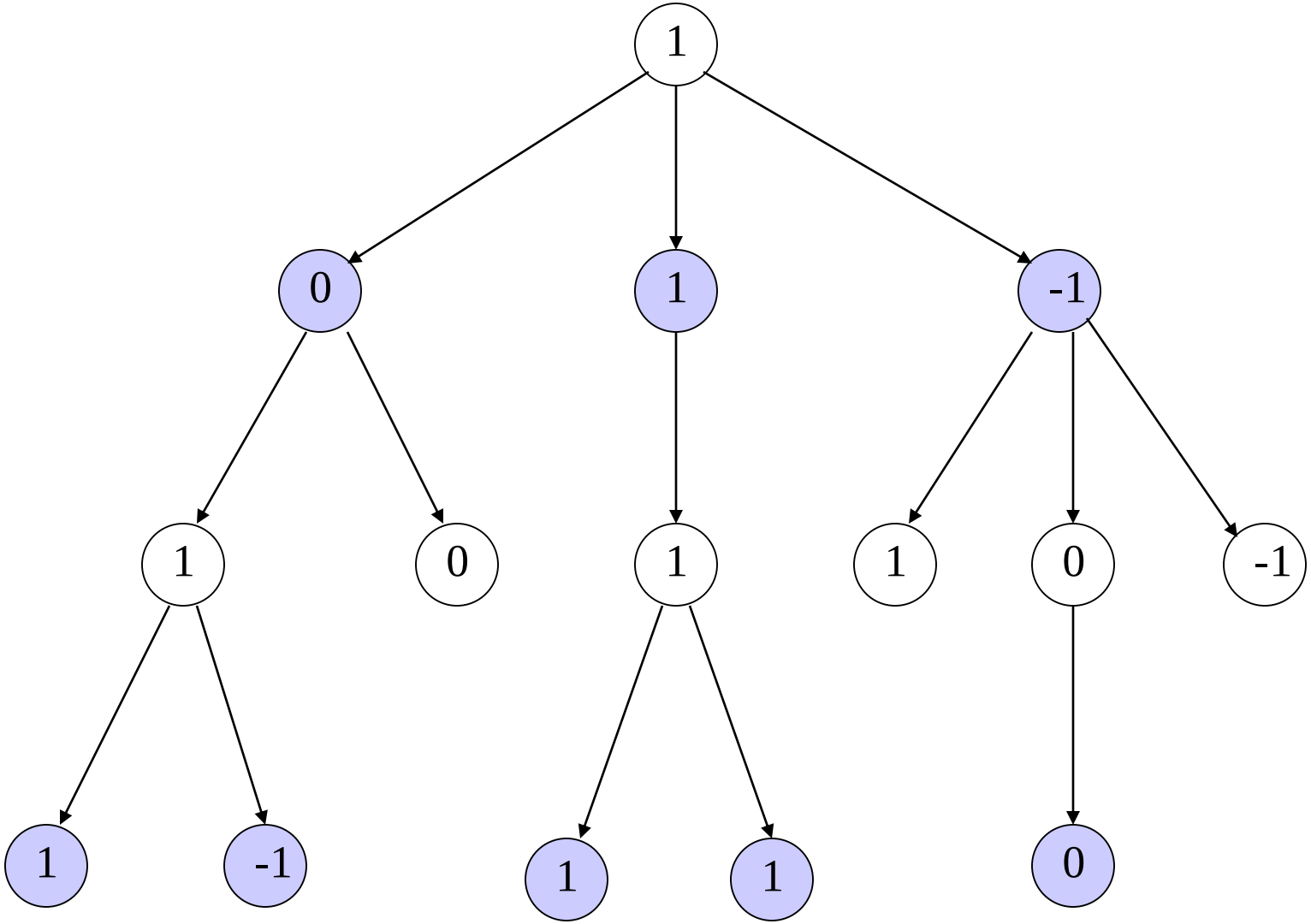
Z hodnot listů se algoritmem minimaxu postupně zdola určí hodnota všech ostatních uzlů, až se získá hodnota kořene
(= nejlepší výsledek, který si může začínající hráč vynutit).

Algoritmus minimaxu:
- hodnota uzlu, kde je na tahu bílý = *maximum* z hodnot jeho synů (bílý si vybere ten tah, který je pro něj nejlepší)
- hodnota uzlu, kde je na tahu černý = *minimum* z hodnot jeho synů (také černý si vybere ten tah, který je zase pro něj nejlepší)

Strom hry se ohodnocuje zdola od listů po vrstvách, v jednotlivých vrstvách stromu se počítají střídavě minima a maxima z hodnot synů, dokud se nezíská hodnota kořene.

Pokud je v kořeni na tahu bílý a kořen bude mí hodnotu 1, může si bílý vynutit vítězství. Pokud získá kořen hodnotu 0, může si začínající bílý vynutit aspoň remízu. Bude-li mít kořen hodnotu –1, bílý si vítězství ani remízu vynutit nemůže (což neznamená, že nemůže vyhrát – ale jedině při chybě černého).

Zvolený tah: ten, který vede z kořene do toho uzlu, kde je ohodnocení stejné jako v kořeni.


1
0
1
-1
1
0
1
1
0
-1
1
-1
1
1
0

Programová realizace

- strom hry je rozsáhlý, není vhodné (nebo ani nelze) vygenerovat ho najednou celý, uložit do paměti a pak zdola po vrstvách procházet
- algoritmus je realizován prohledáváním (tzn. zároveň i vytvářením) stromu hry **do hloubky**, vždy při návratu z podstromu se přepočítá hodnota uzlu, do něhož se vracíme

Minimax – na jednotlivých hladinách stromu hry se hodnoty uzlu počítají střídavě jako minimum a maximum z hodnot synů (musíme vědět, který hráč je na tahu).

Negamax (jiná realizace téhož algoritmu) – hodnota každého uzlu se počítá jako maximum z hodnot synů, před předáním výsledné hodnoty z uzlu nahoru se změní její znaménko.

$$\min(a,b) = -\max(-a,-b)$$

„Velké“ hry (šachy)
→ lze postavit jenom část stromu (zvolený počet hladin),
listy ohodnotit podle statické ohodnocovací funkce
(ocenit materiál každého hráče, příp. vhodné bonusy za pozici):

kladná hodnota = pozice výhodnější pro bílého
(čím vyšší hodnota, tím výhodnější pozice)
záporná hodnota = pozice výhodnější pro černého
(čím vyšší absolutní hodnota, tím výhodnější pozice)
příp. 0 = vyrovnaná pozice

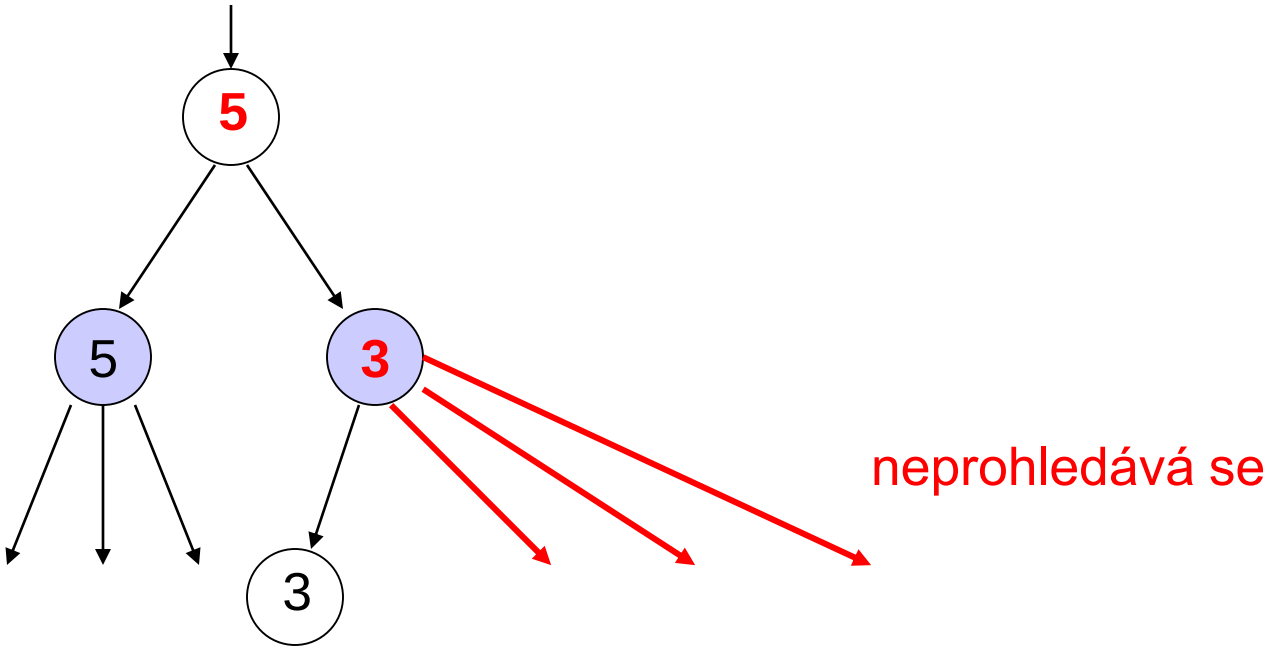
Dále použijeme algoritmus minimaxu stejně jako v předchozím případě.

Vylepšení ohodnocení: pokud by se stala listem „živá“ pozice (tah do ní vedoucí zásadně mění situaci na hracím plánu, např. braní figury v šachu), rozvíjí se zde lokálně strom hry dále do hloubky.

Zrychlení výpočtu – ořezávání stromu hry

- ztrátové: po několika málo vrstvách provést statické ohodnocení pozic a část nejhorších pozic odmítnout (tedy dále nerozvíjet), podrobněji do větší hloubky analyzovat jen nadějnější pozice
 → kaskádové vyhodnocování

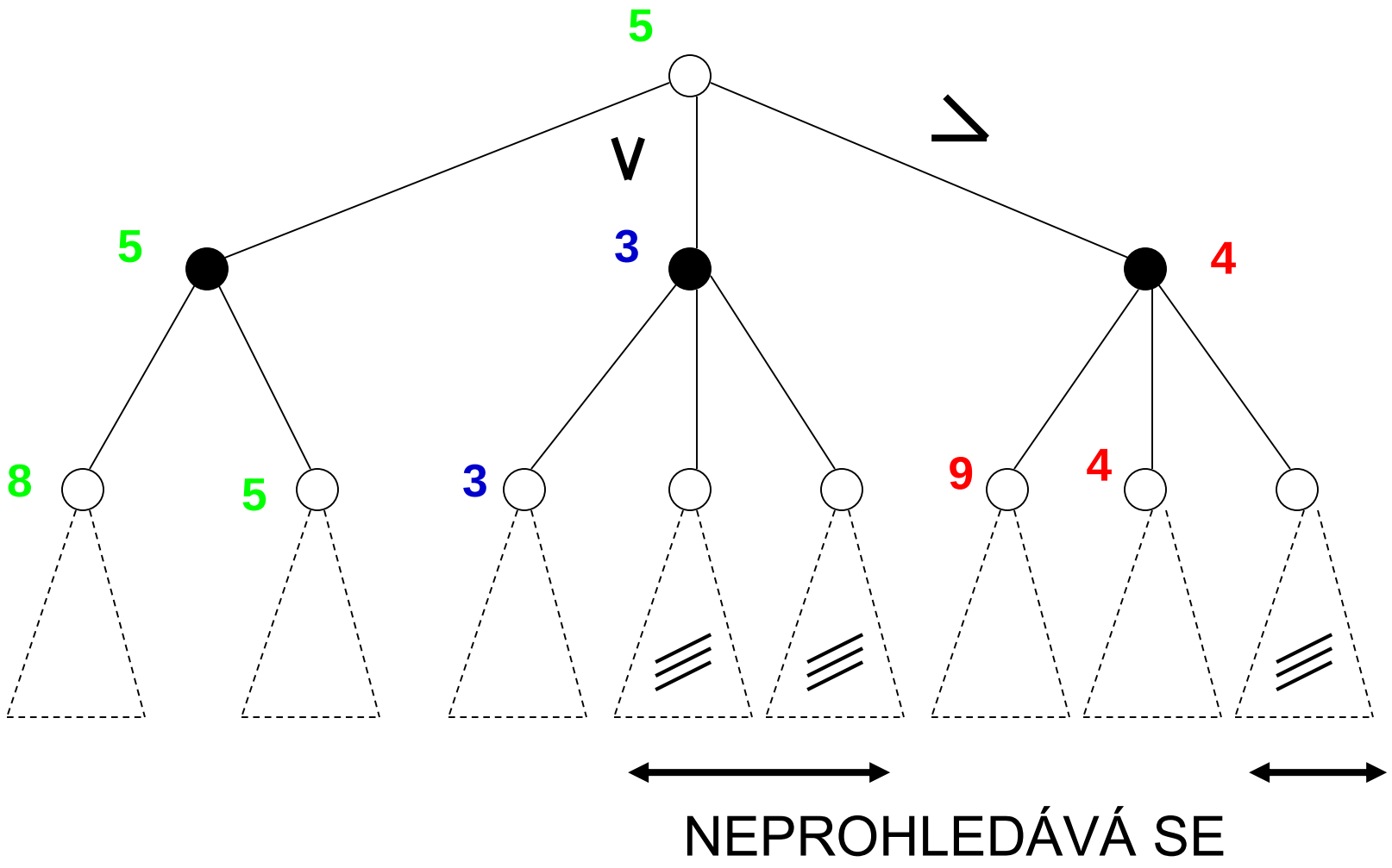
- bezztrátové: *alfa-beta-prořezávání*



- analogické prořezávání se provádí při jednom průchodu stromem hry ve všech vrstvách stromu, pro bílého i pro černého

- pokud hráč v každé pozici zkouší přednostně ty tahy, které jsou pro něj výhodnější, alfa-beta-prořezávání je výrazně účinnější (prořeže se více větví, prochází se mnohem menší část stromu)

$\rightarrow$ uspořádat možné pokračovací tahy v každé pozici podle statické ohodnocovací funkce nebo podle nějaké heuristiky


5
5
3
4
8
5
3
9
4
NEPROHLEDÁVÁ SE