Umělá inteligence I

Roman Barták, KTIML

`roman.bartak@mff.cuni.cz`
`http://ktiml.mff.cuni.cz/~bartak`

3

Na úvod

- **Agent s reflexy** „pouze" převádí současný vjem na jednu akci.
- **Agent s cílem** umí plánovat několik akcí dopředu pro dosažení cíle.
  - Podívejme se na jeden typ agentů s cílem – **agenty řešící úlohy**.
  - Co je to **úloha** a co je její **řešení**?
  - Jak **nalézt řešení** úkolu?
  - **Prohledávací** algoritmy BFS, DFS, DLS, IDS, BiS

Poznámka:
Budeme se zabývat „neinformovanými" algoritmy, tj. algoritmy, které nevyužívají speciální znalost problému.

Příklad

- Inteligentní agenti maximalizují míru svého výkonu, což si mohou zjednodušit formulací cílů a snahou jich dosáhnout.

Situace a možný postup řešení

- agent je v Rumunsku ve městě Arad
- míra výkonu se skládá z mnoha složek (chce být opálený, užít si noční život, prohlédnout okolí, …), což komplikuje rozhodovací proces
- nechť ale agent má na zítra nevratnou letenku z Bukurešti
- **formulací cíle** – dostat se do zítřka do Bukurešti – se rozhodovací problém zjednodušuje
- cíl určuje žádoucí stav světa, potřebuje ale další stavy světa, přes které se k cíli dostaneme, a akce, které nás mezi stavy budou posouvat – **formulace úlohy**
  - stavy mohou být města, přes která pojedeme (aktuální poloha na silnici je zbytečně jemná)
  - akce mohou být přejezdy mezi městy (akce typu zvedni a polož nohu jsou příliš jemné)
- Jak ale najdeme cestu do Bukurešti, když na rozcestí jsou jen tři nejbližší města a žádné není Bukurešť?
  - Uvažujme, že máme mapu Rumunska, můžeme tedy prozkoumat různé cesty (**prohledávání**), vybrat tu nejlepší a potom se po ní vydat (**exekuce**).

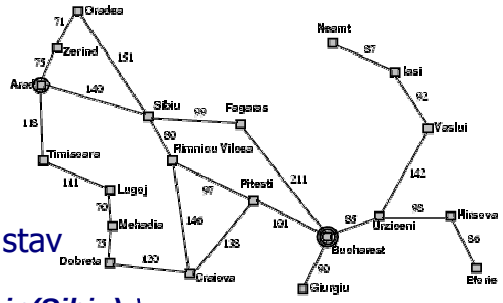
Řešení úloh

Řešení úloh se skládá ze čtyř hlavních kroků:

- formulace cíle
  - Jaké jsou žádoucí stavy světa?
- formulace úlohy
  - Jaké akce a stavy je vhodné uvažovat pro dosažení cíle?
- **vyřešení úlohy**
  - Jak najít nejlepší posloupnost stavů vedoucí k cíli?
- realizace řešení
  - Známe-li požadované akce, jak je budeme realizovat?

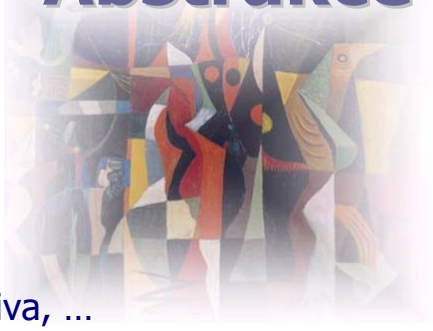
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Formulace úloh



- **Dobře formulovaná úloha** se skládá z:
  - **počátečního stavu**
    - ***in(Arad)***
  - popisu možných **akcí**
    typicky funkcí následníka, který pro každý stav vrátí akci a stav, kam nás akce převede
    - ***SUCCESSOR-FN(in(Arad)) = ⟨ go(Sibiu), in(Sibiu) ⟩***
    - implicitně definuje **stavový prostor** (množina stavů dosažitelných z počátečního stavu)
    - **cesta** je potom posloupnost stavů propojených akcemi
  - **testu cíle**
    funkce určující zda daný stav je cílový nebo ne, může být třeba výčet cílových stavů – ***{ in(Bucharest) }***
  - **ceny cest**
    numerická cena každé cesty, určuje míru výkonu agenta
    - Budeme předpokládat, že cena cesty se skládá z nezáporných cen akcí po cestě.
- **Řešením úlohy** je cesta z počátečního stavu do stavu cílového.
- **Optimální řešení** je řešení z nejmenší cenou cesty mezi všemi řešeními.

Abstrakce



- V naší formulaci jsme použili
  - **abstrakci stavu světa**
    - ignorujeme počasí, stav silnice, …
  - **abstrakci akcí**
    - neuvažujeme zapnutí rádia, čerpání paliva, …
- Jaká je **správná abstrakce**?
  - abstraktní řešení lze rozvést na konkrétní (**validita**)
    - z libovolného místa v Aradu najdeme cestu do libovolného místa v Sibiu
  - provádění akcí z řešení je snazší než původní úloha (**užitečnost**)
    - cestu z Aradu do Sibiu můžeme svěřit „řidiči" bez dalšího plánování

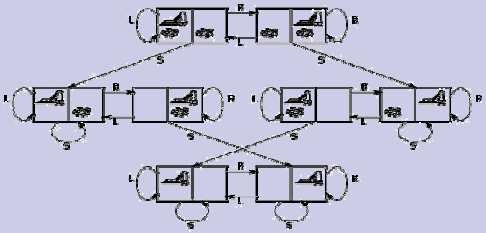
Bez možnosti pracovat s abstrakcemi by nás reálný svět zahltil.

Hračky

- Hračky (**toy problems**) slouží pro ilustraci a porovnání řešících technik.

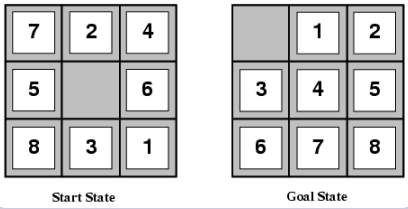
Svět vysavače

stavy (robot × smetí)
počátek, cíl
následník (L, R, S)



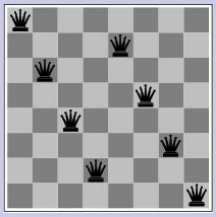
Loydova osmička

stavy (pozice kostiček)
počátek, cíl
následník (L, R, U, D)



8-královen

umístit na šachovnice 8 královen bez ohrožení
- inkrementální reprezentace (přidáváme královny)
- reprezentace úplných stavů (posouváme královny)

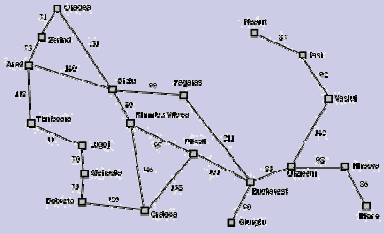


Reálné úlohy

- To je to, co lidi zajímá.

Hledání cest

stavy (reprezentace míst)
počátek (teď a tady), cíl (tam a včas)
následník
cenová funkce (čas, cena,...)

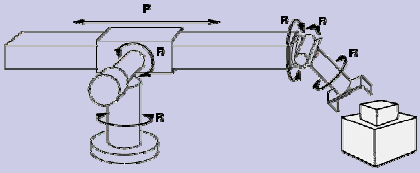


Okružní cesty

cíl (navštívit každé město alespoň jednou)
stavy (aktuální a navštívený místo)
TSP (Travelling Salesman Problem), NP-těžký

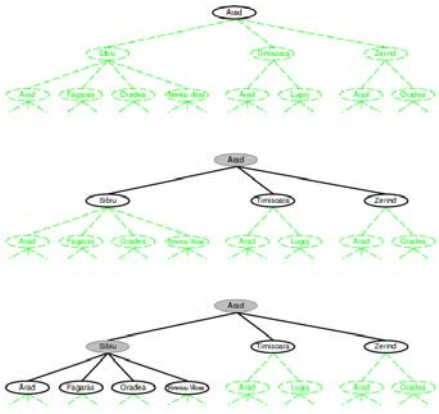
Sestavování produktů

stav (pozice robotické ruky × součástky)
cíl (sestavený produkt)
následník (pohyby „kloubů")
cena (celkový čas sestavení
sestavení proteinů z posloupnosti aminokyselin



Hledáme řešení

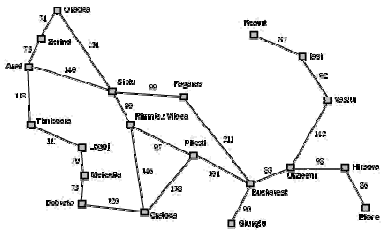
- Když už máme formulované úlohy, jak je vyřešíme?
- **Prohledávání stavového prostoru**
  - začneme v **počátečním stavu** (kořenový uzel)
  - **otestujeme**, zda počáteční stav není zároveň **cílový**
  - pokud není cílový, provedeme **expanzi stavu**, čímž generujeme novou množinu stavů
  - **vybereme další stav** pro pokračování pomocí **prohledávací strategie**

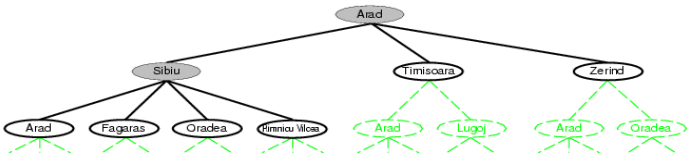


```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

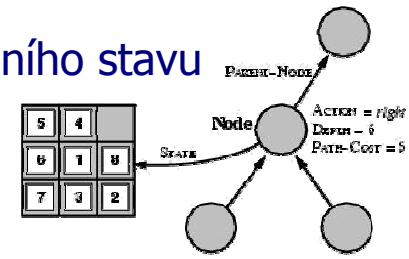

Uzel vs. stav

- **Stavový prostor se liší prohledávacího stromu stejně jako stav není uzel stromu!**



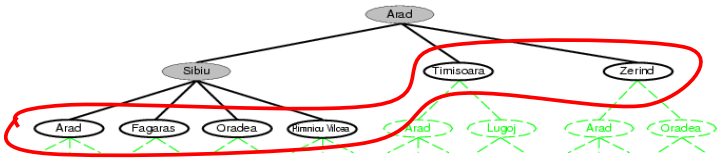


- **Uzel** prohledávacího stromu obsahuje
  - aktuální **stav**
  - odkaz na **rodiče**
  - **akci**, která vedla od rodiče do aktuálního stavu
  - **cenu** cesty (z kořene), g(n)
  - **hloubku** (počet kroků od kořene)



Okraj

- Při prohledávání potřebujeme také reprezentovat soubor uzlů pro expandování – **okraj** (fringe).



- Uzly z okraje jsou nazývány **listy**.
- Abychom usnadnili výběr uzlu z okraje, budeme okraj reprezentovat **frontou** s následujícími operacemi:
  - MAKE-QUEUE(element,…)
  - EMPTY?(queue)
  - FIRST(queue)
  - REMOVE-FIRST(queue)
  - INSERT(element, queue)
  - INSERT-ALL(elements, queue)

Stromové prohledávání

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Měření výkonu

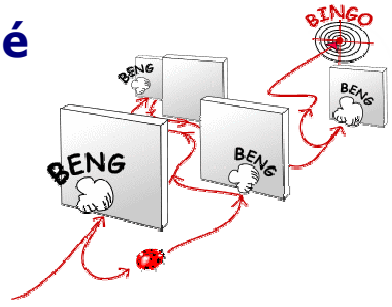
- Prohledávací algoritmus vrátí **řešení** nebo **neúspěch**.
- **Výkon algoritmu** budeme posuzovat ze čtyř hledisek:
  - **úplnost**
    - Pokud existuje řešení, najde ho?
  - **optimalita**
    - Najde algoritmus optimální řešení?
  - **časová složitost**
    - Jak dlouho trvá nalezení řešení?
  - **prostorová složitost**
    - Kolik paměti pro prohledávání potřebujeme?

  - Čas a prostor typicky měříme vzhledem k nějaké míře složitosti problému.
    - **větvící faktor b** (maximální počet následníků) – vhodné, když je stavový prostor reprezentován implicitně (počátek a následníci)
    - **hloubka d** (hloubka nejméně zanořeného cílového uzlu)
    - **cesta m** (délka maximální cesty ve stavovém prostoru)
- **cena prohledávání** (kolik času a prostoru potřebujeme)
- **celková cena** (kombinuje cenu prohledávání a cenu nalezeného řešení)

Informovanost

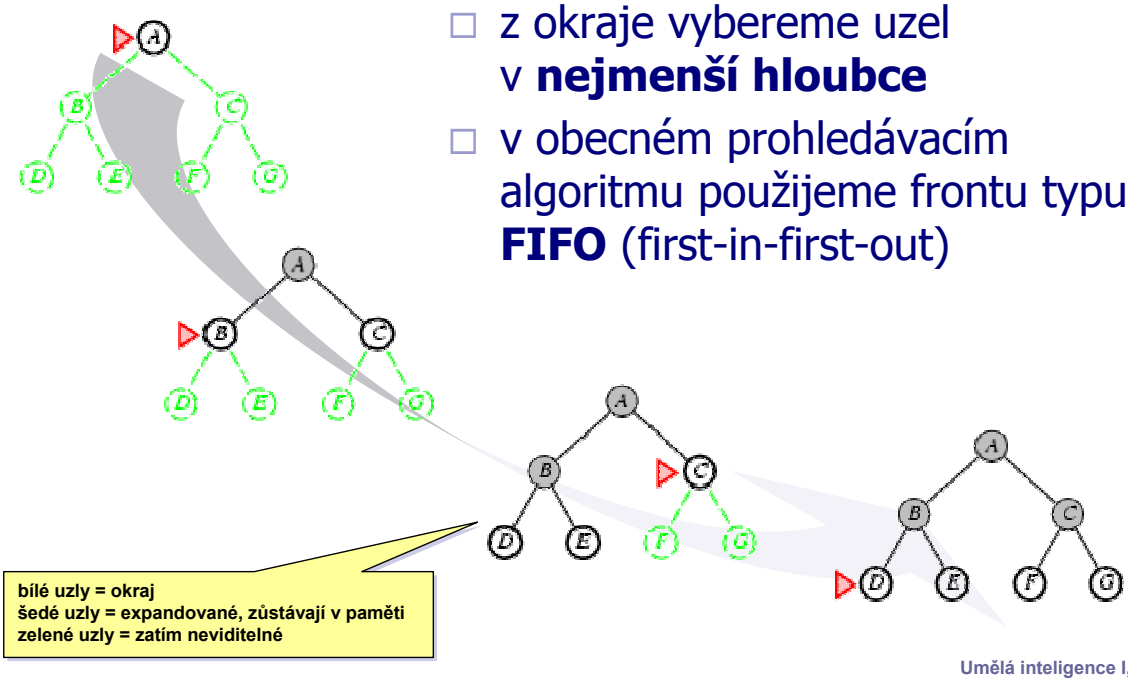
- Dnes se podíváme na tzv. **neinformované** (**slepé**) prohledávání
  - nemá **žádnou dodatečnou informaci** o stavech, kromě popisu úlohy
  - tj. umí rozpoznat cílový stav od necílového a generovat následníky

- Příště doplníme **informované** (**heuristické**) prohledávání
  - u stavů dokáže rozlišit, jak jsou **slibné** na cestě k řešení
  - např. prostřednictvím odhadu ceny zbývající do cíle



Jdeme do šířky

- Vždy nejprve **expandujeme celou vrstvu** prohledávacího stromu než postoupíme do další
  - z okraje vybereme uzel v **nejmenší hloubce**
  - v obecném prohledávacím algoritmu použijeme frontu typu **FIFO** (first-in-first-out)



Jdeme do šířky
analýza

- Je to metoda **úplná** (za předpokladu konečného větvení).
- Nejdříve nalezený cíl **nemusí být optimální!**
  - optimum lze garantovat, pokud je cena neklesající funkcí hloubky
- **Časová složitost** (počet navštívených uzlů pro cíl v hloubce d, který je na konci vrstvy)
  - $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = \mathbf{O(b^{d+1})}$
- **Paměťová složitost**
  - v paměti drží všechny navštívené uzly, tj. stejná jako časová složitost
  - $\mathbf{O(b^{d+1})}$
- **Paměť je větší problém než čas ale ani čas není zanedbatelný.**

hloubka	uzly	čas	paměť
2	1100	0.11 sec.	1 megabyte
4	111100	11 sekund	106 megabytes
6	10^7	19 minut	10 gigabytes
8	10^9	31 hodin	1 terabyte
10	10^{11}	129 dní	101 terabytes
12	10^{13}	35 let	10 petabytes
14	10^{15}	3523 let	1 exabyte

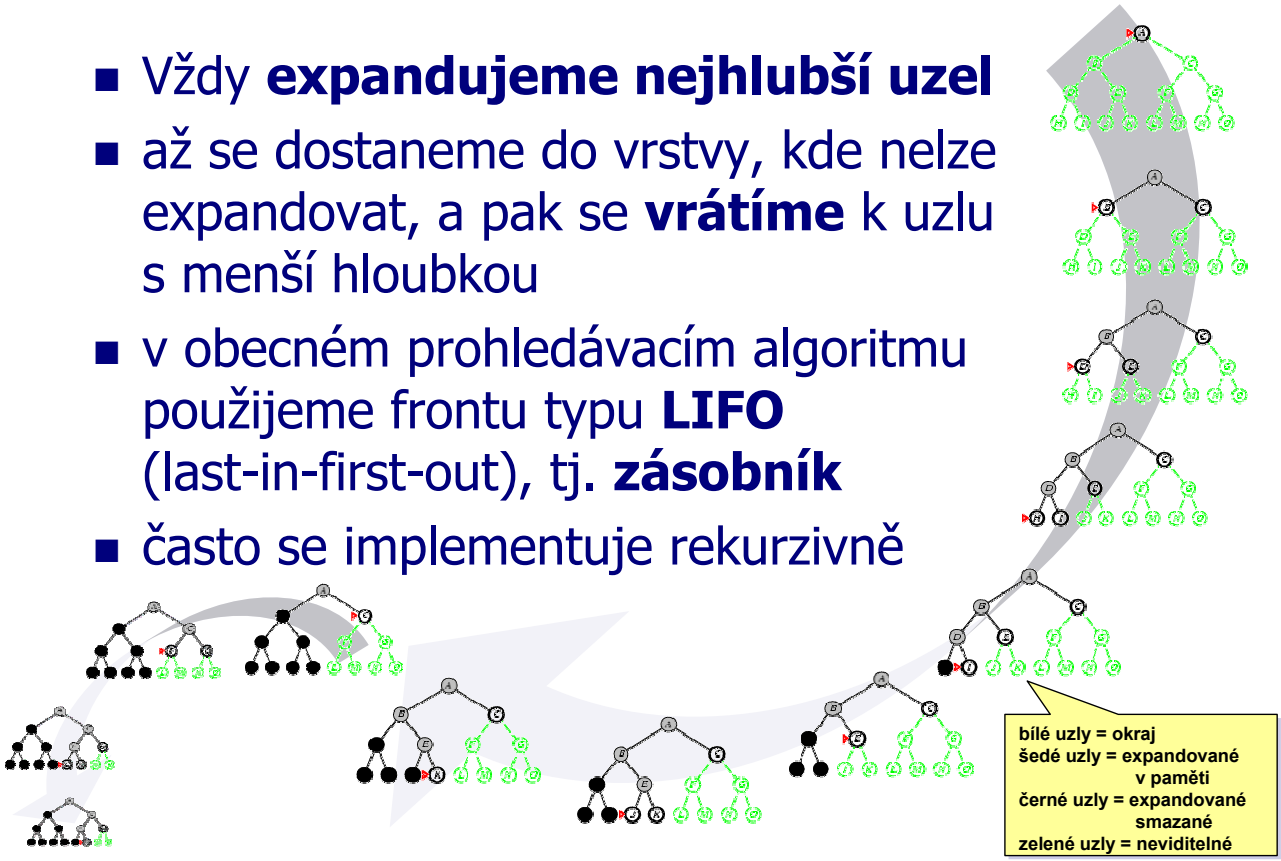
b = 10
10.000 uzlů/sec.
1000 bytů/uzel

Do šířky dle ceny

- Modifikace BFS pro **hledání optimálního řešení.**
- Jako první expandujeme uzel s **nejmenší cenou** cesty od kořene.
- Pozor na kroky s nulovou cenou, mohou vést k cyklům!
- **Úplnost** (a nalezení optima) lze garantovat, pokud je cena každé akce větší či rovna konstantě ε.
- **Časová (a paměťová) složitost**
  - záleží na ceně cesty spíše než na hloubce
  - **$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$**, kde C* je cena optimálního řešení
  - může být **mnohem horší než u BFS**, protože se často prohledávají dlouhé cesty s „krátkými" kroky než se zkusí cesty s delšími a možná vhodnějšími kroky

Jdeme do hloubky

- Vždy **expandujeme nejhlubší uzel**
- až se dostaneme do vrstvy, kde nelze expandovat, a pak se **vrátíme** k uzlu s menší hloubkou
- v obecném prohledávacím algoritmu použijeme frontu typu **LIFO** (last-in-first-out), tj. **zásobník**
- často se implementuje rekurzivně



Jdeme do hloubky
analýza

- Pokud se vydá špatnou cestou, nemusí najít řešení (**není úplný**) a samozřejmě **nemusí najít optimum**.
- **Časová složitost**
  - **O(b^m)**, kde m je maximální hloubka
  - může se stát, že d << m, kde d je hloubka řešení
- **Paměťová složitost**
  - stačí si pomatovat jednu cestu z kořene společně se sousedními uzly
  - expandovaný uzel s prozkoumanými potomky lze odstranit
  - **O(bm)**, kde m je maximální navštívená hloubka
  - Lze ještě zlepšit technikou známou jako **backtracking**!
    - generuje jednoho následníka (místo všech) → **O(m)** stavů
    - mění stav na místě (místo kopírování), pokud tedy umí obnovit původní stav na základě akce → **O(1)** stavů a **O(m)** akcí

Omezení hloubky

- Problém s neomezenými stromy můžeme řešit přidáním **limitu l na hloubku prohledávání.**
  - uzly v hloubce l se považují jako uzly bez následníků
  - algoritmus vrací řešení, neúspěch nebo vyčerpání limitu (cut-off)
  - **časová** složitost **O(b^l)**, **prostorová** složitost **O(bl)**
  - pokud l < d, pak řešení nenajdeme (d je hloubka řešení)
  - pokud d << l, pak často prohledáváme zbytečně mnoho

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Jak určit vhodný limit?

- použitím informací o problému
- např. pro hledání cesty ve světě 20-ti měst, můžeme použít limit 19
- studiem mapy můžeme získat ještě přesnější limit – mezi libovolnými dvěma městy se lze dostat do 9-ti kroků (tzv. **průměr grafu**)

Iterování hloubky

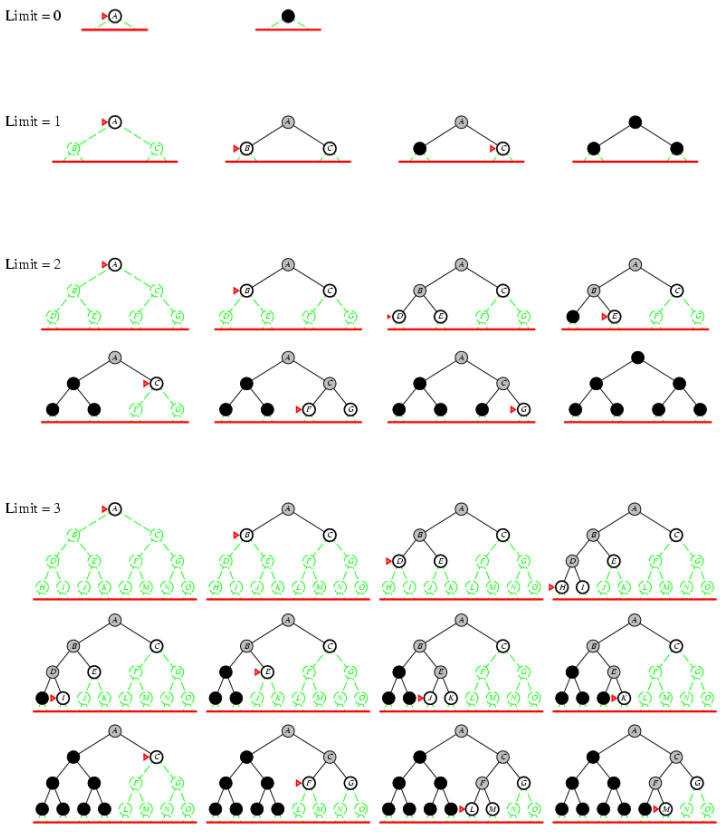
Jak zúplnit prohledávání s omezenou hloubkou?

- budeme postupně zvětšovat limit l
- spojíme tak výhody BFS a DFS
  - **úplnost** (pro konečné větvení)
  - garance **optimality** (pro cenu závislou na hloubce)
  - malé **paměťové** nároky **O(bd)**
  - a co **časová** složitost?
    - d b^1 + (d-1)b^2 + … + 3b^{d-2} +2b^{d-1} + 1b^d = **O(b^d)**
    - fakticky lepší než BFS, které často přidá ještě jednu vrstvu
      - Př. (b= 10, d = 5): BFS = 1.111.100, DFS = 111.110, IDS = 123.450
- **IDS je preferovaná metoda slepého prohledávání ve velkém prostoru a s neznámou hloubkou řešení.**

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

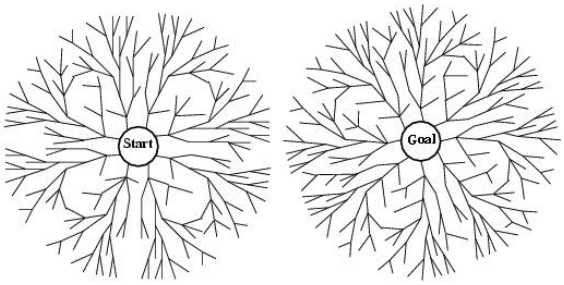

Iterování hloubky

příklad



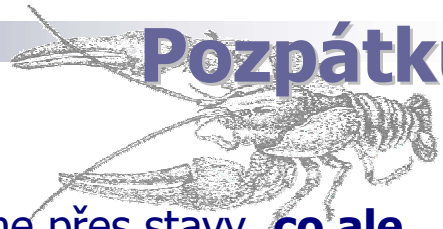
Z obou stran

- Můžeme také **kombinovat prohledávání od počátku s prohledáváním od cíle** (dokud se někde nepotkají).



- **Proč?**
  - $b^{d/2} + b^{d/2} << b^d$
  - Př. (b= 10, d = 6): BFS = 11.111.100, BiS = 22.200
- **Jak?**
  - Před expanzí uzlu vždy zkontrolujeme, zda není v okraji druhého prohledávacího stromu.
  - Potřebujeme mít v **paměti** alespoň jeden ze stromů **$O(b^{d/2})$**, test přítomnosti uzlu lze provést v konstantním čase (hašování).
  - Použijeme-li z obou stran prohledávání do šířky, dostaneme **úplný algoritmus** garantující nalezení **optima**.

Pozpátku?



- Při prohledávání od počátku jdeme přes stavy, **co ale prohledáváme při cestě od cíle?**
  - Je-li cíl jediný stav, můžeme jít také přes stavy (cesta Arad → Bucharest).
  - Je-li cílem více explicitně daných stavů (svět vysavače), můžeme zavést **umělý stav**, jehož jsou dané cílové stavy předchůdci
  - nebo můžeme skupinu stavů vidět jako **jediný meta-stav**.
  - Nejtěžším případem jsou **implicitně dané cílové stavy** (8-královen), kdy je potřeba najít kompaktní popis umožňující test přítomnosti stavu z dopředné fáze.
- Jiný problém je **jak definovat předchůdce stavu**.
  - Předchůdcem n je každý stav, jehož je n následníkem.
  - Nejtěžším případem jsou implicitně definované přechody pomocí nějaké funkce.

Shrnutí

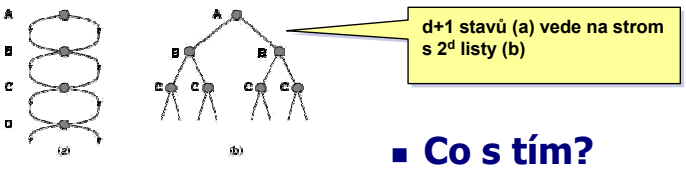
Kritérium	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Úplnost?	ANO*	ANO*	NE	ANO, if $l \geq d$	ANO	ANO*
Čas	b^{d+1}	$b^{C^*/\varepsilon}$	b^m	b^l	b^d	$b^{d/2}$
Prostor	b^{d+1}	$b^{C^*/\varepsilon}$	bm	bl	bd	$b^{d/2}$
Optimalita?	ANO*	ANO*	NE	NE	ANO	ANO

b – větvící faktor
d – nejmenší hloubka (optimálního) řešení
C* – cena optimálního řešení
ε – minimální nárůst ceny (minimální cena akce)
m – maximální navštívená hloubka
l – limit hloubky
* úplnost při omezeném větvení (BFS)
optimalita při minimálním nárůstu ceny resp. jednotkové ceně (BFS)

Závěrečné poznámky

opakující se stavy

- Zatím jsme ignorovali jednu z typických obtíží prohledávání – expanzi již navštívených stavů.
  - ne vždy je to problém (vhodná formulace problému 8-královen)
  - někdy ale výrazně zvětší prohledávaný strom



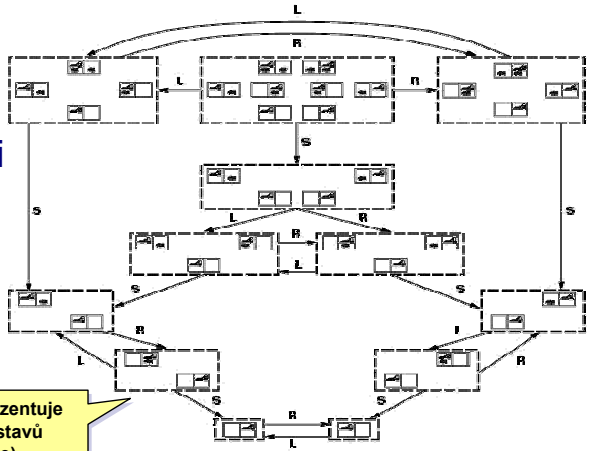
- **Co s tím?**
  - můžeme se pomatovat již expandované stavy – tzv. **uzavřené stavy**
  - pokud se do uzavřeného stavu dostaneme znova, dále ho v prohledávání neexpandujeme

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Závěrečné poznámky

částečná informace

- Zatím jsme uvažovali statické, plně pozorovatelné, diskrétní a deterministické prostředí.
- Co dělat v případě, že agentovi **chybí informace**?
  - **žádné senzory** (nevíme v jakém jsme stavu)
    - pracujeme s **domnělým (belief) stavem** – množina reálných stavů – a hledáme domnělý stav obsahující pouze cílové stavy (*conformant problems*)
  - **neurčitý výsledek akce**
    - plány s alternativními větvemi (*contingency problems*)
  - **neznámé akce**
    - řešení průzkumem (*exploration problems*)



Domnělý stav reprezentuje množinu možných stavů světa (svět vysavače).