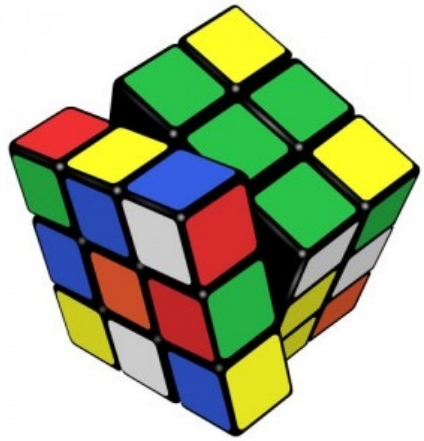
PROGRESIVNÍ TECHNOLOGIE V INFORMATICE II

UMĚLÁ INTELIGENCE
HEURISTICKÉ PROHLEDÁVÁNÍ

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

PROHLEDÁVÁNÍ STAVOVÉHO PROSTORU

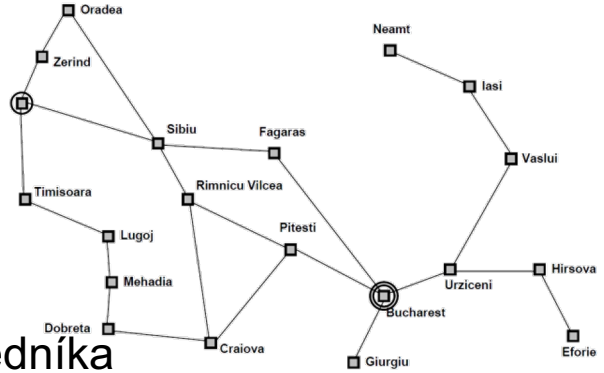
- **Několik kroků vedoucích k řešení úlohy**
  - Formulace **cíle**
    - požadavky → žádoucí stav světa
  - Formulace **úlohy**
    - jak vypadají stavy světa a akce, které jej mění
  - **Vyřešení** úlohy
    - určení posloupnosti stavů/akcí vedoucích k cíli
  - **Provedení** řešení
    - vykonání nalezené posloupnosti akcí
    - může dojít k chybám při vykonávání



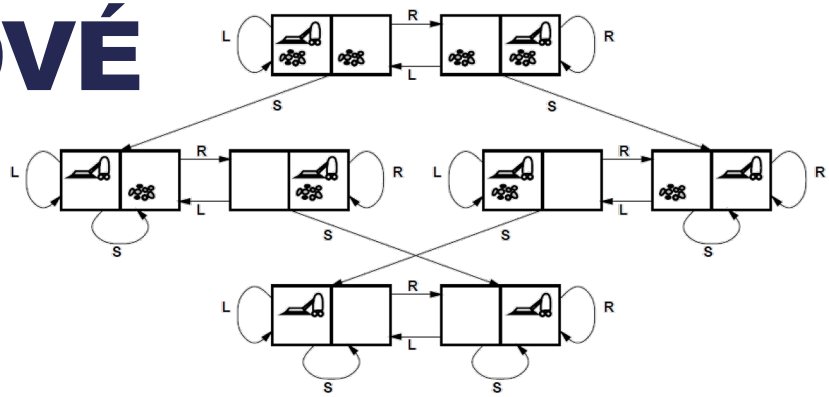
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

FORMULACE ÚLOHY

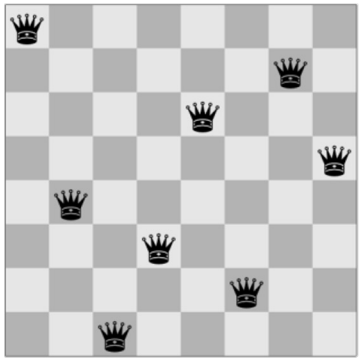
- **Situace**
  - „agent se nachází v **Rumunsku** ve městě **Arad** a chce si užít dovolenou, tj. opálit se, učit se Rumunsky, navštívit památky, okusit noční život, atd.“
- **Zjednodušení**
  - Přijme cíl = jet do Bukurešti
- **Dobře formulovaná úloha a řešení**
  - Počáteční stav
    - `in(Arad)`
  - Popis akcí, zpravidla určené pomocí funkce následníka
    - pro daný stav *x* vrací dvojici *(y,a)* následného stavu *y* a akce *a*, která nás převede z *x* do *y*
    - `{(go(Sibiu),in(Sibiu));(go(Timisoara),in(Timisoara);go(Zerind),in(Zerind)}`
  - Test, zda daný stav je cílový
    - například výčtem: `{in(Bucharest)}`
  - **Řešení**
    - Posloupnost akcí a stavů - cesta
      - **Vrcholy** na cestě – odpovídají stavům, **hrany** – odpovídají akcím

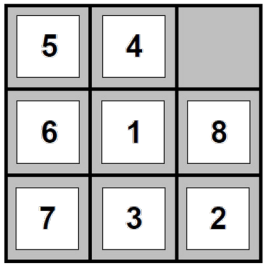


PŘÍKLADY **HRAČKOVÉ**

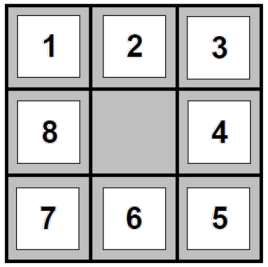


- **Původně zkoumány v rámci mikro-světů**
  - důležité pro vývoj, testování a porovnávání algoritmů
- **Vysavač**
  - úkolem je vysát smetí
    - **stavy:** pozice vysavače a smetí
    - **akce:** vysát smetí, přesun
- **8 královen (obecně N královen)**
  - umístit královny na šachovnici bez vzájemného ohrožení
    - **stavy:** rozmístění královen
    - **akce:** přidání královny
- **Lloydova osmička (Lišák)**
  - srovnat kameny na hracím poli
    - **stavy:** rozložení kamenů
    - **akce**: pohyb kamenem na volné místo





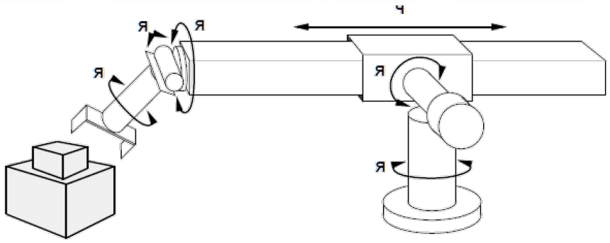
Start State

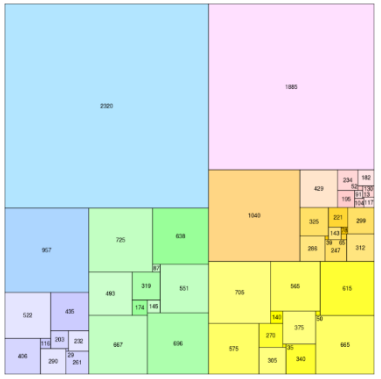


Goal State

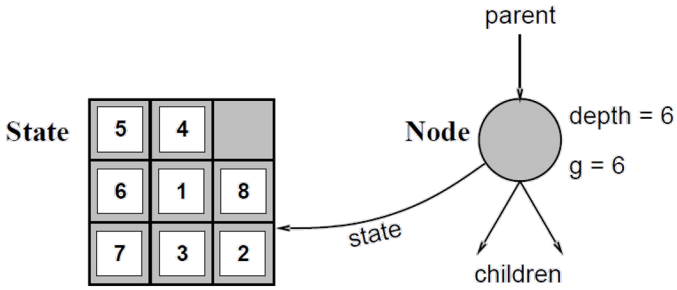
PŘÍKLADY **REÁLNÉ**

- **Těžké pro původní algoritmy UI**
  - obrovský stavový prostor
    - kombinatorická exploze
  - důležité pro praxi
    - průmysl, doprava, obchod, věda
- **Multi-agentní hledání cest, multi-robotické plánování pohybu**
  - mnoho robotů, každý má svůj počátek a cíl
    - **Stavy:** rozložení robotů, konfigurace
    - **Akce:** pohyby robotů
- **Montáž výrobků**
  - Robot má za úkol sestavit výrobek
    - **Stavy:** konfigurace robota a dílů
    - **Akce:** vše, co robot může udělat
- **Perfektní čtverec**
  - UI v roli servisní vědy pro matematiku

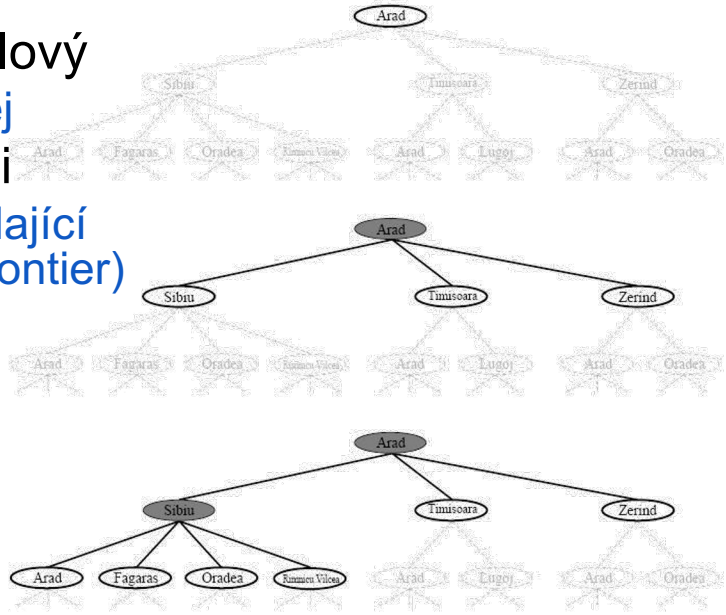




NEINFORMOVANÉ PROHLEDÁVÁNÍ



- **Předběžnosti**
  - stav × uzel
    - **stav** – stav světa
    - **uzel** – datová struktura obsahující stav a další informace (rodič, cena, …)
- **Prohledáváme stavový prostor**
  - Začneme v počátečním stavu
  - Otestujeme, zda počáteční stav není cílový
    - je-li cílový, máme řešení a vrátíme jej
  - Není-li cílový, **vybereme** uzel k expanzi
    - expanzí vzniknou nové uzly, odpovídající následníkům, přidáme do „okraje“ (frontier)



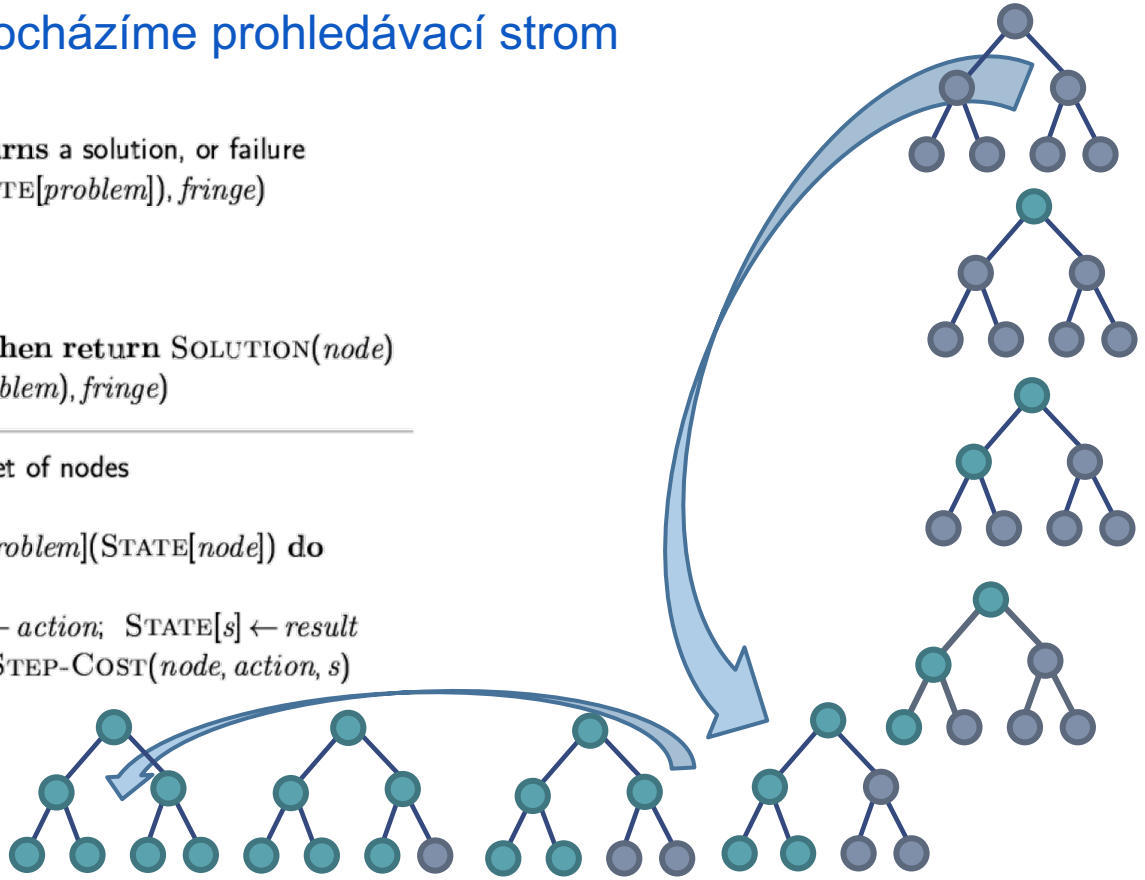
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

PROHLEDÁVÁNÍ **DO HLOUBKY**

- **DFS**: vybíráme uzel v **největší hloubce**
  - okraj implementujeme jako zásobník (**LIFO**)
    - ve výsledku procházíme prohledávací strom **zleva doprava**

```
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)
```

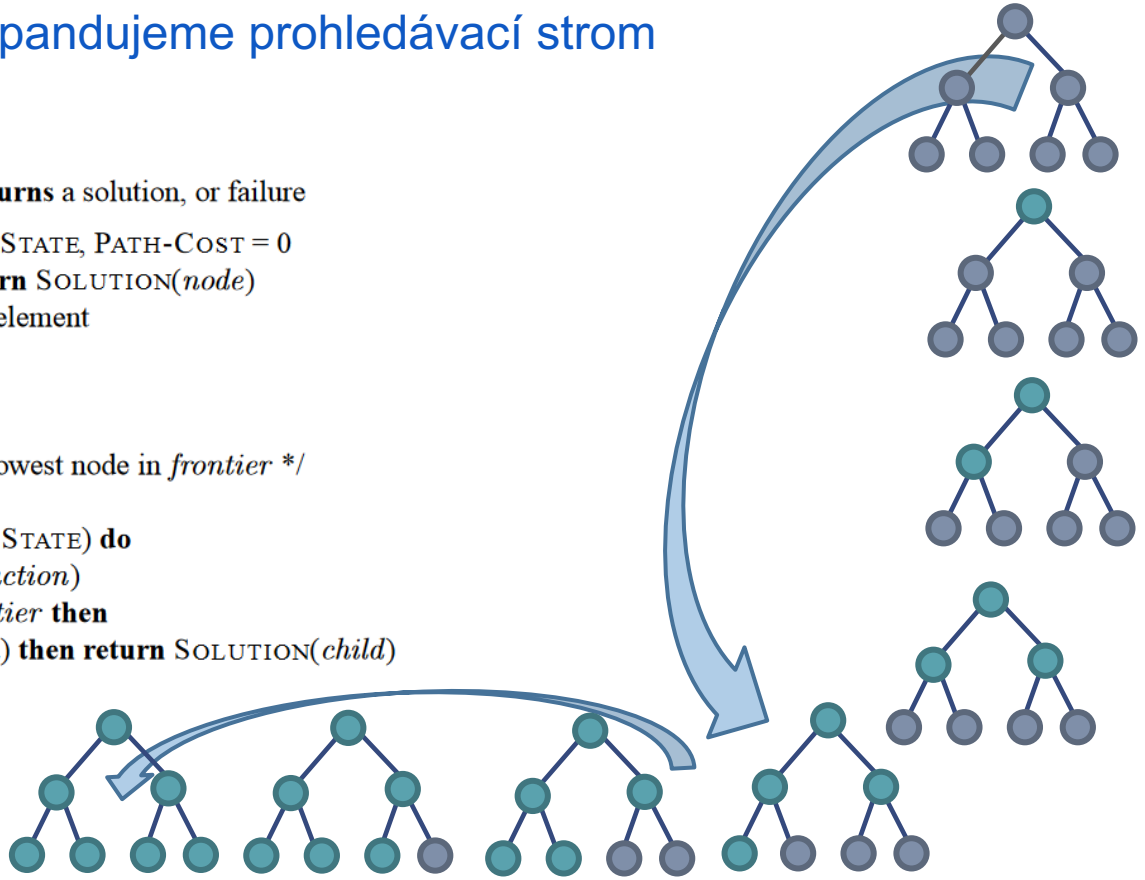
```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node,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



PROHLEDÁVÁNÍ **DO ŠÍŘKY**

- **BFS**: vybíráme uzel v **nejmenší hloubce** (s nejmenší cenou)
  - okraj implementujeme jako frontu (**FIFO**)
    - ve výsledku expandujeme prohledávací strom **po vrstvách**

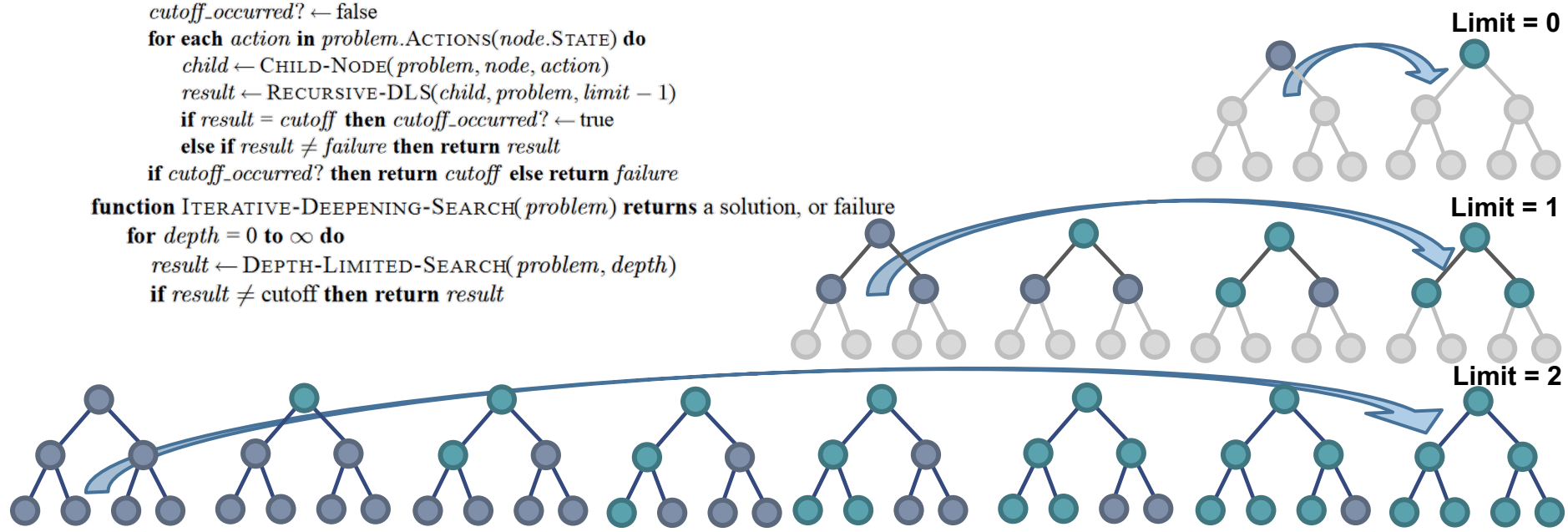
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```



ITERATIVNĚ DO HLOUBKY

- **IDS**: kombinuje výhody DFS a BFS
  - zaručuje **optimalitu** (za předpokladu monotonie) a **úplnost**
  - malé paměťové nároky (jako DFS)

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure

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



VLASTNOSTI DFS A BFS

- **Úplnost**
  - Algoritmus vždy skončí a vrátí správnou odpověď
- **Časová složitost, paměťová složitost**
  - b – větvící faktor, d – nejmenší hloubka obsahující řešení, m – maximální hloubka stromu, l – limit na hloubku

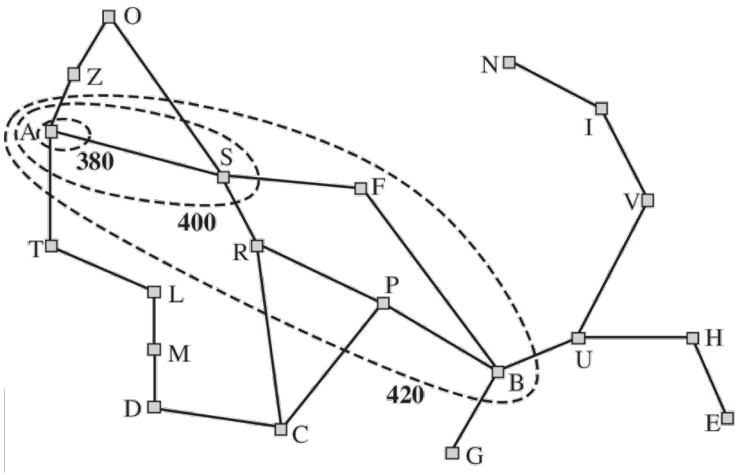
	DFS	BFS	DLS	IDS
Úplnost	NE	ANO	NE	ANO
Čas	$O(b^m)$	$O(b^{d+1})$	$O(b^l)$	$O(b^d)$
Paměť	$O(bm)$	$O(b^{d+1})$	$O(bl)$	$O(bd)$
Optimalita	NE	ANO*	NE	ANO*

- **Ukázka výpočtu časové složitosti pro IDS:**
  - $d \cdot b^1 + (d-1) \cdot b^2 + (d-2) \cdot b^3 + (d-3) \cdot b^4 + \ldots + 2 \cdot b^{d-1} + b^d = O(b^d)$

PROHLEDÁVÁNÍ **HEURISTICKY**

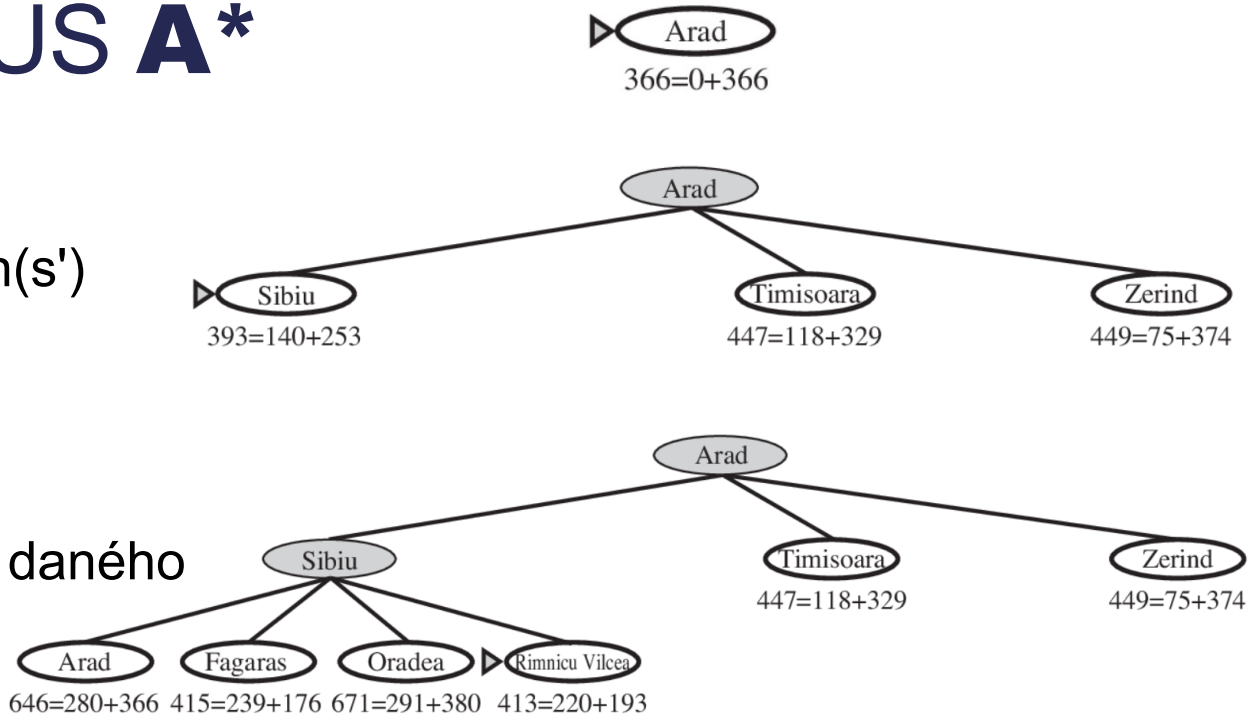
- **Algoritmus A*: okraj** implementujeme uspořádaně (pro snadné hledání minima/maxima)
  - stavy ohodnotíme pomocí *f* = *g* + *h*
    - *f* = odhad ceny cesty z **počátku do cíle**
    - *g* = cena cesty z **počátku do** daného **stavu**
    - *h* = *odhad ceny cesty z daného* ***stavu do cíle***
  - expandujeme stav s nejmenším *f*

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



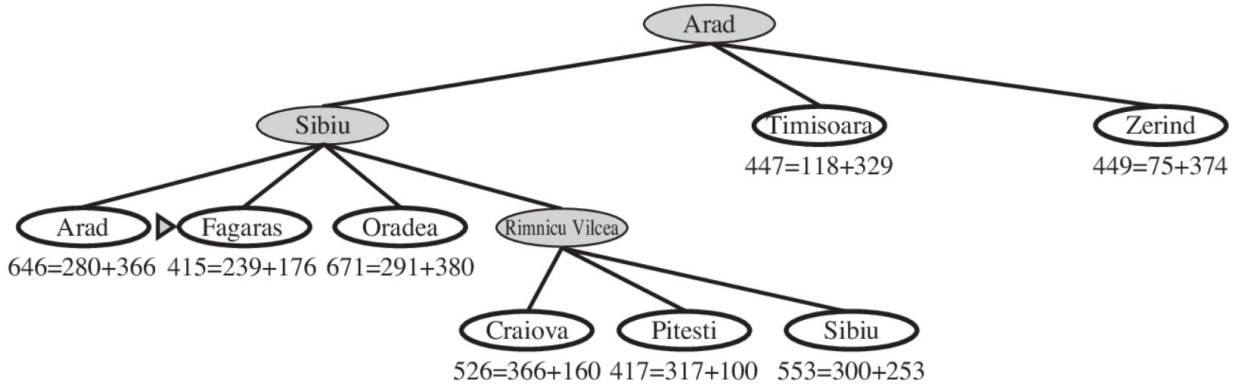
ALGORITMUS **A***

- **Vlastnosti**
  - implikuje
    - h(s) ≤ c(s,s') + h(s')
    - ⇓ **monotonie**
    - h(s) ≤ h*(s)
    - **přípustnost**
      - h* cena cesty z daného stavu do cíle
- ⇒ nalezení **optima**



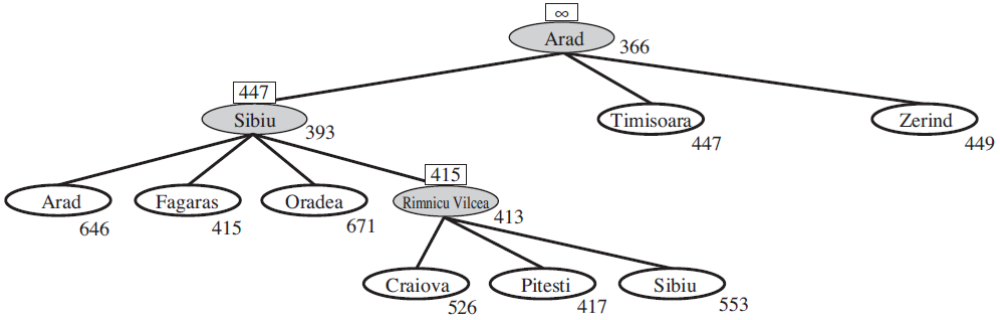
Vzdušná vzdálenost
monotonní h

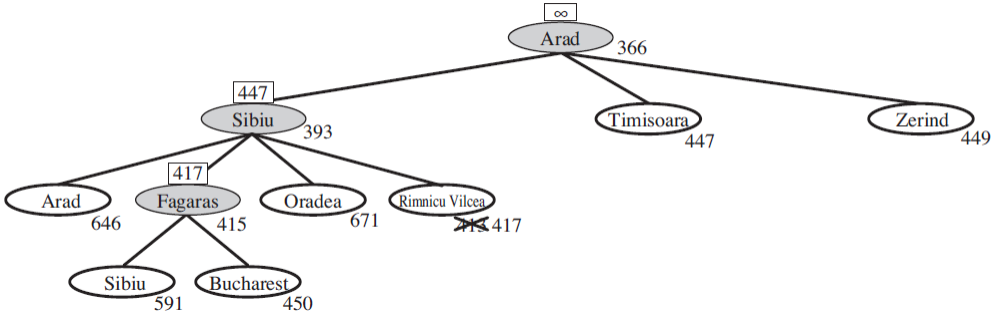
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

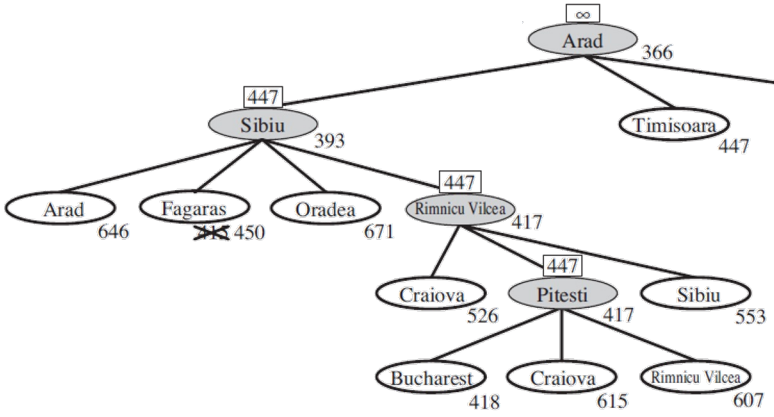


ALGORITMUS **RBFS**

- **Napodobuje A* podobně jako IDS napodobuje BFS**
  - šetří paměť, zachovává optimalitu







ZÁVĚREČNÉ POZNÁMKY

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

- **Pozor na opakující se stavy**
  - Zapamatovat
    - znovu neexpandovat
- **Rozšiřující témata**
  - Prohledávání s využitím **externí paměti** (disk, …)
  - **Téměř přípustná** heuristika
  - **Obousměrné** prohledávání (*bi-directional search*)
    - směrem od počátku i od cílového stavu najednou
    - v ideálním případě se prohledávací strom rozvine 2x do hloubky d/2, celková složitost $O(b^{d/2})$
  - Prohledávání s **částečnou informací**
    - neznámý počáteční stav
    - výsledek akce je nejistý
    - stavy a akce nejsou předem známé