

## Dekompozice problému, AND/OR grafy

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

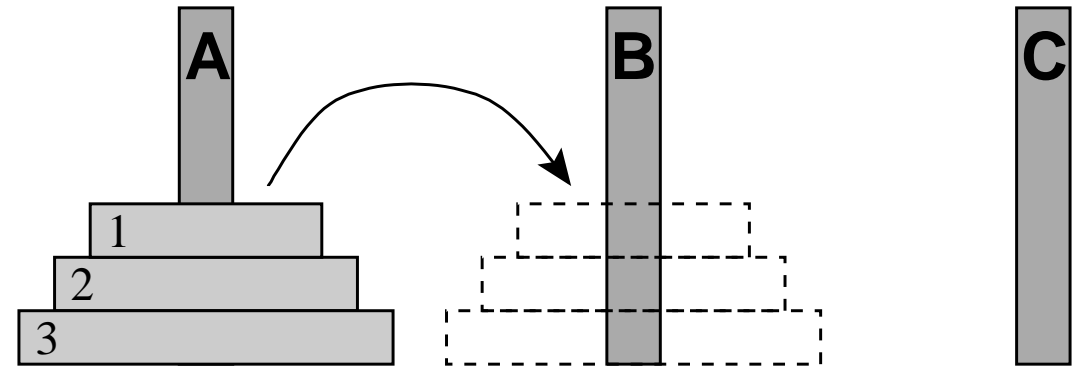
- Připomínka – průběžná písemka
- Příklad – Hanoiské věže
- AND/OR grafy
- Prohledávání AND/OR grafů

## PŘIPOMÍNKA – PRŮBĚŽNÁ PÍSEMKA

- termín – **příští týden**, **27. října**, **10:00**, **B007**, na přednášce
- náhradní termín: **není**
- příklady (formou testu – odpovědi A, B, C, D, E, z látky probrané do 24.10.):
  - uveden příklad v Prologu, otázka **Co řeší tento program?**
  - uveden příklad v Prologu a cíl, otázka **Co je (návratová) hodnota výsledku?**
  - **upravte** (doplňte/změňte řádek) uvedený **program tak, aby...**
  - uvedeno několik **tvrzení**, potvrďte jejich pravdivost/nepravdivost
  - porovnání **vlastností** několika **algoritmů**
- rozsah: **4 příklady**
- hodnocení: **max. 32 bodů** – za *správnou odpověď* 8 bodů, za *žádnou odpověď* 0 bodů, za *špatnou odpověď* -3 bodů.

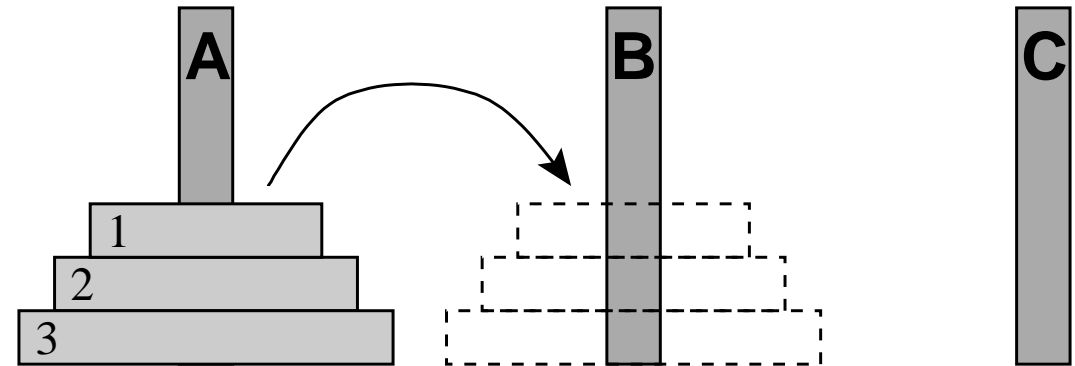
## PŘÍKLAD – HANOISKÉ VĚŽE

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti)  $n$  kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps.  $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$ ) bez porušení uspořádání



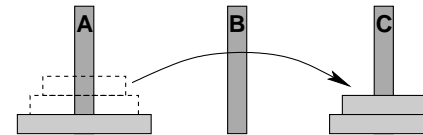
## PŘÍKLAD – HANOISKÉ VĚŽE

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti)  $n$  kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps.  $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$ ) bez porušení uspořádání



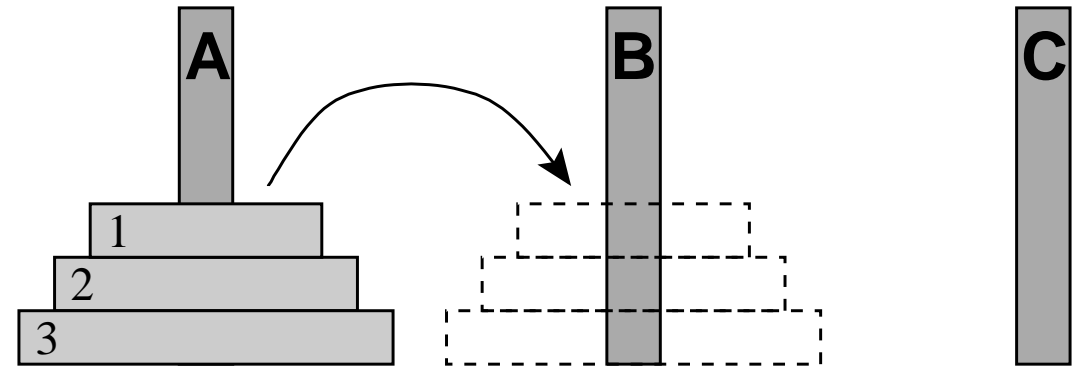
Můžeme rozložit na fáze:

1. přeskládat  $n - 1$  kotoučů z **A** pomocí **B** na **C**.



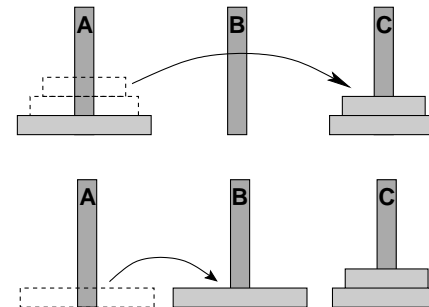
## PŘÍKLAD – HANOISKÉ VĚŽE

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti)  $n$  kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps.  $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$ ) bez porušení uspořádání



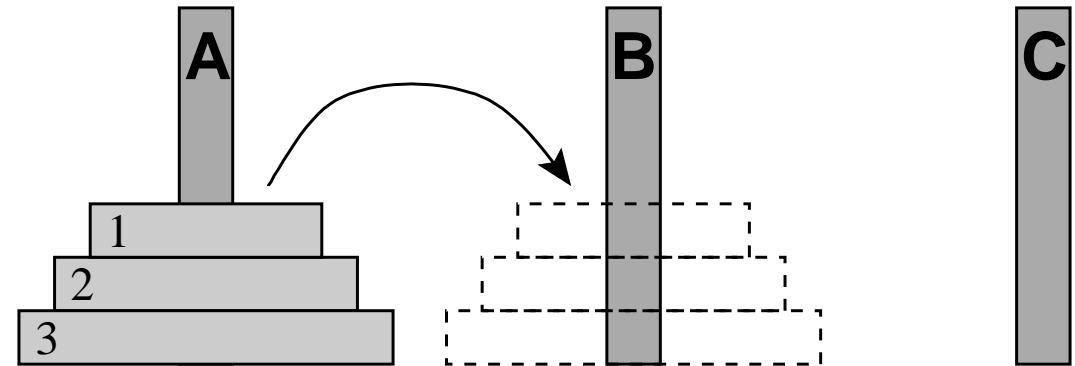
Můžeme rozložit na fáze:

1. přeskládat  $n - 1$  kotoučů z **A** pomocí **B** na **C**.
2. přeložit 1 kotouč z **A** na **B**



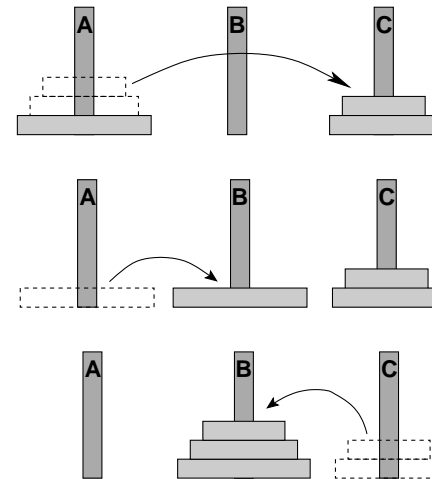
## PŘÍKLAD – HANOISKÉ VĚŽE

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti)  $n$  kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps.  $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$ ) bez porušení uspořádání



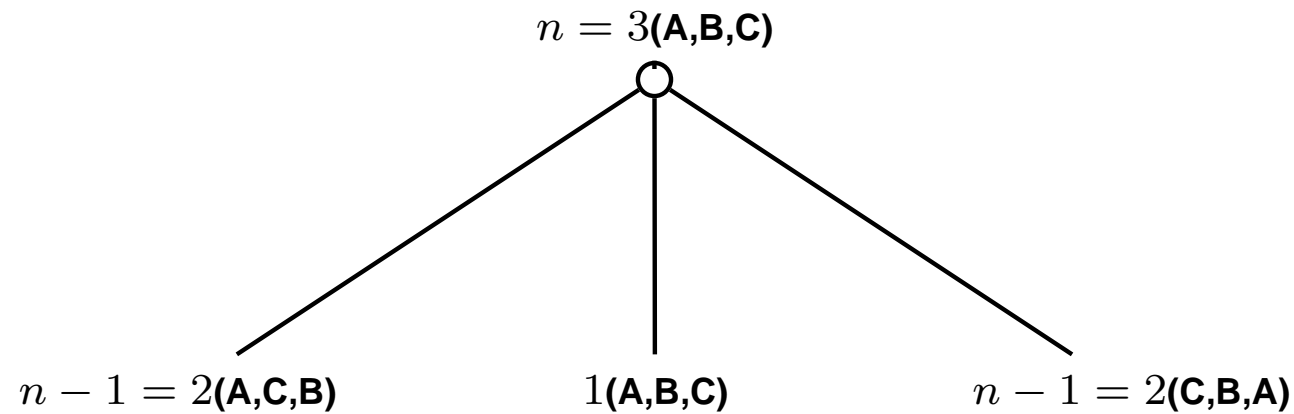
Můžeme rozložit na fáze:

1. přeskládat  $n - 1$  kotoučů z **A** pomocí **B** na **C**.
2. přeložit 1 kotouč z **A** na **B**
3. přeskládat  $n - 1$  kotoučů z **C** pomocí **A** na **B**



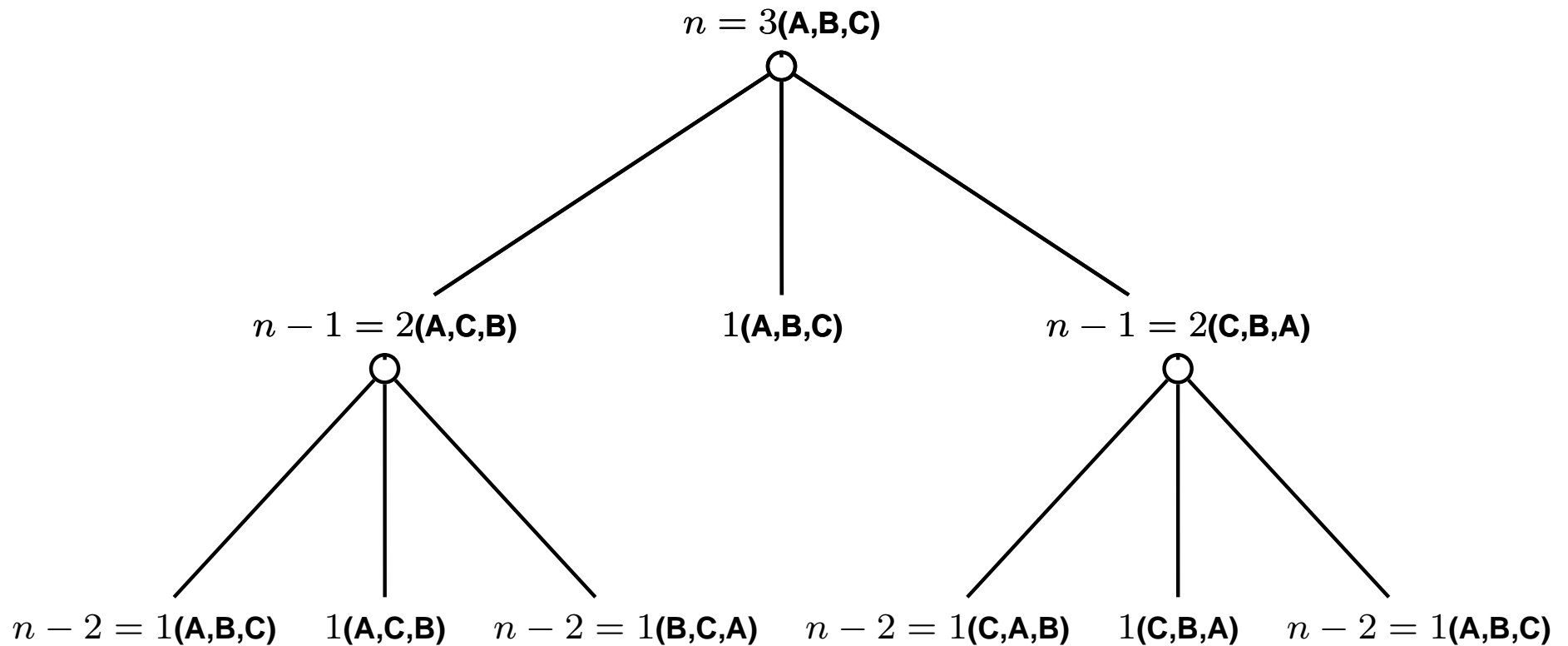
## PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

schéma celého řešení pro  $n = 3$ :



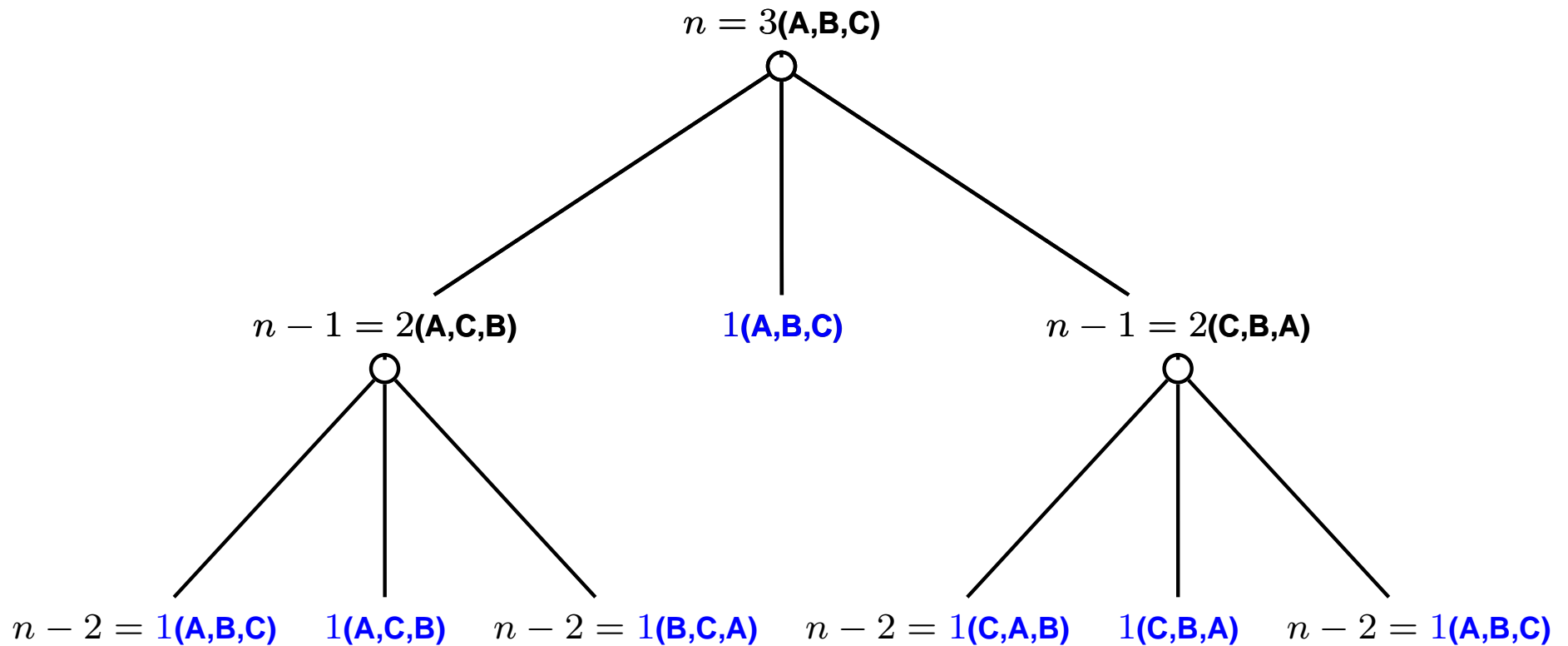
## PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

schéma celého řešení pro  $n = 3$ :





## PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

schéma celého řešení pro  $n = 3$ :

## PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

? – `op(100,xfx,to), dynamic(hanoi/5).`

`hanoi(1,A,B,C,[A to B]).`

`hanoi(N,A,B,C,Moves) :- N > 1, N1 is N - 1, lemma(hanoi(N1,A,C,B,Ms1)),  
hanoi(N1,C,B,A,Ms2), append(Ms1,[A to B|Ms2],Moves).`

`lemma(P) :- P,asserta((P :- !)).`

## PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

?– **op**(100,xfx,to), dynamic(**hanoi**/5).

**hanoi**(1,A,B,C,[A to B]).

**hanoi**(N,A,B,C,Moves) :- N>1, N1 is N-1, **lemma**(**hanoi**(N1,A,C,B,Ms1)),  
**hanoi**(N1,C,B,A,Ms2), append(Ms1,[A to B|Ms2],Moves).

**lemma**(P) :- P,**asserta**((P :- !)).

?– **hanoi**(3,a,b,c,M).

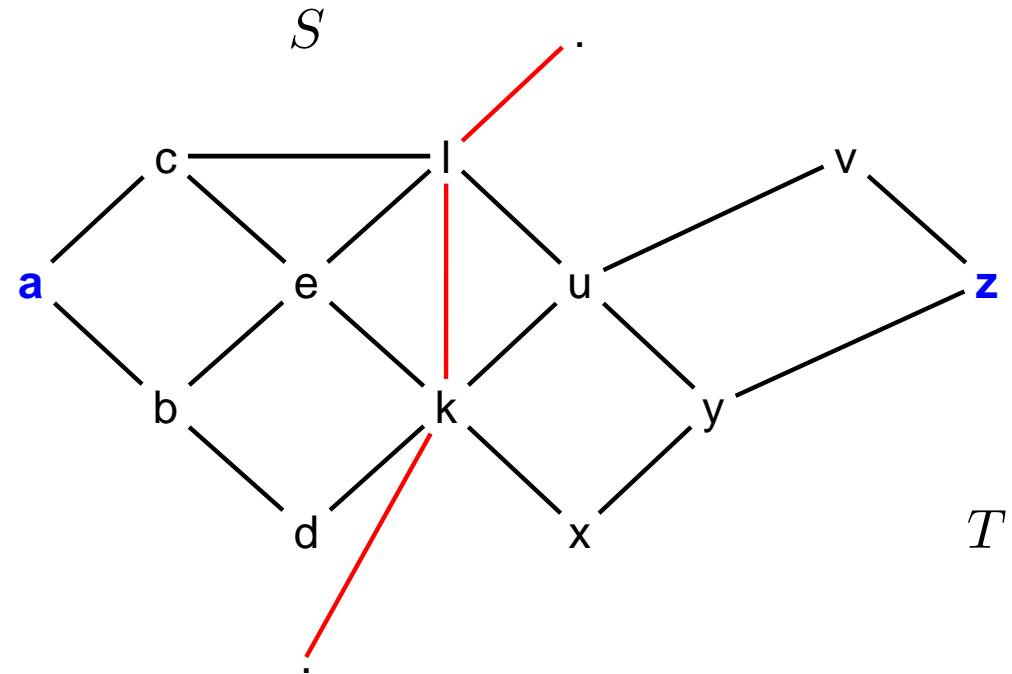
M = [a to b, a to c, b to c, a to b, c to a, c to b, a to b] ;  
No

## CESTA MEZI MĚSTY POMOCÍ AND/OR GRAFŮ

města: **a**, ..., **e** ... ve státě  $S$   
**l** a **k** ... hraniční přechody  
**u**, ..., **z** ... ve státě  $T$

hledáme cestu z **a** do **z**:

- ➔ cesta z **a** do hraničního přechodu
- ➔ cesta z hraničního přechodu do **z**



## CESTA MEZI MĚSTY POMOCÍ AND/OR GRAFŮ pokrač.

schéma řešení pomocí rozkladu na podproblémy = AND/OR graf

přímý zápis AND/OR grafu v Prologu:

**OR uzel**  $v$  s následníky  $u_1, u_2, \dots, u_N$ :

```
v :- u1.
v :- u2.
...
v :- uN.
```

**AND uzel**  $x$  s následníky  $y_1, y_2, \dots, y_M$ :

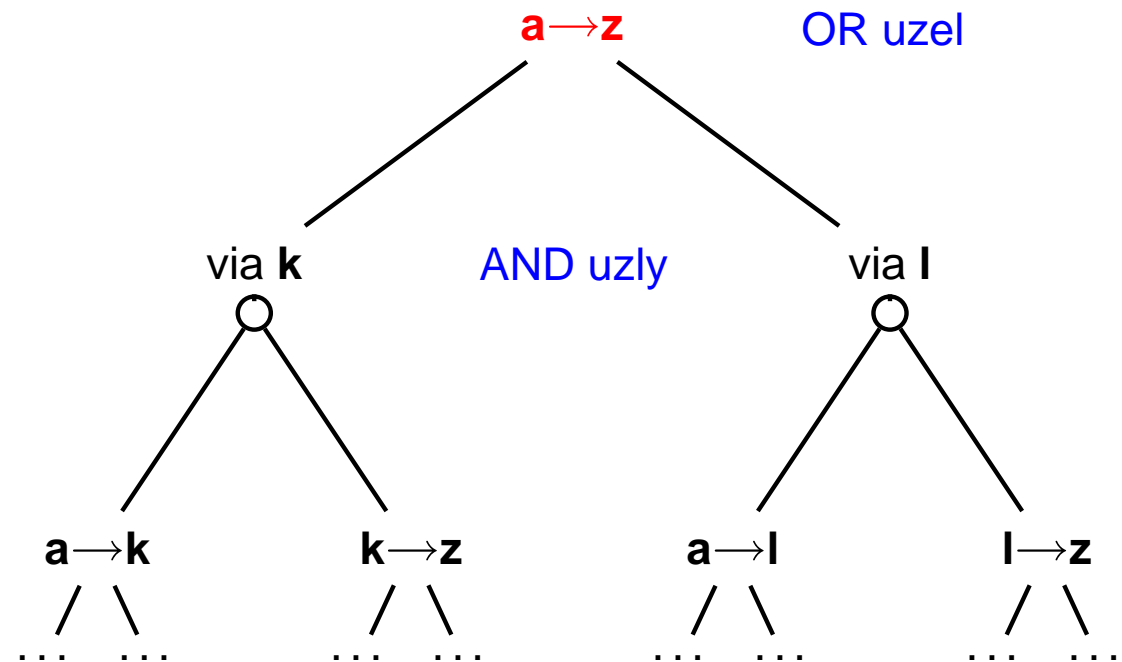
```
x :- y1, y2, ..., yM.
```

**cílový** uzel  $g$  ( $\hat{=}$  elementární problém):

```
g.
```

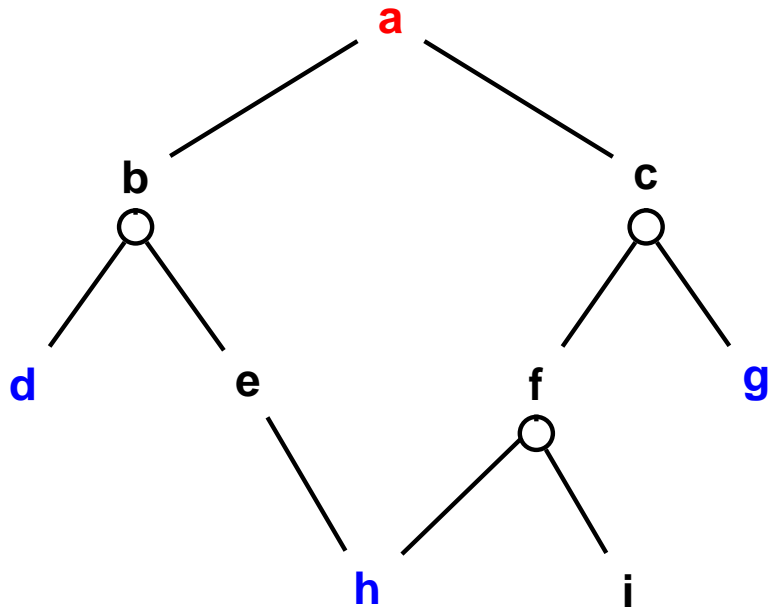
**kořenový** uzel **root**:

```
?- root.
```



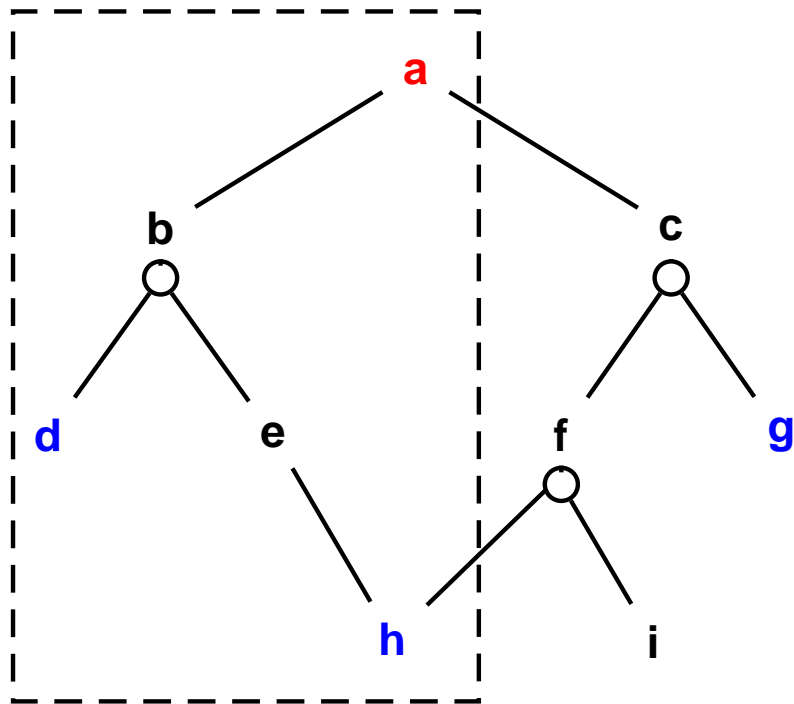
**Celkové řešení** = podgraf AND/OR grafu, který nevynechává žádného následníka AND-uzlu.

## TRIVIÁLNÍ PROHLEDÁVÁNÍ AND/OR GRAFU V PROLOGU



```
a :- b.  
a :- c.  
b :- d, e.  
e :- h.  
c :- f, g.  
f :- h, i.  
d.  
g.  
h.
```

## TRIVIÁLNÍ PROHLEDÁVÁNÍ AND/OR GRAFU V PROLOGU



```
a :- b.  
a :- c.  
b :- d, e.  
e :- h.  
c :- f, g.  
f :- h, i.  
d.  
g.  
h.  
  
?- a.  
Yes
```

## REPREZENTACE AND/OR GRAFU

AND/OR graf = graf s 2 typy vnitřních uzlů – AND uzly a OR uzly

- AND uzel jako součást řešení vyžaduje průchod všech svých poduzlů
- OR uzel se chová jako běžný uzel klasického grafu

Reprezentace AND/OR grafu v Prologu:

- zavedeme operátory '---->' a ':'

```
?- op(600, xfx, ---->).  
?- op(500, xfx, :).
```

- AND/OR graf budeme zapisovat

```
a ----> or:[b, c].  
b ----> and:[d, e].
```



## REPREZENTACE AND/OR GRAFU

AND/OR graf = graf s 2 typy vnitřních uzlů – AND uzly a OR uzly

- AND uzel jako součást řešení vyžaduje průchod všech svých poduzlů
- OR uzel se chová jako běžný uzel klasického grafu

Reprezentace AND/OR grafu v Prologu:

- zavedeme operátory '---->' a ':'

```
?- op(600, xfx, ---->).
?- op(500, xfx, :).
```

- AND/OR graf budeme zapisovat

```
a ----> or:[b, c].
b ----> and:[d, e].
```

**op(+Priorita, +Typ, +Jméno)**

**Priorita** číslo 0..1200

**Typ** jedno z *xf, yf, xfx, xfy, yfx, yfy, fy* nebo *fx*

**Jméno** *funktor* nebo *symbol*

## REPREZENTACE AND/OR GRAFU

AND/OR graf = graf s 2 typy vnitřních uzlů – AND uzly a OR uzly

- AND uzel jako součást řešení vyžaduje průchod všech svých poduzlů
- OR uzel se chová jako běžný uzel klasického grafu

Reprezentace AND/OR grafu v Prologu:

→ zavedeme operátory '---->' a ':'

→ AND/OR graf budeme zapisovat

```
?- op(600, xfx, ---->).
?- op(500, xfx, :).
```

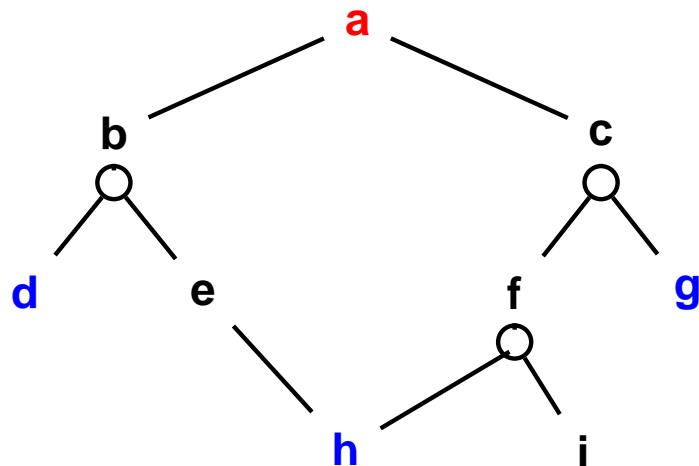
```
a ----> or:[b, c].
b ----> and:[d, e].
```

**op(+Priorita, +Typ, +Jméno)**

**Priorita** číslo 0..1200

**Typ** jedno z *xf*, *yf*, *xfx*, *xfy*,  
*yfx*, *yfy*, *fy* nebo *fx*

**Jméno** funktor nebo symbol



```
a ----> or:[b,c].
b ----> and:[d,e].
c ----> and:[f,g].
e ----> or:[h].
f ----> and:[h,i].
goal(d).
goal(g).
goal(h).
```

## STROM ŘEŠENÍ AND/OR GRAFU

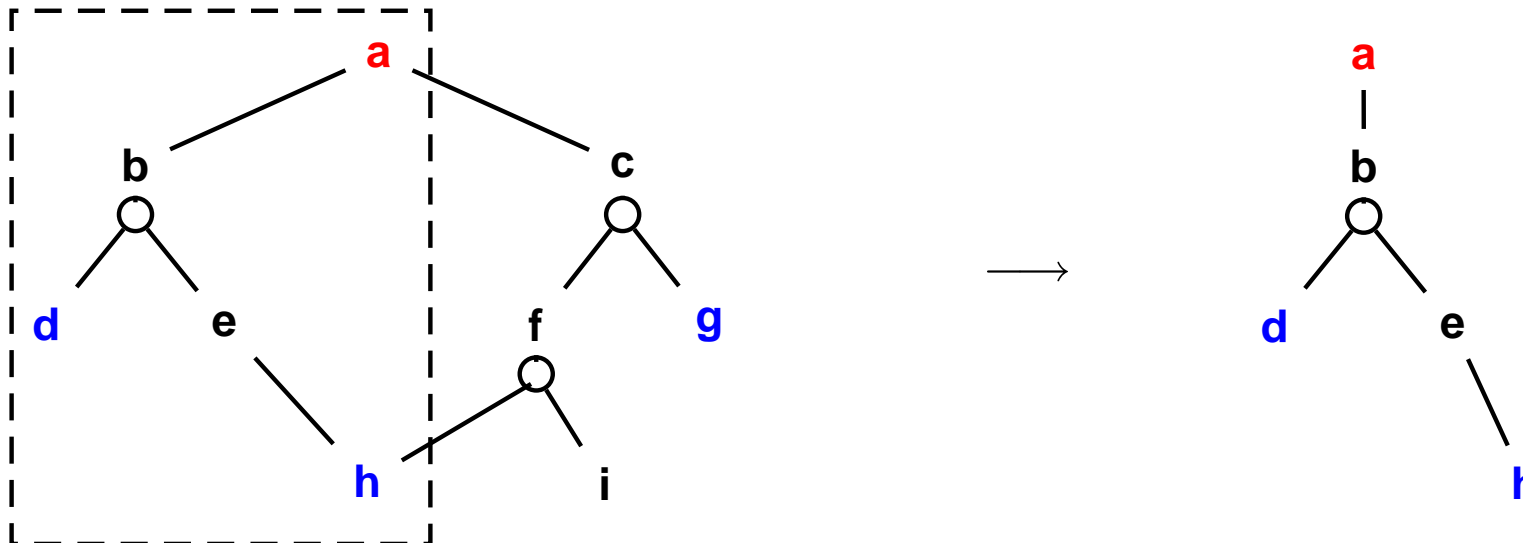
strom řešení  $T$  problému  $P$  s AND/OR grafem  $G$ :

- problém  $P$  je kořen stromu  $T$
- jestliže  $P$  je OR uzel grafu  $G \Rightarrow$  právě jeden z jeho následníků se svým stromem řešení je v  $T$
- jestliže  $P$  je AND uzel grafu  $G \Rightarrow$  všichni jeho následníci se svými stromy řešení jsou v  $T$
- každý list stromu řešení  $T$  je cílovým uzlem v  $G$

## STROM ŘEŠENÍ AND/OR GRAFU

strom řešení  $T$  problému  $P$  s AND/OR grafem  $G$ :

- problém  $P$  je kořen stromu  $T$
- jestliže  $P$  je OR uzel grafu  $G \Rightarrow$  právě jeden z jeho následníků se svým stromem řešení je v  $T$
- jestliže  $P$  je AND uzel grafu  $G \Rightarrow$  všichni jeho následníci se svými stromy řešení jsou v  $T$
- každý list stromu řešení  $T$  je cílovým uzlem v  $G$



## PROHLEDÁVÁNÍ AND/OR GRAFU DO HLOUBKY

```
% solve (+Node, - SolutionTree)
solve(Node,Node) :- goal(Node).
solve(Node,Node ----> Tree) :-
    Node ----> or:Nodes, member(Node1,Nodes), solve(Node1,Tree).
solve(Node,Node ----> and:Trees) :-
    Node ----> and:Nodes, solveall(Nodes,Trees).

% solveall ([Node1,Node2, ...], [SolutionTree1, SolutionTree2, ...])
solveall ([],[]).
solveall ([Node|Nodes],[Tree|Trees]) :- solve(Node,Tree), solveall(Nodes,Trees).
```

## PROHLEDÁVÁNÍ AND/OR GRAFU DO HLOUBKY

```

% solve (+Node, - SolutionTree )
solve(Node,Node) :- goal(Node).
solve(Node,Node ----> Tree) :-
    Node ----> or:Nodes, member(Node1,Nodes), solve(Node1,Tree).
solve(Node,Node ----> and:Trees) :-
    Node ----> and:Nodes, solveall(Nodes,Trees).

% solveall ([Node1,Node2, ...], [SolutionTree1 , SolutionTree2 , ...])
solveall ([],[]).
solveall ([Node|Nodes],[Tree|Trees]) :- solve(Node,Tree), solveall(Nodes,Trees).

?- solve(a,Tree).
Tree = a----> (b---->and:[d, e---->h]) ;
No
    
```

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU

→ doplnění reprezentace o **cenu přechodové hrany** (=odhad složitosti podproblému):

Uzel  $\dashrightarrow$  AndOr:[NaslUzel1/Cena1, NaslUzel2/Cena2, ..., NaslUzelN/CenaN].

→ definujeme **cenu uzlu** jako cenu optimálního řešení jeho podstromu

→ pro každý uzel  $N$  máme daný odhad jeho ceny:

$h(N)$  = heuristický odhad ceny optimálního podgrafu s kořenem  $N$

→ pro každý uzel  $N$ , jeho následníky  $N_1, \dots, N_b$  a jeho předchůdce  $M$  definujeme:

$$F(N) = \text{cena}(M, N) + \begin{cases} h(N), & \text{pro ještě neexpandovaný uzel } N \\ 0, & \text{pro cílový uzel (elementární problém)} \\ \min_i (F(N_i)), & \text{pro OR-uzel } N \\ \sum_i F(N_i), & \text{pro AND-uzel } N \end{cases}$$

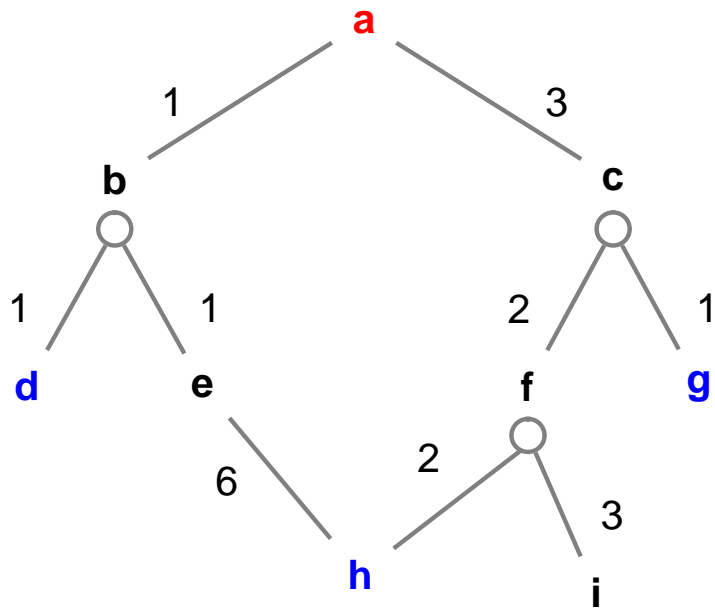
Pro optimální strom řešení  $S$  je tedy  $F(S)$  právě cena tohoto řešení (=suma  $\forall$  hran z  $S$ ).

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$



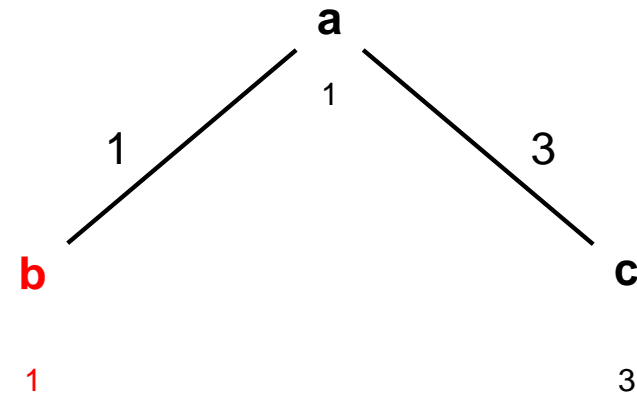
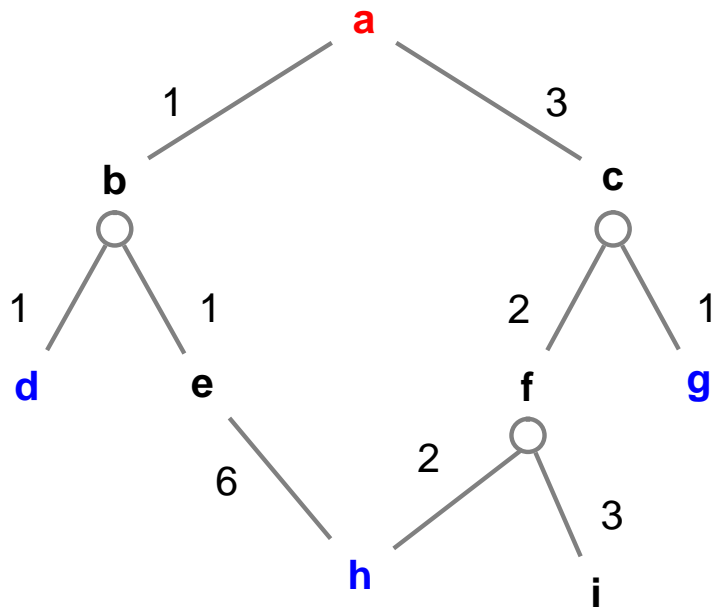


## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

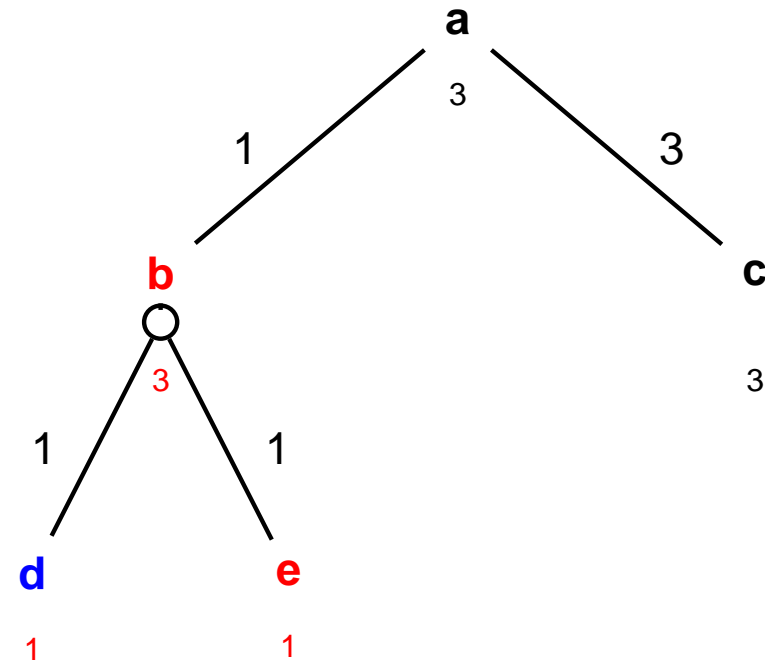
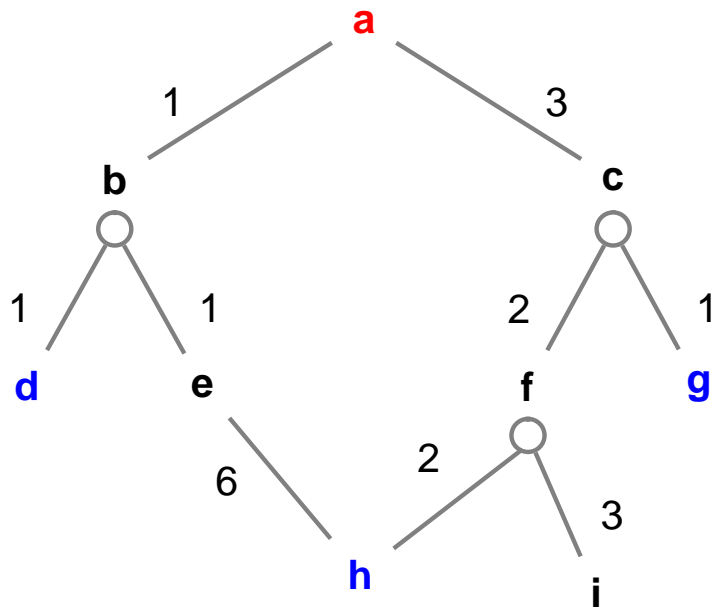


## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

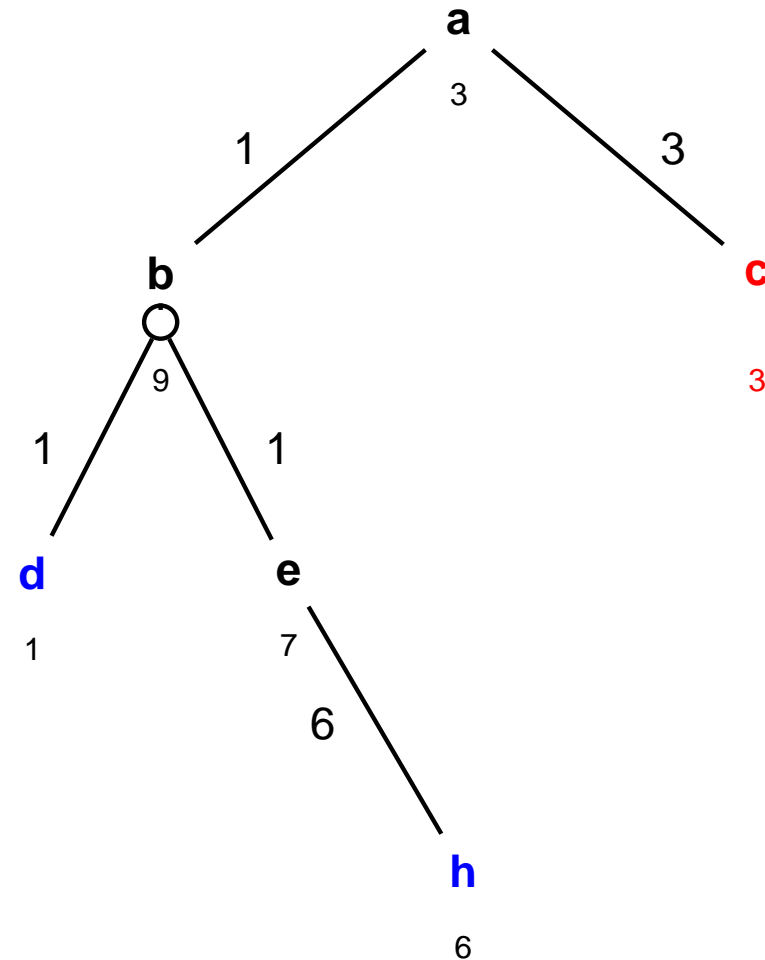
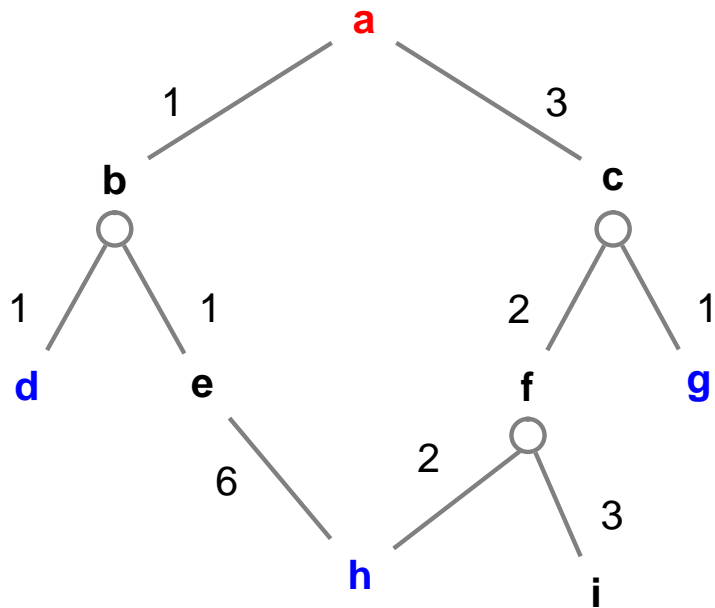


## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

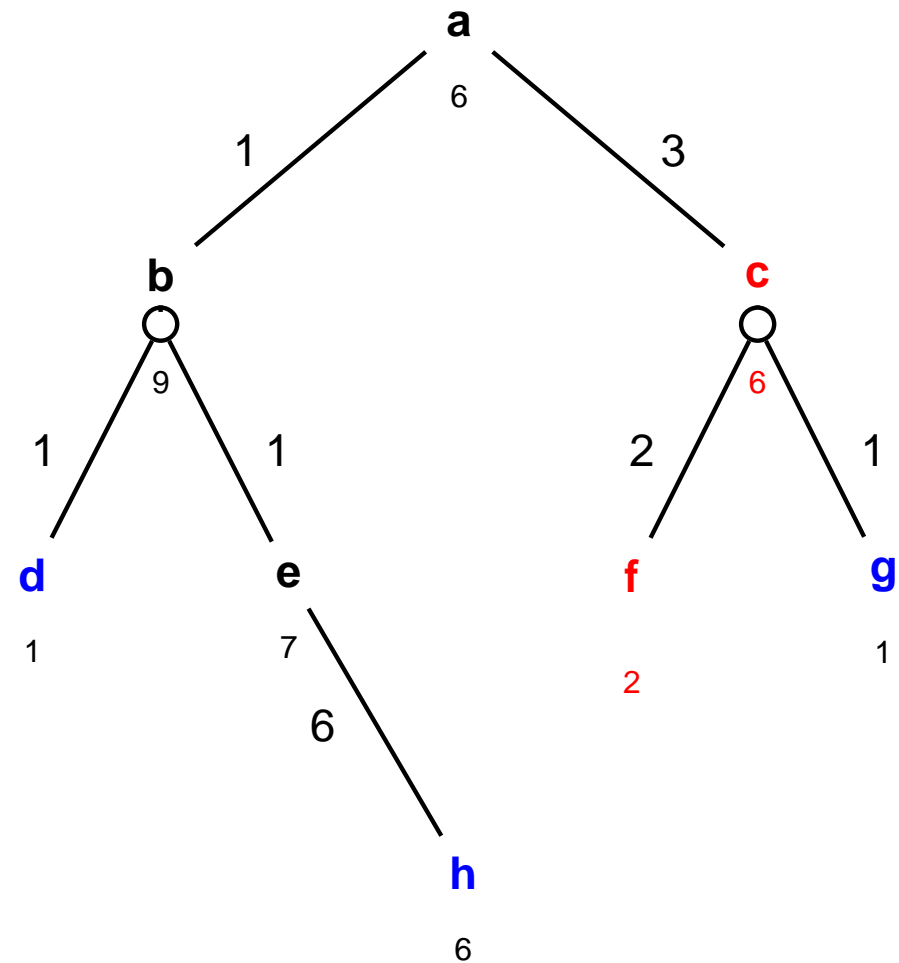
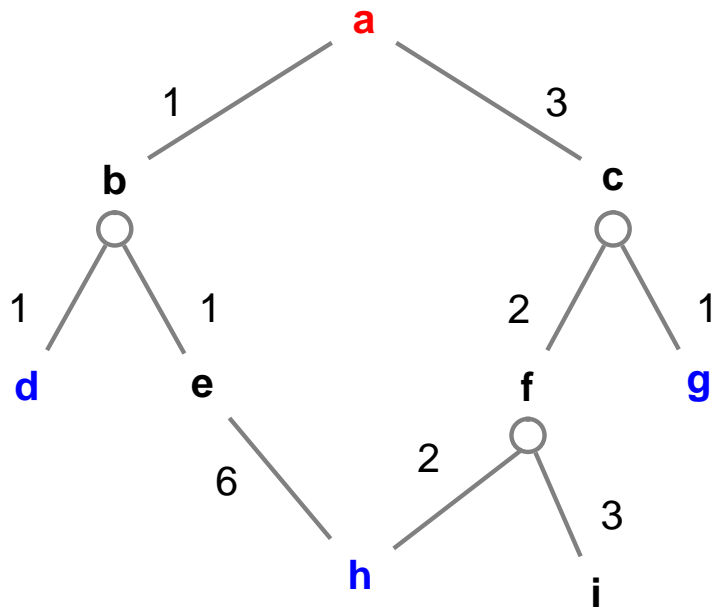


## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

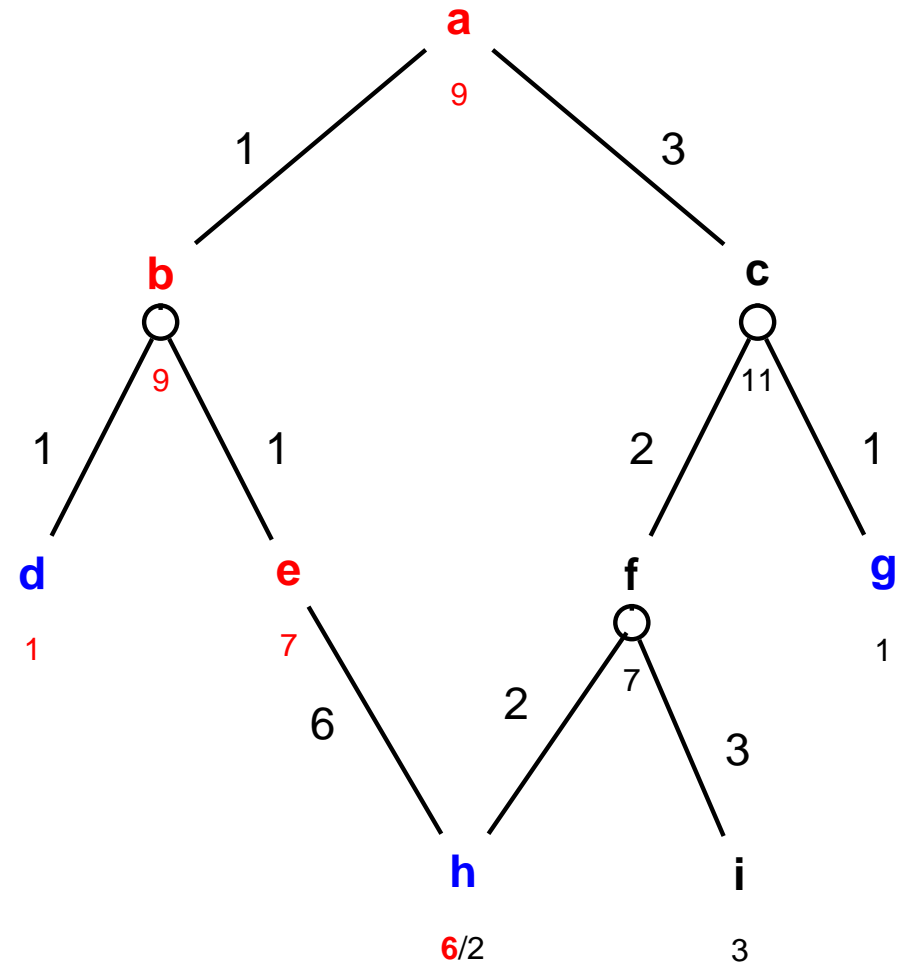
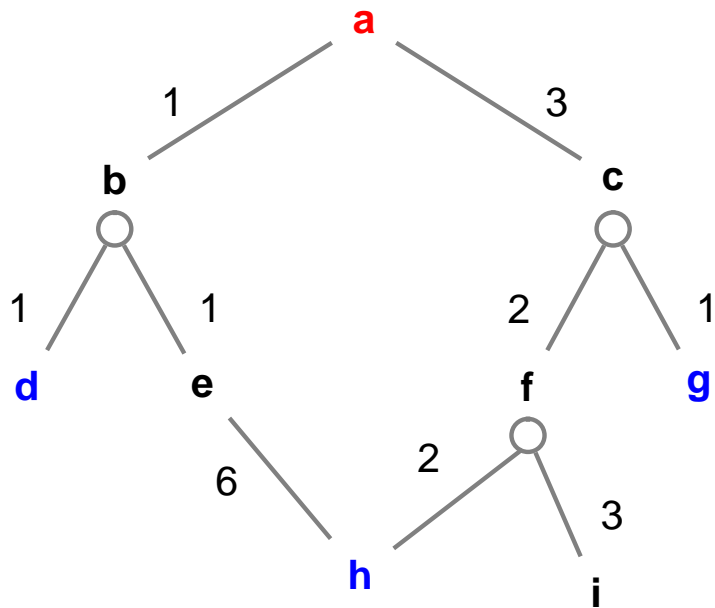


## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU – PŘÍKLAD

seříděný seznam částečně expandovaných grafů =

[Nevyřešený<sub>1</sub>, Nevyřešený<sub>2</sub>, ..., Vyřešený<sub>1</sub>, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$



## REPREZENTACE AND/OR GRAFU PŘI HEURISTICKÉM PROHLEDÁVÁNÍ

**list** AND/OR grafu ... struktura **leaf(N,F,C)**.

$$F = C + h(N)$$

**OR uzel** AND/OR grafu ... struktura **tree(N,F,C,or:[T1,T2,T3,...])**.

$$F = C + \min_i F_i$$

**AND uzel** AND/OR grafu ... struktura **tree(N,F,C,and:[T1,T2,T3,...])**.

$$F = C + \sum_i F_i$$

**vyřešený list** AND/OR grafu ... struktura **solvedleaf(N,F)**.

$$F = C$$

**vyřešený OR uzel** AND/OR grafu ... struktura **solvedtree(N,F,T)**.

$$F = C + F_1$$

**vyřešený AND uzel** AND/OR grafu ... struktura **solvedtree(N,F,and:[T1,T2,...])**.

$$F = C + \sum_i F_i$$

**C** ... cena hrany do uzlu **N**

**F** ... příslušná heuristická **F**-hodnota uzlu **N**

**N** ... identifikátor uzlu

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU

```
andor(Node,SolutionTree) :- biggest(Bound),expand(leaf(Node,0,0),Bound,SolutionTree,yes).
```

```
expand(Tree,Bound,Tree,no) :- f(Tree,F),F > Bound,!. 
```

```
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
```

```
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
  (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
```

```
expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved) :- Bound1 is Bound-C,
  expandlist(SubTrees,Bound1,NewSubs,Solved1),
  continue(Solved1,Node,C,NewSubs,Bound,NewTree,Solved).
```

```
expandlist(Trees,Bound,NewTrees,Solved) :-
  selecttree(Trees,Tree,OtherTrees,Bound,Bound1),
  expand(Tree,Bound1,NewTree,Solved1),
  combine(OtherTrees,NewTree,Solved1,NewTrees,Solved).
```

```
continue(yes,Node,C,SubTrees,_,solvedtree(Node,F,SubTrees),yes) :-
  bestf(SubTrees,H), F is C+H,!
```

```
continue(never,_,_,_,_,_,never) :- !.
```

```
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- bestf(SubTrees,H),
  F is C+H,!,expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved).
```

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU

```
andor(Node,SolutionTree) :- biggest(Bound),expand(leaf(Node,0,0),Bound,SolutionTree,yes).
```

```
% 1: limit Bound překročen (ve všech dalších klauzulích platí  $F \leq Bound$ )
```

```
expand(Tree,Bound,Tree,no) :- f(Tree,F),F > Bound,!. ←
```

```
% 2: nalezen cíl
```

```
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
```

```
% 3: expanze listu
```

```
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
    (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
```

```
% 4: expanze stromu
```

```
expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved) :- Bound1 is Bound-C,
    expandlist(SubTrees,Bound1,NewSubs,Solved1),
    continue(Solved1,Node,C,NewSubs,Bound,NewTree,Solved).
```

```
expandlist(Trees,Bound,NewTrees,Solved) :-
```

```
    selecttree (Trees,Tree,OtherTrees,Bound,Bound1),
```

```
    expand(Tree,Bound1,NewTree,Solved1),
```

```
    combine(OtherTrees,NewTree,Solved1,NewTrees,Solved).
```

```
continue(yes,Node,C,SubTrees,_,solvedtree(Node,F,SubTrees),yes) :-
```

```
    bestf(SubTrees,H), F is C+H,!
```

```
continue(never,_,_,_,_,_,never) :- !.
```

```
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- bestf(SubTrees,H),
    F is C+H,!,expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved).
```

```
expand(+Tree, +Bound,
-NewTree, ?Solved)
expanduje Tree po Bound.
Výsledek je NewTree se
stavem Solved
```



## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU

```
andor(Node,SolutionTree) :- biggest(Bound),expand(leaf(Node,0,0),Bound,SolutionTree,yes).
```

```
% 1: limit Bound překročen (ve všech dalších klauzulích platí  $F \leq Bound$ )
```

```
expand(Tree,Bound,Tree,no) :- f(Tree,F),F > Bound,!.
```

```
% 2: nalezen cíl
```

```
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
```

```
% 3: expanze listu
```

```
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
    (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
```

```
% 4: expanze stromu
```

```
expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved) :- Bound1 is Bound-C,
    expandlist(SubTrees,Bound1,NewSubs,Solved1),
    continue(Solved1,Node,C,NewSubs,Bound,NewTree,Solved).
```

```
expandlist(Trees,Bound,NewTrees,Solved) :-
    selecttree(Trees,Tree,OtherTrees,Bound,Bound1),
    expand(Tree,Bound1,NewTree,Solved1),
    combine(OtherTrees,NewTree,Solved1,NewTrees,Solved).
```

```
continue(yes,Node,C,SubTrees,_,solvedtree(Node,F,SubTrees),yes) :-
    bestf(SubTrees,H), F is C+H,!.
```

```
continue(never,_,_,_,_,_,never) :- !.
```

```
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- bestf(SubTrees,H),
    F is C+H,! ,expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved).
```

**expand(+Tree, +Bound, -NewTree, ?Solved)**  
 expanduje Tree po Bound.  
 Výsledek je NewTree se stavem Solved

**expandlist** expanduje všechny grafy v seznamu Trees se závorou Bound.  
 Výsledek je v seznamu NewTrees a celkový stav v Solved

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU

```
andor(Node,SolutionTree) :- biggest(Bound),expand(leaf(Node,0,0),Bound,SolutionTree,yes).
```

```
% 1: limit Bound překročen (ve všech dalších klauzulích platí  $F \leq Bound$ )
```

```
expand(Tree,Bound,Tree,no) :- f(Tree,F),F > Bound,!.
```

```
% 2: nalezen cíl
```

```
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
```

```
% 3: expanze listu
```

```
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
    (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
```

```
% 4: expanze stromu
```

```
expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved) :- Bound1 is Bound-C,
    expandlist(SubTrees,Bound1,NewSubs,Solved1),
    continue(Solved1,Node,C,NewSubs,Bound,NewTree,Solved).
```

```
expandlist(Trees,Bound,NewTrees,Solved) :-
    selecttree(Trees,Tree,OtherTrees,Bound,Bound1),
    expand(Tree,Bound1,NewTree,Solved1),
    combine(OtherTrees,NewTree,Solved1,NewTrees,Solved).
```

```
continue(yes,Node,C,SubTrees,_,solvedtree(Node,F,SubTrees),yes) :-
    bestf(SubTrees,H), F is C+H,!
```

```
continue(never,_,_,_,_,_,never) :- !.
```

```
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- bestf(SubTrees,H),
    F is C+H,!,expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved).
```

**expand(+Tree, +Bound, -NewTree, ?Solved)**  
 expanduje Tree po Bound.  
 Výsledek je NewTree se stavem Solved

**expandlist** expanduje všechny grafy v seznamu Trees se závorou Bound.  
 Výsledek je v seznamu NewTrees a celkový stav v Solved

**continue** určuje, jak pokračovat po expanzi seznamu grafů

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```

combine(or:_,Tree,yes,Tree,yes) :- !.
combine(or:Trees,Tree,no,or:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
combine(or:[],_,never,_,never) :- !.
combine(or:Trees,_,never,or:Trees,no) :- !.
combine(and:Trees,Tree,yes,and:[Tree|Trees],yes) :- allsolved(Trees),!.
combine(and:_,_,never,_,never) :- !.
combine(and:Trees,Tree,YesNo,and:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.

expandnode(Node,C,tree(Node,F,C,Op:SubTrees)) :- Node ----> Op:Successors,
    expandsucc(Successors,SubTrees),bestf(Op:SubTrees,H),F is C+H.
expandsucc ([],[]).
expandsucc([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,expandsucc(NodesCosts,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

allsolved ([]).
allsolved ([Tree|Trees]) :- solved(Tree),allsolved(Trees).

solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
    
```

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```

combine(or:_,Tree,yes,Tree,yes) :- !. ←
combine(or:Trees,Tree,no,or:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
combine(or:[],_,never,_,never) :- !.
combine(or:Trees,_,never,or:Trees,no) :- !.
combine(and:Trees,Tree,yes,and:[Tree|Trees],yes) :- allsolved(Trees),!.
combine(and:_,_,never,_,never) :- !.
combine(and:Trees,Tree,YesNo,and:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.

expandnode(Node,C,tree(Node,F,C,Op:SubTrees)) :- Node ---> Op:Successors,
    expandsucc(Successors,SubTrees),bestf(Op:SubTrees,H),F is C+H.
expandsucc([],[]).
expandsucc([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,expandsucc(NodesCosts,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

allsolved([]).
allsolved([Tree|Trees]) :- solved(Tree),allsolved(Trees).

solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
    
```

**combine(OtherTrees,NewTree,  
Solved1,NewTrees,Solved)**  
*kombinuje výsledky expanze  
stromu a seznamu stromů*

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```

combine(or:_,Tree,yes,Tree,yes) :- !. ←
combine(or:Trees,Tree,no,or:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
combine(or:[],_,never,_,never) :- !.
combine(or:Trees,_,never,or:Trees,no) :- !.
combine(and:Trees,Tree,yes,and:[Tree|Trees],yes) :- allsolved(Trees),!.
combine(and:_,_,never,_,never) :- !.
combine(and:Trees,Tree,YesNo,and:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.

expandnode(Node,C,tree(Node,F,C,Op:SubTrees)) :- Node ----> Op:Successors,
    expandsucc(Successors,SubTrees,bestf(Op:SubTrees,H),F is C+H).
expandsucc([],[]).
expandsucc([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,expandsucc(NodesCosts,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

allsolved([]).
allsolved([Tree|Trees]) :- solved(Tree),allsolved(Trees).

solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
    
```

**combine(OtherTrees,NewTree,  
Solved1,NewTrees,Solved)**  
kombinuje výsledky expanze  
stromu a seznamu stromů

**expandnode** převede uzel z  
**Node** → **AndOr:S** do  
**tree(Node,F,C,SS)**



## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```

combine(or:_,Tree,yes,Tree,yes) :- !. ←
combine(or:Trees,Tree,no,or:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
combine(or:[],_,never,_,never) :- !.
combine(or:Trees,_,never,or:Trees,no) :- !.
combine(and:Trees,Tree,yes,and:[Tree|Trees],yes) :- allsolved(Trees),!.
combine(and:_,_,never,_,never) :- !.
combine(and:Trees,Tree,YesNo,and:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.

expandnode(Node,C,tree(Node,F,C,Op:SubTrees)) :- Node ----> Op:Successors,
    expandsucc(Successors,SubTrees),bestf(Op:SubTrees,H),F is C+H.
expandsucc([],[]).
expandsucc([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,expandsucc(NodesCosts,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

allsolved([]).
allsolved([Tree|Trees]) :- solved(Tree),allsolved(Trees). ←

solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
    
```

**combine(OtherTrees,NewTree, Solved1,NewTrees,Solved)**  
*kombinuje výsledky expanze stromu a seznamu stromů*

**expandnode** převede uzel z **Node** → **AndOr:S** do **tree(Node,F,C,SS)**

**allsolved** zkontroluje, jestli všechny stromy v seznamu jsou vyřešené

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```

f(Tree,F) :- arg(2,Tree,F),!.

insert(T,[],[T]) :- !.
insert(T,[T1|Ts],[T,T1|Ts]) :- solved(T1),!.
insert(T,[T1|Ts],[T1|Ts1]) :- solved(T),insert(T,Ts,Ts1),!.
insert(T,[T1|Ts],[T,T1|Ts]) :- f(T,F),f(T1,F1),F=<F1,!.
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).

% první následovník v OR-uzlu je nejlepší
bestf(or:[Tree|_],F) :- f(Tree,F),!.
bestf(and:[],0) :- !.
bestf(and:[Tree1|Trees],F) :- f(Tree1,F1),bestf(and:Trees,F2),F is F1+F2,!.
bestf(Tree,F) :- f(Tree,F).

selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !. % The only candidate
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- bestf(Op:Trees,F),
  (Op=or,!,min(Bound,F,Bound1);Op=and,Bound1 is Bound-F).

min(A,B,A) :- A<B,!.
min(A,B,B).

```

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```
f(Tree,F) :- arg(2,Tree,F),!.
```

```
insert(T,[],[T]) :- !.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- solved(T1),!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- solved(T),insert(T,Ts,Ts1),!.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- f(T,F),f(T1,F1),F=<F1,!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).
```

```
% první následovník v OR-uzlu je nejlepší
```

```
bestf(or:[Tree|_],F) :- f(Tree,F),!.
```

```
bestf(and:[],0) :- !.
```

```
bestf(and:[Tree1|Trees],F) :- f(Tree1,F1),bestf(and:Trees,F2),F is F1+F2,!.
```

```
bestf(Tree,F) :- f(Tree,F).
```

```
selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !. % The only candidate
```

```
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- bestf(Op:Trees,F),  
(Op=or,! ,min(Bound,F,Bound1);Op=and,Bound1 is Bound-F).
```

```
min(A,B,A) :- A<B,!.
```

```
min(A,B,B).
```

insert vkládá strom do seznamu stromů se zachováním třídění



## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```
f(Tree,F) :- arg(2,Tree,F),!.
```

```
insert(T,[],[T]) :- !.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- solved(T1),!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- solved(T),insert(T,Ts,Ts1),!.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- f(T,F),f(T1,F1),F=<F1,!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).
```

```
% první následovník v OR-uzlu je nejlepší
```

```
bestf(or:[Tree|_],F) :- f(Tree,F),!.
```

```
bestf(and:[],0) :- !.
```

```
bestf(and:[Tree1|Trees],F) :- f(Tree1,F1),bestf(and:Trees,F2),F is F1+F2,!.
```

```
bestf(Tree,F) :- f(Tree,F).
```

```
selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !. % The only candidate
```

```
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- bestf(Op:Trees,F),  
(Op=or,! ,min(Bound,F,Bound1);Op=and,Bound1 is Bound-F).
```

```
min(A,B,A) :- A<B,!.
```

```
min(A,B,B).
```

**insert** vkládá strom do seznamu stromů se zachováním třídění

**bestf** vyhledá uloženou  $F$ -hodnotu AND/OR stromu/uzlu

## HEURISTICKÉ PROHLEDÁVÁNÍ AND/OR GRAFU pokrač.

```
f(Tree,F) :- arg(2,Tree,F),!.
```

```
insert(T,[],[T]) :- !.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- solved(T1),!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- solved(T),insert(T,Ts,Ts1),!.
```

```
insert(T,[T1|Ts],[T,T1|Ts]) :- f(T,F),f(T1,F1),F=<F1,!.
```

```
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).
```

*% první následovník v OR-uzlu je nejlepší*

```
bestf(or:[Tree|_],F) :- f(Tree,F),!.
```

```
bestf(and:[],0) :- !.
```

```
bestf(and:[Tree1|Trees],F) :- f(Tree1,F1),bestf(and:Trees,F2),F is F1+F2,!.
```

```
bestf(Tree,F) :- f(Tree,F).
```

```
selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !. % The only candidate
```

```
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- bestf(Op:Trees,F),  
(Op=or,! ,min(Bound,F,Bound1);Op=and,Bound1 is Bound-F).
```

```
min(A,B,A) :- A<B,!.
```

```
min(A,B,B).
```

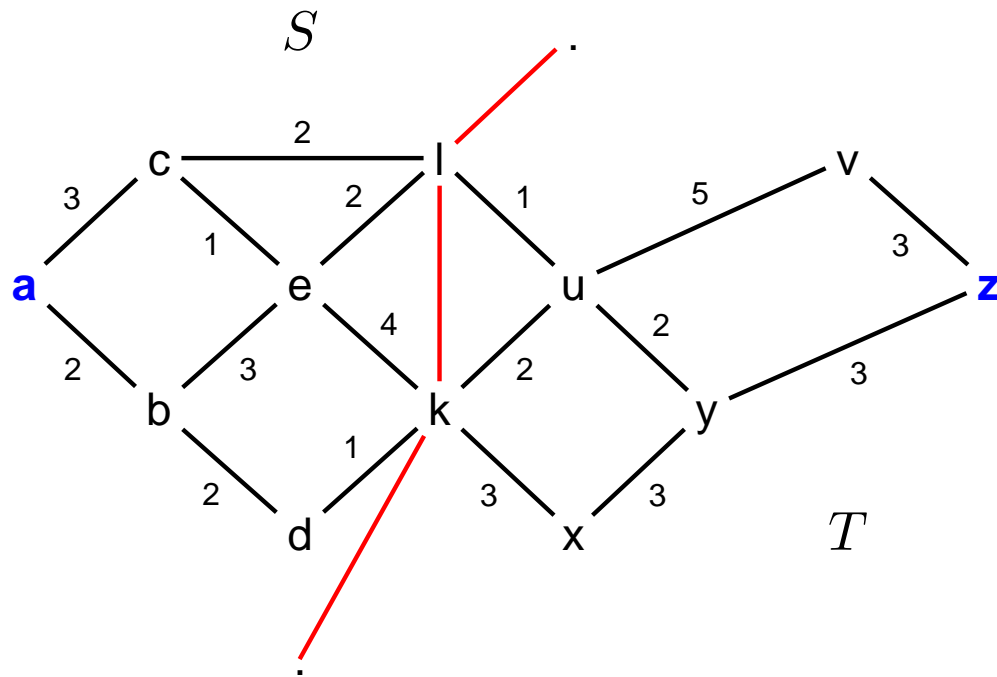
**insert** vkládá strom do seznamu stromů se zachováním třídění

**bestf** vyhledá uloženou  $F$ -hodnotu AND/OR stromu/uzlu

**selecttree** (+Trees, -BestTree, -OtherTrees, +Bound, -Bound1) vybere BestTree z Trees, zbytek je v OtherTrees. Bound je závora pro Trees, Bound1 pro BestTree

## CESTA MEZI MĚSTY HEURISTICKÝM AND/OR HLEDÁNÍM

- cesta mezi **Mesto1** a **Mesto2** – predikát **move(Mesto1,Mesto2,Vzdal)**.
- klíčové postavení města **Mesto3** – predikát **key(Mesto1–Mesto2,Mesto3)**.



```

move(a,b,2). move(a,c,3). move(b,e,3).
move(b,d,2). move(c,e,1). move(c,l,2).
move(e,k,4). move(e,l,2). move(k,u,2).
move(k,x,3). move(u,v,5). move(x,y,3).
move(y,z,3). move(v,z,3). move(l,u,1).
move(d,k,1). move(u,y,2).
    
```

```

stateS(a). stateS(b). stateS(c). stateS(d). stateS(e).
stateT(u). stateT(v). stateT(x). stateT(y). stateT(z).
border(l). border(k).
    
```

```

key(M1–M2,M3) :- stateS(M1), stateT(M2), border(M3).
    
```

```

city (X) :- (stateS(X);stateT(X);border(X)).
    
```

## CESTA MEZI MĚSTY HEURISTICKÝM AND/OR HLEDÁNÍM pokrač.

- vlastní hledání cesty:
1. **Y1, Y2,...** klíčové body mezi městy **A** a **Z**. Hledej jednu z cest:
    - cestu z **A** do **Z** přes **Y1**
    - cestu z **A** do **Z** přes **Y2**
    - ...
  2. Není-li mezi městy **A** a **Z** klíčové město  $\Rightarrow$  hledej souseda **Y** města **A** takového, že existuje cesta z **Y** do **Z**.

## CESTA MEZI MĚSTY HEURISTICKÝM AND/OR HLEDÁNÍM pokrač.

Konstrukce příslušného AND/OR grafu:

? – **op**(560,xfx,via).    % *operátory* *X–Z* a *X–Z via Y*

a–z ----> **or**:[a–z via k/0,a–z via l/0]

a–v ----> **or**:[a–v via k/0,a–v via l/0]

...

a–l ----> **or**:[c–l/3,b–l/2]

b–l ----> **or**:[e–l/3,d–l/2]

...

a–z via l ----> **and**:[a–l/0,l–z/0]

a–v via l ----> **and**:[a–l/0,l–v/0]

...

**goal**(a–a). **goal**(b–b). ...

## CESTA MEZI MĚSTY HEURISTICKÝM AND/OR HLEDÁNÍM pokrač.

Konstrukce příslušného AND/OR grafu:

? – **op**(560,xfx,via).    % *operátory*  $X-Z$  a  $X-Z$  *via*  $Y$

$a-z$  ----> **or**:[ $a-z$  via  $k/0$ , $a-z$  via  $l/0$ ]

$a-v$  ----> **or**:[ $a-v$  via  $k/0$ , $a-v$  via  $l/0$ ]

...

$a-l$  ----> **or**:[ $c-l/3$ , $b-l/2$ ]

$b-l$  ----> **or**:[ $e-l/3$ , $d-l/2$ ]

...

$a-z$  via  $l$  ----> **and**:[ $a-l/0$ , $l-z/0$ ]

$a-v$  via  $l$  ----> **and**:[ $a-l/0$ , $l-v/0$ ]

...

**goal**( $a-a$ ). **goal**( $b-b$ ). ...

$X-Z$  ----> **or**:**Problemlist** :- city( $X$ ),city( $Z$ ), bagof(( $X-Z$  via  $Y$ )/0, **key**( $X-Z$ , $Y$ ), **Problemlist**),!.

$X-Z$  ----> **or**:**Problemlist** :- city( $X$ ),city( $Z$ ), bagof(( $Y-Z$ )/ $D$ , **move**( $X$ , $Y$ , $D$ ), **Problemlist**).

$X-Z$  via  $Y$  ----> **and**:[( $X-Y$ )/0,( $Y-Z$ )/0]:- city( $X$ ),city( $Z$ ),**key**( $X-Z$ , $Y$ ).

**goal**( $X-X$ ).

*/\* h(Node,H). ... heuristická funkce \*/*

Když  $\forall n : h(n) \leq h^*(n)$ , kde  $h^*$  je minimální cena řešení uzlu  $n \Rightarrow$  najdeme **vždy optimální řešení**