

Dekompozice problému, AND/OR grafy

Aleš Horák

E-mail: hales@fi.muni.cz
<http://nlp.fi.muni.cz/uui/>

Obsah:

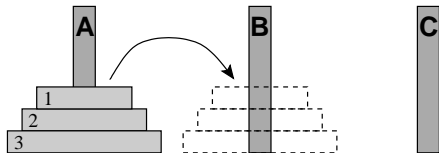
- ▶ Připomínka – průběžná písemka
- ▶ AND/OR grafy
- ▶ Prohledávání AND/OR grafů

Připomínka – průběžná písemka

- ▶ termín – **příští přednášku, 24. října, 10:00, A318**, na začátku přednášky
- ▶ náhradní termín: **není**
- ▶ příklady (formou testu – odpovědi A, B, C, D, E, z látky probrané na prvních pěti přednáškách, včetně dnešní):
 - uveden příklad v Prologu+Pythonu, otázka **Co řeší tento program?**
 - uveden příklad v Prologu+Pythonu a cíl/volání programu, otázka **Co je (návratová) hodnota výsledku?**
 - **upravte** (vyberte úpravu/doplnění) uvedený **program tak, aby...**
 - uvedeno několik **tvrzení**, potvrďte jejich pravdivost/nepřavdivost
 - 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 body.

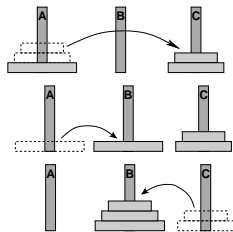
Příklad – Hanoiské věže

- ▶ 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



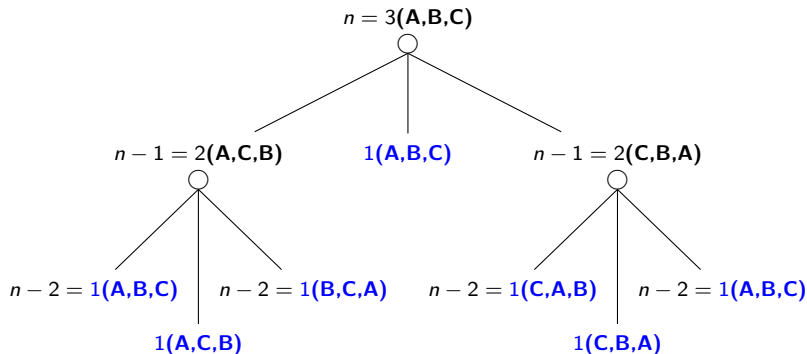
Můžeme rozložit na fáze:

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.

op(+Priorita, +Typ, +Jméno)

Priorita číslo 0..1200

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

Jméno funktor nebo symbol

?–**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 dekompozice

města:

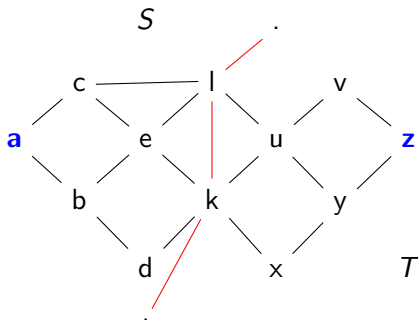
a, ..., **e** ... ve státě S

l a **k** ... hraniční přechody

u, ..., **z** ... ve státě T

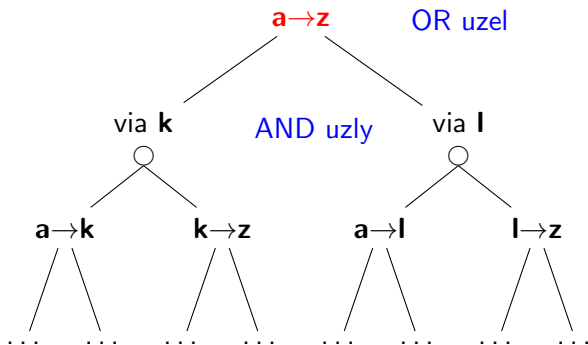
hledáme cestu z **a** do **z**:

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



Cesta mezi městy pomocí dekompozice – pokrač.

schéma řešení pomocí rozkladu na podproblémy = AND/OR graf

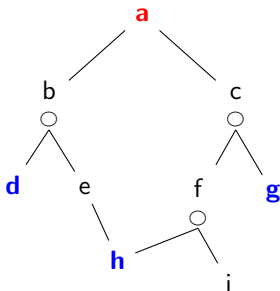


Celkové řešení = podgraf AND/OR grafu, který nevynechává žádného následníka AND-uzlu.

AND/OR graf a strom řešení

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

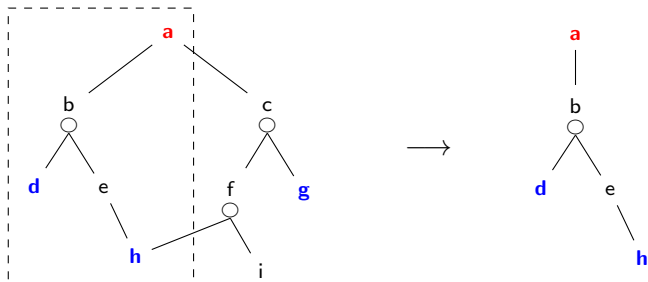
- ▶ *AND uzel* jako součást řešení vyžaduje průchod všech svých poduzlů
- ▶ *OR uzel* se chová jako běžný uzel klasického grafu



AND/OR graf a strom řešení

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

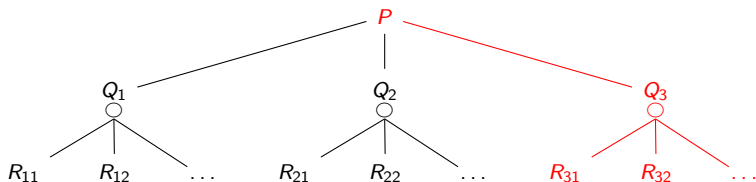


Příklad – výherní strategie

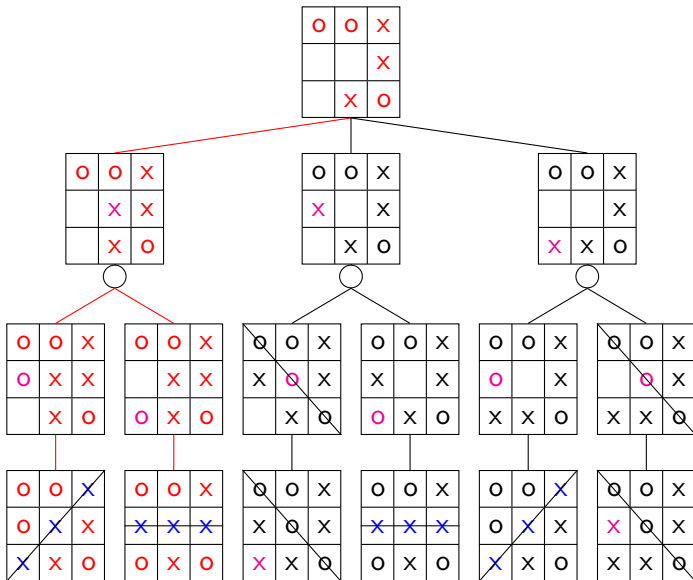
Hra 2 hráčů s perfektními znalostmi, 2 výstupy $\left\{ \begin{array}{l} \text{výhra} \\ \text{prohra} \end{array} \right.$

Výherní strategii je možné formulovat jako **AND/OR graf**:

- ▶ počáteční stav P typu *já-jsem-na-tahu*
- ▶ moje tahy vedou do stavů Q_1, Q_2, \dots typu *soupeř-je-na-tahu*
- ▶ následně soupeřovy tahy vedou do stavů R_{11}, R_{12}, \dots *já-jsem-na-tahu*
- ▶ cíl – stav, který je **výhra** podle pravidel (*prohra* je neřešitelný problém)
- ▶ stav P *já-jsem-na-tahu* je **výherní** \Leftrightarrow **některý** z Q_i je výherní, **OR**
- ▶ stav Q_i *soupeř-je-na-tahu* je **výherní** \Leftrightarrow **všechny** R_{ij} jsou výherní, **AND**
- ▶ **výherní strategie** = řešení AND/OR grafu



Příklad – výherní strategie



Reprezentace AND/OR grafu

přímý zápis AND/OR grafu v Prologu:

► OR uzel **v** s následníky **u1**, **u2**, ..., **uN**:

v :- **u1**.

v :- **u2**.

...

v :- **uN**.

► AND uzel **x** s následníky **y1**, **y2**, ..., **yM**:

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

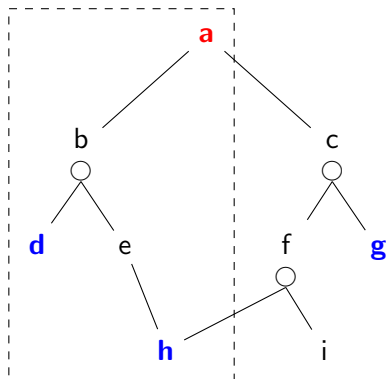
► cílový uzel **g** ($\hat{=}$ elementární problém):

g.

► kořenový uzel **root**:

?- **root**.

Triviální prohledávání AND/OR grafu v Prologu



a :- b.

a :- c.

b :- d, e.

e :- h.

c :- f, g.

f :- h, i.

d.

g.

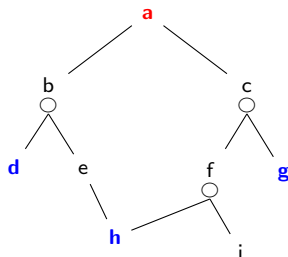
h.

?- a.

Yes

Reprezentace AND/OR grafu v Prologu

- ▶ zavedeme operátory '--->' a ':'
`?- op(700, xfx, --->).`
`?- op(500, xfx, :).`
- ▶ AND/OR graf budeme zapisovat
`a ---> or:[b, c].`
`b ---> and:[d, e].`



```

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).
  
```

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 (AO*)

- ▶ doplnění reprezentace o **cenu přechodové hrany** (=míra složitosti podproblému):

Uzel \rightarrow 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:

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

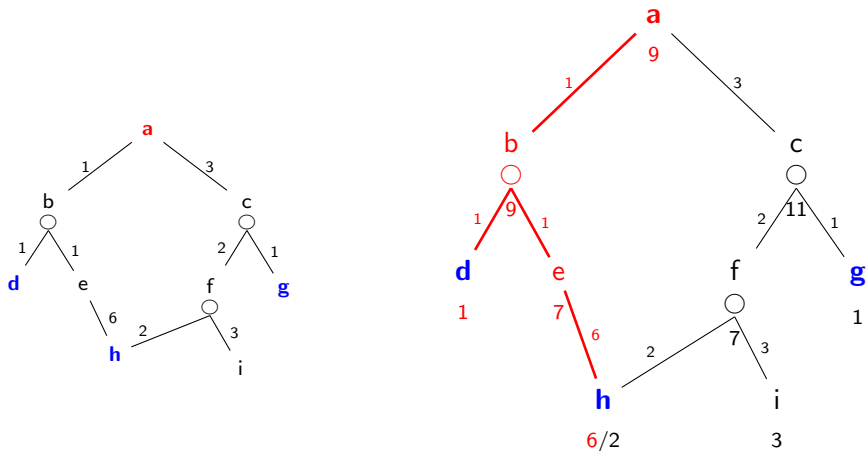
Pro optimální strom řešení S je tedy $F(S)$ právě cena tohoto řešení (=suma všech hran z S).

Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných grafů =

[Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

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



Reprezentace AND/OR grafu při heuristickém prohledávání

F ... příslušná heuristická F -hodnota uzlu N

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

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

C ... cena hrany do uzlu N

N ... identifikátor uzlu

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

$$F = C + \min_j F_j$$

- ▶ **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 ... **solvedtree(N,F,and:[T1,T2,...])**

$$F = C + \sum_i F_i$$

Python – ("typ uzlu", n, f, ...):

("leaf",n,f,c), ("tree",n,f,c,("or",subtrees)), ...

Heuristické prohledávání AND/OR grafu (AO*)

```
def andor(node):
```

```
    sol, solved = expand(("leaf", node, 0, 0), biggest)
```

```
    if solved == "yes": return sol
```

```
    else: raise ValueError("Reseni neexistuje.")
```

expand(tree, bound) → (newtree, solved)

expanduje tree po bound. Výsledek je newtree se stavem solved.

```
def expand(tree, bound):
```

```
    if f(tree) > bound: return (tree, "no")
```

```
    tree_type = tree[0]
```

```
    if tree_type == "leaf":
```

```
        _, node, f_, c = tree
```

```
        if is_goal(node): return (("solved_leaf", node, f_), "yes")
```

```
        tree1 = expandnode(node, c)
```

```
        if tree1 is None: return (None, "never") # neexistuji naslednici
```

```
        return expand(tree1, bound)
```

```
    elif tree_type == "tree":
```

```
        _, node, f_, c, subtrees = tree
```

```
        newsubs, solved1 = expandlist(subtrees, bound-c)
```

```
        return continue_(solved1, node, c, newsubs, bound)
```

expandlist → (newtrees, solved)

expanduje všechny grafy v seznamu trees se závorkou bound. Výsledek je v seznamu newtrees a celkový stav v solved

```
def expandlist(trees, bound):
```

```
    tree, othertrees, bound1 = select_tree(trees, bound)
```

```
    newtree, solved = expand(tree, bound1)
```

```
    return combine((newtree, solved), othertrees)
```

Heuristické prohledávání AND/OR grafu (AO*) – pokrač.

```
def continue_(subtr_solved, node, c, subtrees, bound):
    if subtr_solved == "never": return (None, "never")
    h_ = bestf(subtrees)
    f_ = c + h_
    if subtr_solved == "yes": return (("solved_tree", node, f_, subtrees), "yes")
    if subtr_solved == "no": return expand(("tree", node, f_, c, subtrees), bound)
```

continue → (solution, solved)
určuje, jak pokračovat po expanzi seznamu grafů

```
def combine(subtrees, tree, solved):
    op, trees = subtrees
    if op == "or":
        if solved == "yes": return (("or_result", tree), "yes")
        if solved == "no":
            newtrees = insert(tree, trees)
            return (("or", newtrees), "no")
        if solved == "never":
            if trees == Nil: return (None, "never")
            return (("or", trees), "no")
    if op == "and":
        if solved == "yes" and are_all_solved(trees):
            return (("and_result", Cons(tree, trees)), "yes")
        if solved == "never": return (None, "never")
        newtrees = insert(tree, trees)
        return (("and", newtrees), "no")
```

combine(othertrees, newtree, solved) → (newtrees, solved)
kombinuje výsledky expanze stromu a seznamu stromů

Heuristické prohledávání AND/OR grafu (AO*) – pokrač.

def expandnode(node, c):

```

succ = get_successors(node) # podle zadaného AND/OR grafu
if succ is None: return None
op, successors = succ
subtrees = evaluate(successors)
f_ = c + bestf((op, subtrees)) # c + best h
return ("tree", node, f_, c, (op, subtrees))

```

expandnode převede uzel z ("leaf", node, f, c) do ("tree", node, f, c, subtrees)

def evaluate(nodes):

```

if nodes == Nil: return Nil
node, c = nodes.head
f_ = c + h(node)
trees1 = evaluate(nodes.tail)
trees = insert(("leaf", node, f_, c), trees1)
return trees

```

evaluate vypočítá hodnoty pro seznam následovníků

def are_all_solved(trees):

```

if trees == Nil: return True
return is_solved(trees.head) and are_all_solved(trees.tail)

```

are_all_solved zkontroluje, jestli všechny stromy v seznamu jsou vyřešené

def is_solved(tree):

```

tree_type = tree[0]

```

```

return tree_type == "leaf" or tree_type == "tree" and is_solved(tree[1])

```

Heuristické prohledávání AND/OR grafu (AO*) – pokrač.

insert vkládá strom do seznamu stromů se zachováním třídění

```
def insert(t, trees):
    if trees == Nil: return Cons(t, Nil)
    t1 = trees.head
    ts = trees.tail
    if is_solved(t1): return Cons(t, trees)
    if is_solved(t): return Cons(t1, insert(t, ts))
    if f(t) <= f(t1): return Cons(t, trees)
    return Cons(t1, insert(t, ts))
```

select_tree(trees, bound) → (besttree, (op, othertrees), bound1)
vybere **besttree** z **trees**, zbytek je v **othertrees**. **bound** je závora pro **trees**, **bound1** pro **besttree**

```
def select_tree(subtrees, bound):
    op, trees = subtrees
    if trees.tail == Nil: return (trees.head, (op, Nil), bound)
    f_ = bestf((op, trees.tail))
    if op == "or": bound1 = min(bound, f_)
    if op == "and": bound1 = bound - f_
    return (trees.head, (op, trees.tail), bound1)
```

Heuristické prohledávání AND/OR grafu (AO*) – pokrač.

```
def f(tree):
```

```
    return tree[2]
```

```
def bestf(subtrees):
```

```
    op = subtrees[0]
```

```
    if op == "or":
```

```
        trees = subtrees[1]
```

```
        return f(trees.head)
```

```
    if op == "and" or op == "and_result":
```

```
        trees = subtrees[1]
```

```
        if trees == Nil: return 0
```

```
        return f(trees.head) + bestf(("and", trees.tail))
```

```
    if op == "or_result":
```

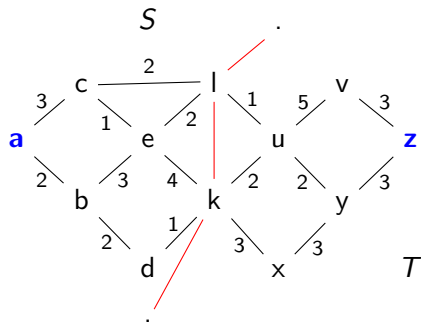
```
        tree = subtrees[1]
```

```
        return f(tree)
```

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

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

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

Konstrukce příslušného AND/OR grafu:

“pravidlová” definice 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 $l/0$]

% ...

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

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

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

% ...

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

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

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

% ...

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

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

goal($X-X$).

Cesta mezi městy heuristickým AND/OR hledáním – pokrač.

jednoduchá heuristika $h(X - Z \mid X - Z \text{ via } Y)$:

- ▶ stejné město: $h = 0$ (cíl, elementární problém)
- ▶ hrana mezi X a Y $\text{move}(X, Y, C)$: $h = C$
- ▶ jinak, stejný stát: $h = 1$
- ▶ jinak, různý stát: $h = 2$

jiná možnost – *vzdušná vzdálenost*

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

Cesta mezi městy heuristickým AND/OR hledáním – pokrač.

```

:- andor(a-z,SolutionTree), write(SolutionTree).
   solvedtree(a-z,11,
     solvedtree(a-z via l,11,
       and:[
         solvedtree(l-z,6,solvedtree(u-z,6,solvedtree(y-z,5,solvedleaf(z-z,3)))),
         solvedtree(a-l,5,solvedtree(c-l,5,solvedleaf(l-l,2))))])
  
```

