# 7. Implementace základních prohledávacích algoritmů v jazyku PROLOG

## **PROLOG** – PROgramming in LOGic

*Syntax jazyka*:

**atom**: posloupnost alfanumerických znaků začínající malým písmenem

**konstanta**: číslo (integer / real) | atom

**proměnná**: posloupnost alfanumerických znaků začínající velkým písmenem nebo znakem _

**term**: konstanta | proměnná | seznam | řetěz | predikát

**argumenty**: term [, argumenty]

**seznam**: [ ] | [argumenty] | [argumenty | proměnná] | [argumenty | seznam]

**řetěz**: posloupnost znaků uzavřená apostrofy (nebo znaky $)

Pozn.: Posloupnost znaků uzavřená uvozovkami je ekvivalentní seznamu ordinálních čísel těchto znaků.

**predikát:** jméno(argumenty) | [term] operátor [term] | jméno
**jméno:** atom
**operátor:** definovaná posloupnost znaků
**predikáty**: [not] predikát [, predikáty] | [not] predikát [; predikáty]
**pravidlo**: hlava :- tělo.
**hlava**: predikát
**tělo**: predikáty
**fakt**: predikát.
**klauzule**: fakt | pravidlo
**databáze**: množina klauzulí
**otázka/cíl**: predikáty.
**poznámka**: % libovolný řetěz znaků na zbytku řádku

Pozn.: Pomocné symboly (kulaté a hranaté závorky, čárka, středník, tečka) nejsou v předchozím výpisu syntaxe uvedeny samostatně, ale přímo na místech, kde se mohou používat. Mezery jsou nevýznamné.

Interpret PROLOGu vyzve uživatele dvojicí znaků ?- k zadání cíle/otázky. Pak odpoví slovy *Yes* nebo *True*, nebo dvojicí znaků –>, pokud zadaný cíl splní (najde kladnou odpověď na zadanou otázku), v opačném případě odpoví slovy *No* nebo *False*. Pokud PROLOG odpoví znaky –>, je možné plnění cíle buď ukončit (stiskem klávesy ENTER), nebo vyvolat další plnění cíle (stiskem znaku ;).

**Je nutné si uvědomit, že nesplnění cíle neznamená jeho negaci, ale pouze skutečnost, že z dané databáze nelze zadaný cíl splnit/dokázat !!!**

**Postup při plnění cíle:**

1. *Cílem je predikát*:

   PROLOG se snaží ztotožnit cíl s faktem nebo s hlavou pravidla (stejné jméno, stejný počet argumentů a stejné argumenty, volná proměnná se může vázat na libovolný term).

   Cíl je splněn, jestliže se ztotožní:

   a) s faktem,
   b) s hlavou pravidla a současně lze splnit tělo pravidla (tělo pravidla je chápáno jako nový cíl, databáze se prohledává vždy od počátku).

2. *Cílem je konjunkce predikátů*

Jestliže je cílem konjunkce predikátů, snaží se PROLOG postupně (zleva doprava) splnit všechny tyto predikáty – každý představuje nový cíl a vázané proměnné jsou globální.

Při prvním pokusu o plnění každého cíle $C_i$ (prochází se zleva doprava) se databáze prochází od počátku:

- Pokud se cíl $C_i$ nesplní, vrátí se PROLOG k předcházejícímu cíli $C_{i-1}$ a pokouší se tento cíl splnit jiným způsobem (pokračuje v prohledávání databáze od aktuálního místa pro tento cíl).
- Pokud se cíl $C_{i-1}$ splní, přechází PROLOG opět k cíli $C_i$ (databázi pro tento cíl začne procházet znovu od začátku).
- Pokud se cíl $C_{i-1}$ nesplní, vrací se PROLOG k předcházejícímu cíli $C_{i-2}$, neuspěje-li cíl $C_{i-2}$, vrací se PROLOG k cíli $C_{i-3}$ atd. Jde o typickou aplikaci metody zpětného navracení – Backtracking.

Pokud jsou splněny všechny cíle, je původní cíl daný konjunkcí predikátů splněn, v opačném případě je nesplněn.

3. *Cílem je disjunkce predikátů*

   Jestliže je cílem disjunkce predikátů, snaží se PROLOG postupně (zleva doprava) splnit některý z nich. Databázi prochází pro každý predikát od počátku a proměnné jsou pro každý predikát lokální.

1. příklad

**********

```
girl(susan).
girl(ann).
boy(john).
boy(george).
boy(bob).
pair(B, G) :- boy(B), girl(G).
```

**********

```
?- boy(bob).
Yes
?- boy(james).
No
?- pair(john, ann).
Yes
?- pair(ann, john).
No
```

```
?- boy(X).
X = john ->;
X = george ->;
X = bob ->;
No
?- boy(X).
X = john ->
Yes
?- girl(Girl).
Girl = susan ->
Yes
?- pair(X,Y).
X = john
Y = susan ->;
```

```
X = john
Y = ann ->;
X = george
Y = susan ->;
X = george
Y = ann ->;
X = bob
Y = susan ->;
X = bob
Y = ann ->;
No
 ?-
```

2. příklad

**********

likes(Everybody, juice).
likes(john, beer).

**********

?- likes(mary, juice).
Yes
?- likes(john, X).
X = juice -> ;
X = beer ->
Yes

?- likes(X, Y).
X = _0038
Y = juice -> ;
X = john
Y = beer -> ;
No
?- not likes(ann, beer).
Yes
?-

3. příklad

**********

```
likes(kate, roastbeef).
likes(kate, budgie).
likes(jane, juice).
likes(jack, jane).
likes(john, kate).
```

**********

```
?- likes(X, Y), likes(Y, budgie).
X = john
Y = kate -> ;
No
?- likes(X, Y); likes(Y, budgie).
X = kate
Y = roastbeef -> ;
X = kate
Y = budgie ->;
X = jane
Y = juice ->;
X = jack
Y = jane ->;
X = john
Y = kate ->;
X = _0038
Y = kate ->;
No
?-
```

## Seznamy

```
?- L=[a,b,c,d,1,2,3].
L = [a,b,c,d,1,2,3] -> ;
No
?- L=[susan,fred,[jane,jim,jack]].
L = [susan,fred,[jane,jim,jack]] ->
Yes
?- L=[ ].
L = [ ] ->
Yes
?- L=[a,1,X,b(c,d),[5,6,7],"hallo"].
L = [a,1,_004C,b(c,d),[5,6,7],[104,97,108,108,111]] ->
Yes
?- abc = ‘abc’
Yes
?-
```

```
?- [a,b,c,d]=[H|T].
H = a
T = [b,c,d] ->;
No
?- [a,b,c] = [H1,H2|T].
H1 = a
H2 = b
T = [c] ->
Yes
?- [a,b,c] = [H1,H2,H3|T].
H1 = a
H2 = b
H3 = c
T = [ ] ->
Yes
```

```
?- [a,b,c] = [H1,H2,H3,H4|T].
No
?- T = [b,c,d], L = [a|T].
T = [b,c,d]
L = [a,b,c,d] ->
Yes
?- [a,b,c,d] = [ _ |T].
T = [b,c,d] ->
Yes
?- [a,b,c,d] = [H| _ ].
H = a ->;
No
?- [a,b,c,d] = [H| _A].
H = a
_A = [b,c,d] ->
Yes
?-
```

# Definice nových klauzulí – programování v PROLOGu

```
member(X,[X | _ ]).
member(X,[ _ |T]) :- member(X,T).
**********************************************

append([ ],L,L).
append([H|T1],L2,[H|T3]) :- append(T1,L2,T3).
**********************************************

write_list([ ]).
write_list([H|T]) :- write(H), nl, write_list(T).
**********************************************

reverse_write_list([ ]).
reverse_write_list([H|T]) :-
        reverse_write_list(T), write(H), nl.
```

```
?- member(3,[1,2,3,4,5]).
Yes
?- member(X,[a,b,c]).
X = a ->;
X = b ->;
X = c ->;
No
?- append([a,b,c],[d,e,f,g],L).
L = [a,b,c,d,e,f,g] ->
Yes
?- append(L,[e,f,g],[1,2,e,f,g]).
L = [1,2] ->
Yes
?-
```

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- mother(X,Y).
ancestor(X,Y) :- father(X,Z),
                 ancestor(Z,Y).
ancestor(X,Y) :- mother(X,Z),
                 ancestor(Z,Y).

father(george,charles).
father(charles,robert).
father(robert,john).
father(eve,paul).
mother(george,eve).
mother(charles,jane).
mother(robert,mary).
mother(eve,ann).
```

```
?- ancestor(george,X).
X = charles ->;
X = eve ->;
X = robert ->;
X = jane ->;
X = john ->;
X = mary ->;
X = paul ->;
X = ann ->;
No
?- ancestor(eve,Y).
Y = paul ->;
Y = ann ->;
No
?-
```

## Některé důležité zabudované predikáty

Konvence zápisů argumentů:

| | |
|---|---|
| +ARG | Argument musí být navázán na term, který splňuje požadavky na typ tohoto argumentu („vstupní“ argument). |
| –ARG | Argumentem musí být volná proměnná („výstupní“ argument). |
| ?ARG | Argumentem musí být term příslušného typu (buď „vstupní“, nebo „výstupní“ argument). |

## Predikáty pro práci s databází

| | |
|---|---|
| consult(+Filename) | Čte klauzule ze souboru a přidává je na konec databáze |
| assert(+Clause) | Přidá klauzuli na konec databáze |
| asserta(+Clause) | Přidá klauzuli na začátek databáze |
| retract(+Clause) | Odstraní první výskyt klauzule z databáze |
| retractall(+Clause) | Odstraní všechny výskyty klauzule z databáze |
| listing | Vypíše aktuální databázi |

Příklad použití

```
?- assert(a(b)).
Yes
?- asserta(a(z)).
Yes
?- assert(c(d)).
Yes
?- assert ((a(X):- c(X))).
X = _0038 ->
Yes
?- listing.
c(d).
a(z).
a(b).
a(A) :- c(A).
Yes
?-
```

## I/O predikáty

| | |
|---|---|
| tell(+Filename) | Otevírá soubor pro zápis |
| told | Uzavírá soubor otevřený pro zápis |
| write(+Term) | Zapíše term do souboru otevřeného pro zápis (standardně na obrazovku) |
| nl | Přejde na nový řádek (v zápisu!) |
| see(+Filename) | Otevírá soubor pro čtení |
| seen | Uzavírá soubor otevřený pro čtení |
| read(–Term) | Čte term ze souboru otevřeného pro čtení (standardně z klávesnice) |

# Predikáty – Operátory (infixová a prefixová notace)

```
?- (15 - 6) = '-'(15,6).
Yes
?- (15 - 6).
No
?-
```

**********

**X is E** **Vyhodnotí výraz E a výsledek přiřadí X**

**is(X, E)** **(akceptuje operátory + , - , * , / , // , mod )**

**********

```
?- X = 5 + 2, write(X).
5 + 2
X = 5 + 2 ->
Yes
```

```
?- X is 5 + 2, write(X).
7
X = 7 ->
Yes
?-
```

?- X is '+'(4,5).
X = 9 ->
Yes
?-
**********

Porovnání hodnot s vyhodnocením výrazů E1 a E2

E1 < E2
E1 =< E2
E1 > E2
E1 >= E2
E1 =:= E2
E1 =\= E2

## Ztotožnění a porovnání termů

| | |
|---|---|
| T1 = T2 | Uspěje, jde-li termy ztotožnit (volné proměnné se naváží) |
| T1 = = T2 | Uspěje, jsou-li termy stejné |
| T1 \= T2 | Uspěje, pokud termy nelze ztotožnit |
| T1 \= = T2 | Uspěje, pokud termy nejsou stejné |

**********

```
?- L = [a,b,c], L = = X.
No
?- L = [a,b,c], L = X.
L = [a,b,c]
X = [a,b,c] ->
Yes
?-
```

## Další užitečné predikáty

bagof(?T,+G,-B) Vrací seznam B všech instancí termu T, pro které uspěje cíl G

**********

```
ma(karel,marii).
ma(honza,evu).
ma(karel,alika).
ma(jiri,zeryka).
ma(karel,nuz).
ma(honza,sirky).
ma(karel,zizen).
co_ma(X,S):- bagof(Y,ma(X,Y),S).
```

**********

```
?- co_ma(karel,S).
S = [marii,alika,nuz,zizen] ->
Yes
?-
```

! Řez - uspěje, ale nepovolí návrat.

**********

```
girl(susan).
girl(ann).
boy(john).
boy(george).
pair(B,G) :- boy(B), !, girl(G).
```

**********

```
?- pair(X,Y).
X = john
Y = susan -> ;
X = john
Y = ann -> ;
No
?- halt.
```

Ukončuje činnost PROLOGu

# ADT v Prologu (zásobník, fronta, množina)

empty([ ]). *vytvoření prázdného seznamu / test na prázdný seznam (stejné pro zásobník, frontu i množinu)*

**********

?- empty(L).
L = [ ] ->
Yes
?- empty([a,b,c]).
No
?-

Pozn.: V následujících predikátech pro vložení a výběr prvku je prvním argumentem predikátu vkládaný/vybíraný prvek (E - element), druhým argumentem aktuální seznam (zásobník, fronta, množina) a třetím argumentem nový seznam po provedené operaci.

**Zásobník (Stack)**

```
push(E,T,[E|T]).
```
*vložení prvku*

```
pop(E,[E|T],T).
```
*výběr prvku*

**Fronta (Queue)**

```
enqueue(E,[ ],[E]).
enqueue(E,[H|T2],[H|T3]) :- enqueue(E,T2,T3).
```
*vložení prvku*

```
dequeue(E,[E|T],T).
```
*výběr prvku*

**Fronta s prioritou (Priority Queue)**

```
insert(E,[ ],[E]).
insert(E,[H|T],[E,H|T]) :- precedes(E,H).
insert(E,[H|T2],[H|T3]) :- precedes(H,E), insert(E,T2,T3).
```
*vložení prvku*

```
precedes(A,B) :- A < B.
```
*pro čísla (jako příklad)*

```
dequeue(E,[E|T],T).
```
*výběr prvku*

**Množina (Set)**

```
add(E,S,S) :- member(E,S), !.                    vložení prvku
add(E,S,[E|S]).

delete(E,[ ],[ ]).                               výběr prvku
delete(E,[E|T],T) :- !.
delete(E,[H|T2],[H|T3]) :- delete(E,T2,T3).
```

***Sjednocení dvou množin***

```
union([ ],S,S).
union([H|T],S2,S3) :- union(T,S2,S4), add(H,S4,S3).
```

***Průnik dvou množin***:

```
intersection([ ],_,[ ]).
intersection([H|T1],S,[H|T3]) :-  member(H,S),!,
                                  intersection(T1,S,T3]).
intersection([H|T1],S2,S3]) :- intersection(T1,S2,S3).
```

***Rozdíl dvou množin***:

```
difference([ ],_,[ ]).
difference([H|T ],S2,S3) :- member(H,S2), difference(T,S2,S3).
difference([H|T1],S,[H|T3]) :- not(member(H,S)),
                               difference(T1,S,T3).
```

***Porovnání dvou množin***:

```
subset([ ],_).
subset([H|T],S) :- member(H,S), subset(T,S).

equal(S1,S2) :- subset(S1,S2), subset(S2,S1).
```

## Depth First Search in PROLOG

```
dfs(Start,Goals) :- path([[Start]],Goals).          % path(Open,Goals)

path([ ],_) :- write('No solution was found.'),!.
path(Open,Goals) :-                                 % node: [State|Parents]
        pop([State|Ancestors],Open,_),
        member(State,Goals),
        write('Solution has been found! '),
        write('The path is:'),!,nl,
        print_solution([State|Ancestors]).
path(Open,Goals) :-
        pop([State|Ancestors],Open,Rest_open),
        get_children(State,Ancestors,Rest_open,Children),
        append(Children,Rest_open,New_open),
        path(New_open,Goals).
```

```
pop(Top,[Top|Stack],Stack).

get_children(State,Ancestors,Rest_open,Children) :-
        bagof(Child,moves(State,Ancestors,Rest_open,Child),Children).
get_children(State,Ancestors,Rest_open,[]).

moves(State,Ancestors,Rest_open,[Next,State|Ancestors]) :-
        move(State,Next),
        not(member(Next,Ancestors)),
        not(member(Next,Rest_open)).

member(E,[E|_]).                                  % platí pro obecný seznam
member(E,[H|_]) :- member(E,H).
member(E,[_|T]) :- member(E,T).

append([ ],S,S).
append([H|T1],T2,[H|T3]) :- append(T1,T2,T3).

print_solution([State]) :- write(State), nl.
print_solution([State|T]) :- print_solution(T), write(State), nl.
```

% clauses for 4-3 Jugs Problem

```
move([V,M],[4,M]) :- V < 4.
move([V,M],[V,3]) :- M < 3.
move([V,M],[0,M]) :- V > 0.
move([V,M],[V,0]) :- M > 0.
move([V,M],[0,MN]) :- V > 0, M < 3, 3 – M >= V, MN is M + V.
move([V,M],[VN,3]) :- V > 0, M < 3, 3 – M < V, VN is V – 3 + M.
move([V,M],[VN,0]) :- M > 0, V < 4, 4 – V >= M, VN is M + V.
move([V,M],[4,MN]) :- M > 0,V < 4, 4 – V < M, MN is M – 4 + V.
```

```
?- dfs([0,0],[[0,2],[2,0]]).
Solution has been found! The path is:
[0, 0]
[4, 0]
[1, 3]
[1, 0]
[0, 1]
[4, 1]
[2, 3]
[2, 0]
Yes
?-
```

**Breadth First Search in PROLOG**

```
bfs(Start,Goals) :- path([[Start,nil]],[ ],Goals).    %path(Open,Closed,Goals)

path([ ],_,_) :- write('No solution was found.'),!.
path(Open,Closed,Goals) :-                          % node: [State,Parent]
        dequeue([State,Parent],Open,_),
        member(State,Goals),
        write('The best solution has been found - the path is:'), nl,
        print_solution([State,Parent],Closed).
path(Open,Closed,Goals) :-
        dequeue([State,Parent],Open,Rest_open),
        get_children(State,Rest_open,Closed,Children),
        append(Rest_open,Children,New_open),
        append([[State,Parent]],Closed,New_closed),
        path(New_open,New_closed,Goals).
```

```
dequeue(E,[E|T],T).

get_children(State,Rest_open,Closed,Children) :-
        bagof(Child,moves(State,Rest_open,Closed,Child),Children).
get_children(State,Rest_open,Closed,[ ]).

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).

append([ ],S,S).
append([H|T1],T2,[H|T3]) :- append(T1,T2,T3).

print_solution([State,nil],_) :-
        write(State), nl.
print_solution([State,Parent],Closed) :-
        member([Parent,Grandparent],Closed),
        print_solution([Parent,Grandparent],Closed),
        write(State), nl.
```

```
moves(State,Rest_open,Closed,[Next,State]):-
        move(State,Next),
        not(member([Next,_],Rest_open)),
        not(member([Next,_],Closed)).

% clauses for 8 - puzzle

move(State,Next) :- N=3,                    % move up
        find_pos(State,SP),                 % SP – actual space position
        SP > N,
        NP is SP – N,                       % NP - new space position
        change(State,NP,SP,Next).
move(State,Next) :- N=3,                    % move right
        find_pos(State,SP),
        R is SP mod N,
        R \= 0,
        NP is SP + 1,
        change(State,NP,SP,Next).
```

```
move(State,Next) :- N=3,                    % move down
        find_pos(State,SP),
        NN is N * (N – 1) + 1,
        SP < NN,
        NP is SP + N,
        change(State,NP,SP,Next).
move(State,Next) :- N=3,                    % move left
        find_pos(State,SP),
        R is SP mod N,
        R \= 1,
        NP is SP – 1,
        change(State,NP,SP,Next).

find_pos([o|_],1).                          % find space position
find_pos([H|T],SP) :-
        H \= o,
        find_pos(T,SPM1),
        SP is SPM1 + 1.
```

```
change(State,NP,SP,Next) :-
        move_sp(State,NP,Piece,Temp_State),
        move_piece(Temp_State,SP,Piece,Next).

move_sp([H|T],1,H,[o|T]).
move_sp([H|T],N,Piece,[H|TN]) :-
        N > 1,
        NM1 is N – 1,
        move_sp(T,NM1,Piece,TN).

move_piece([_|T],1,H,[H|T]).
move_piece([H|T],N,Piece,[H|TN]) :-
        N > 1,
        NM1 is N – 1,
        move_piece(T,NM1,Piece,TN).
```

?- bfs([1,4,2,7,8,3,o,6,5],[[1,2,3,8,o,4,7,6,5]]).
Solution has been found! The path is:

```
142
783
o65

142
o83
765

142
8o3
765

1o2
843
765
```

```
12o
843
765

123
84o
765

123
8o4
765

Yes
?-
```

## Best First Search (A*) in PROLOG

```
a_star(Start,Goal) :-                          % one goal, only !
        estimate(Start,Goal,0,_,H),            % estimation of H
        path([[Start,nil,0,H,H]],[ ],Goal).    % [State,Parent,G,H,F]

path([ ],_,_) :-                               % path(Open,Closed,Goal)
        write('No solution was found.'), !.
path(Open,Closed,Goal) :-
        dequeue([Goal,Parent|_],Open,_),
        write('The best solution has been found - the path is:'), nl,
        print_solution([Goal,Parent | _],Closed).
path(Open,Closed,Goal) :-
        dequeue([State,Parent,G,H,F],Open,Rest_open),
        get_children(State,G,F,Rest_open,Closed,Goal,[ ],Children),
        insert_to_open(Rest_open,Children,New_open),
        append([[State,Parent,G,H,F]],Closed,New_closed),
        path(New_open,New_closed,Goal).
```

```
dequeue(E,[E|T],T).

get_children(State,G,F,Rest_open,Closed,Goal,Temp,Children) :-
        move(State,Next),
        not(member([Next|_],Closed)),
        not(member([Next|_],Temp)),
        estimate(Next,Goal,G,Gnew,H),
        Ftemp is Gnew + H,
        max(F,Ftemp,Fnew),
        get_children(State,G,F,Rest_open,Closed,Goal,
                        [[Next,State,Gnew,H,Fnew]|Temp],Children),!.
get_children(_,_,_,_,_,_,L,L).

max(A,B,A) :- A >= B.
max(A,B,B) :- A < B.
```

```
insert_to_open(Open,[ ],Open).
insert_to_open(Open,[H|T],Open_new) :-
        insert(Open,H,Temp_open),
        insert_to_open(Temp_open,T,Open_new).

insert([ ],H,[H]).
insert(Open,[Next,P,G,H,F],New_open) :-
        member([Next,_,_,_,_],Open), !,
        insert_change(Open,[Next,P,G,H,F],New_open).
insert([[S1,P1,G1,H1,F1]|T1],[S2,P2,G2,H2,F2],
                                    [[S2,P2,G2,H2,F2],[S1,P1,G1,H1,F1]|T1]) :-
        F1 >= F2.
insert([[S1,P1,G1,H1,F1]|T1],[S2,P2,G2,H2,F2],[[S1,P1,G1,H1,F1]|T3]) :-
        F1 < F2,
        insert(T1,[S2,P2,G2,H2,F2],T3).
```

```
%insert_change(Open,Child,Open_new)

insert_change(Open,[Node,Pnew,Gnew,Hnew,Fnew],Open) :-
        remove(Open,[Node,Pnew,Gnew,Hnew,Fnew],
                                        [Node,_,_,_,Fold],_),
        Fold =< Fnew.
insert_change(Open,[Node,Pnew,Gnew,Hnew,Fnew],Open_new) :-
        remove(Open,[Node,Pnew,Gnew,Hnew,Fnew],
                                        [Node,_,_,_,Fold],Open_temp),
        Fold > Fnew,
        insert(Open_temp,[Node,Pnew,Gnew,Hnew,Fnew],Open_new).

remove([[Node,Pold,Gold,Hold,Fold]|T],[Node,_,_,_,_],
                                        [Node,Pold,Gold,Hold,Fold],T).
remove([H|T],[Node,_,_,_,_],[Node,Pold,Gold,Hold,Fold],[H|Ttemp]) :-
         remove(T,[Node,_,_,_,_],[Node,Pold,Gold,Hold,Fold],Ttemp).
```

```
member(H,[H|_]).
member(H,[_|T]) :- member(H,T).

append([ ],S,S).
append([H|T1],T2,[H|T3]) :- append(T1,T2,T3).

print_solution([State,nil|_],_) :-
        write1(State),nl.
print_solution([State,Parent|_],Closed) :-
        member([Parent,Grandparent|_],Closed),
        print_solution([Parent,Grandparent|_],Closed),
        write1(State), nl.

write1([ ]).
write1(L) :- size(N), write2(L,N).

write2([H|T],M) :- M > 0, write(H), MM1 is M - 1, write2(T,MM1).
write2(L,M) :- nl, write1(L).
```

% clauses for ($n \cdot n$-1) – puzzle, $n$ = 3,4,…

```
estimate(State,Goal,G,Gnew,H) :-
        size(N), NN is N * N,
        evaluate(State,Goal,OK),          % OK … number of pieces
        H is NN – 1 – OK,                 % in right positions
        Gnew is G+1.                      % h1 heuristic

evaluate([ ],[ ],0).
evaluate([o|TState],[o|TGoal],OK) :-
                                  evaluate(TState,TGoal,OK).
evaluate([A|TState],[B|TGoal],OK) :- A \= B,
                                  evaluate(TState,TGoal,OK).
evaluate([H|TState],[H|TGoal],OK) :- H \= o,
                                  evaluate(TState,TGoal,TOK),
                                  OK is TOK + 1.
```

```
move(State,Next) :- size(N),                    % up
        find_pos(State,SP),
        SP > N,
        NP is SP – N,
        change(State,NP,SP,Next).
move(State,Next) :- size(N),                    % right
        find_pos(State,SP),
        R is SP mod N,
        R \= 0,
        NP is SP + 1,
        change(State,NP,SP,Next).
move(State,Next) :- size(N),                    % down
        find_pos(State,SP),
        NN is N * (N – 1) + 1,
        SP < NN,
        NP is SP + N,
        change(State,NP,SP,Next).
```

```
move(State,Next) :- size(N),                    % left
        find_pos(State,SP),
        R is SP mod N,
        R \= 1,
        NP is SP – 1,
        change(State,NP,SP,Next).

find_pos([o|_],1).                              % space position
find_pos([H|T],SP) :-
        H \= o,
        find_pos(T,SPM1),
        SP is SPM1 + 1.

change(State,NP,SP,Next) :-
        move_sp(State,NP,Piece,Temp_State),
        move_piece(Temp_State,SP,Piece,Next).
```

```
move_sp([H|T],1,H,[o|T]).
move_sp([H|T],N,Piece,[H|TN]) :-
        N > 1,
        NM1 is N – 1,
        move_sp(T,NM1,Piece,TN).

move_piece([_|T],1,H,[H|T]).
move_piece([H|T],N,Piece,[H|TN]) :-
        N > 1,
        NM1 is N – 1,
        move_piece(T,NM1,Piece,TN).
```

```
?- assert(size(0)), retractall(size(N)), assert(size(4)),
   a_star([o,1,2,4,5,6,3,8,9,a,7,b,d,e,f,c],[1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,o]).

Solution was found. The path is:
o124 | 1234 | 1234
5638 | 56o8 | 5678
9a7b | 9a7b | 9abc
defc | defc | defo
     |      | Yes
1o24 | 1234 | ?-
5638 | 5678
9a7b | 9aob
defc | defc
     |
12o4 | 1234
5638 | 5678
9a7b | 9abo
defc | defc
```

| Počet stavů v Open a Closed (8 – puzzle) | | | | |
|---|---|---|---|---|
| Hloubka řešení | Breadth First Search | | Best First Search – A* | |
| | Open | Closed | Open | Closed |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 3 | 3 | 3 | 1 |
| 2 | 4 | 4 | 3 | 2 |
| 3 | 10 | 10 | 4 | 3 |
| 4 | 16 | 22 | 6 | 4 |
| 5 | 30 | 42 | 7 | 5 |
| 6 | 41 | 60 | 7 | 6 |