În acest laborator ne vom concentra atenția asupra unor jocuri de tip puzzle şi a altor probleme de inteligență artificială care pot fi rezolvate de o singură persoană.

Probleme rezolvate

1. Problema labirintului.

Primul exemplu de puzzle este un labirint simplu. Scopul este acela de a găsi o cale de parcurgere a labirintului de la un capăt (*Intrare – Start*) la celălalt (*Ieşire - Sosire*). În primul rând, trebuie să reprezentăm labirintul într-o formă pe care limbajul Prolog o poate înțelege. Labirintul a fost construit pe o hârtie milimetrică şi are 6 pătrățele lățime şi 6 pătrățele lungime. Începem prin a numerota aceste pătrățele (Figura 6.1).

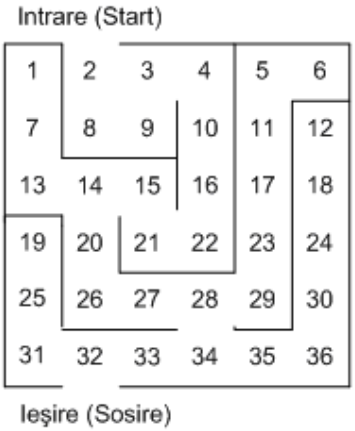

Figura 6.1 *Un labirint simplu în interiorul căruia sunt numerotate locațiile*

Tratând ***Intrare(Start)*** şi ***Ieşire(Sosire*)** ca poziții în labirint, rezultă că avem în total 38 de poziții. De la fiecare din acestea ne putem deplasa spre alte poziții posibile. Dacă putem trece dintr-o poziție în alta, spunem că aceste două poziții sunt conectate. Pentru aceasta introducem predicatul **conecteaza** pentru fiecare pereche de locații conectate. Apoi definim predicatul **conectate** folosind predicatul **conecteaza:**

```
conecteaza(plecare, 2).
conecteaza(1, 7).
```

```
conecteaza(2, 8).
.........................
conecteaza(32, sosire).
conectate(Locatie1,Locatie2):- conecteaza(Locatie1,Locatie2).
conectate(Locatie1,Locatie2):- conecteaza(Locatie2,Locatie1).
```

O cale prin labirint reprezintă o listă de poziții cu **start** la un capăt al listei şi **sosire** la celălalt capăt, astfel încât fiecare poziție din listă este conectată la pozițiile dinainte şi de după ea. Inițial, calea conține un singur punct **start**, pe care îl plasăm într-o listă. De la acest punct inițial vom genera o cale completă (**start,** …, **sosire**). La fiecare pas intermediar în căutarea soluției labirintului, avem o listă a pozițiilor pe care deja le-am vizitat. Primul membru al acestei liste este poziția curentă (*LocatieCurenta*). Avansăm, căutând o poziție nouă la care să ajungem de la poziția curentă.

Pentru a evita învârtirea în cerc sau deplasarea înainte şi înapoi, între (trecând prin) aceleaşi poziții, trebuie ca noua poziție să nu existe deja în calea pe care încercăm să o construim.

```
cale([LocatieCurenta|RestulCaii], Solutie):-
                conectate(LocatieCurenta, urmatoareaLocatie),
                \+ member(UrmatoareaLocatie, RestulCaii),
                cale([UrmatoareaLocatie,
                LocatieCurenta|RestulCaii],Solutie).
```

Dacă funcția **cale** atinge un punct din care nu poate găsi o nouă poziție, atunci limbajul Prolog se va întoarce (înapoi) pe acelaşi drum (metoda backtracking). Pozițiile vor fi plasate în fața căii pe care am construit-o, până când atingem o poziție nouă. Apoi, căutarea va avansa, până când atingem **sosire** sau alt punct final (capăt). Din moment ce labirintul are o soluție, aplicația Prolog o va găsi (evident dacă am definit corect procedura de căutare).

Vom construi un fişier ***labirint-bd.pl***, ce conține baza de cunoştințe (tabelul de conexiuni pentru labirintul din figura 6.1). Fişierul are următorul conținut:

```
conecteaza(plecare,2).          conecteaza(1,7).
conecteaza(2,8).                conecteaza(3,4).
conecteaza(3,9).                conecteaza(4,10).
conecteaza(5,11).               conecteaza(5,6).
conecteaza(7,13).               conecteaza(8,9).
conecteaza(10,16).              conecteaza(11,17).
```

```
conecteaza(12,18).          conecteaza(13,14).
conecteaza(14,15).          conecteaza(14,20).
conecteaza(15,21).          conecteaza(16,22).
conecteaza(17,23).          conecteaza(18,24).
conecteaza(19,25).          conecteaza(20,26).
conecteaza(21,22).          conecteaza(23,29).
conecteaza(24,30).          conecteaza(25,31).
conecteaza(26,27).          conecteaza(27,28).
conecteaza(28,29).          conecteaza(28,34).
conecteaza(30,36).          conecteaza(31,32).
conecteaza(32,33).          conecteaza(33,34).
conecteaza(34,35).          conecteaza(35,36).
conecteaza(32,sosire).
```

Codul sursă SWI-Prolog pentru fişierul ***rezolva-labirint.pl*** este următorul (predicatul \+ are sensul de ***not***):

```
rezolvaLabirint:-
            reconsult('C:/labirint-bd.pl'), nl,
            cale([plecare],Solutie), write(Solutie).

cale([sosire|RestulCaii],[sosire|RestulCaii]).
cale([LocatieCurenta|RestulCaii],Solutie):-
            conectate(LocatieCurenta,UrmatoareaLocatie),
            \+ member(UrmatoareaLocatie,RestulCaii),
            cale([UrmatoareaLocatie,LocatieCurenta
            |RestulCaii],Solutie).

conectate(Locatie1,Locatie2):-
            conecteaza(Locatie1,Locatie2).
conectate(Locatie1,Locatie2):-
            conecteaza(Locatie2,Locatie1).

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

La testare, răspunsul sistemului este:

```
5 ?- rezolvaLabirint.
% C:/labirint-bd.pl compiled 0.00 sec, 2,668 bytes
[sosire, 32, 33, 34, 28, 27, 26, 20, 14, 15, 21, 22, 16,
10, 4, 3, 9, 8, 2, plecare]
true
```

2. Problema misionarilor şi a canibalilor.

Cel de-al doilea puzzle propus în acest laborator se referă la problema misionarilor şi a canibalilor. Trei misionari şi trei canibali trebuie să traverseze un râu, dar singura barcă disponibilă poate duce doar doi oameni la un moment dat. Nu există pod, râul nu poate fi trecut înot şi barca nu poate trece râul fără cineva în ea. Canibalii vor mânca misionarii dacă sunt mai mulți ca ei pe unul dintre maluri. Problema constă în trecerea tuturor persoanelor pe malul celălalt fără nici un misionar mâncat.

Asemenea problemei labirintului, vom începe prin a decide cum vom reprezenta problema în Prolog. Am desemnat un mal al râului ca fiind malul stâng, iar celălalt – malul drept şi presupunem că, la început, barca, misionarii şi canibalii se află pe malul stâng. La fiecare etapă de traversare a misionarilor şi a canibalilor peste râu, vom avea nişte misionari pe malul stâng si pe malul drept, nişte canibali pe malul stâng şi pe malul drept, iar barca va fi, fie pe malul stâng, fie pe cel drept. Putem reprezenta această informație printr-o structură care să ne spună câți misionari sunt pe malul stâng, câți canibali sunt pe malul stâng şi locația bărcii: ***etapa(M, C, B)***. Aici, variabilele *M* şi *C* pot lua valori între 0 şi 3, iar variabila *B* poate lua valoarea *s* (malul stâng) sau *d* (malul drept).

Structura **etapa(3, 3, s)** reprezintă situația de la începutul puzzle-ului, iar structura **etapa(0, 0, d)** reprezintă situația la care vrem să ajungem. O soluție a puzzle-ului ar fi o listă de situații cu **etapa(3, 3, s)** la un capăt al listei şi **etapa(0, 0, d)** la celălalt. Din moment ce programul produce o listă de etape în ordinea inversă celei în care se întâmplă, vom pune programul să inverseze soluția înaintea afişării ei.

```
canibal :- solutie_canibal([etapa(3,3,s)], Solutie),
          reverse(Solutie, [], SolutieOrdonata),
          afisare_etape(SolutieOrdonata).
```

Oricare două etape succesive din soluție vor trebui să satisfacă o serie de condiții:

- În primul rând, barca va fi pe maluri diferite ale râului în cele două etape. Asta deoarece nimeni nu poate traversa râul fără barcă.

- În al doilea rând, canibalii nu vor fi niciodată mai mulți ca misionarii pe nici un mal al râului.
- În al treilea rând, fiecare etapă va rezulta din cealaltă prin trimiterea bărcii cu misionari şi canibali peste râu, în direcția potrivită.

Vom rezolva acest puzzle în aproape acelaşi mod în care am rezolvat problema labirintului, cu excepția faptului că, la fiecare etapă din căutarea soluției, restricțiile pentru următoarea mutare vor fi mult mai complicate. Vrem ca Prolog-ul să exploreze toate combinațiile posibile de treceri peste râu până va găsi una potrivită. La orice moment din acest proces, vom avea o listă de stări care rezultă din trecerile care au fost făcute până atunci. Primul membru al acestei liste este starea curentă. Vrem ca Prolog-ul să caute o nouă stare care poate fi ajunsă din starea curentă. Starea curentă trebuie să fie rezultatul unei treceri legale din starea curentă, nu trebuie să pună în pericol nici un misionar şi trebuie să fie o stare care nu a mai fost. Ultima cerință împiedică Prolog-ul să mute aceiaşi oameni de pe un mal pe altul fără a face nici un progres. Pentru a îndeplini acestea, adăugăm două clauze la definiția **solutie_canibal**, una pentru momentul când barca este pe malul stâng şi cealaltă atunci când este pe malul drept, astfel:

```
solutie_canibal([etapa(M1, C1, s) | Etapele_anterioare], Solutie):-
   membru([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),     %Condiția 1
   M1>=M,                                              %Condiția 2
   C1>=C,                                              %Condiția 3
   M2 is M1-M,                                         %Condiția 4
   C2 is C1-C,                                         %Condiția 5
   membru([M2,C2],[[3,_],[0,_],[N,N]]),                %Condiția 6
   \+ membru(etapa(M2,C2,d),Etapele_anterioare),   %Cond 7
   solutie_canibal([etapa(M2, C2, d), etapa(M1, 1, s) |
   Etapele_anterioare], Solutie).

solutie_canibal([etapa(M1, C1, d)|Etapele_anterioare], Solutie) :-
   membru([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),
   3-M1>=M, 3-C1>=C,
   M2 is M1+M, C2 is C1+C,
   membru([M2,C2],[[3,_],[0,_],[N,N]]),
   \+membru(etapa(M2,C2,s),Etapele_anterioare),
```

solutie_canibal([etapa(M2,C2,s),etapa(M1,C1,d)|Etapele_anterioare], Solutie).

Pentru a înțelege programul trebuie să înțelegem condițiile din prima clauză prezentată mai sus.

Condiția 1 arată că barca trebuie să care cel puțin unul şi cel mult doi indivizi peste râu.

Condițiile 2 şi 3 asigură că nu vor intra pe barcă mai mulți misionari şi canibali decât cei prezenți pe malul stâng al râului.

Condițiile 4 şi 5 determină câți misionari şi câți canibali vor fi pe malul stâng după următoarea traversare.

Condiția 6 verifică dacă misionarii sunt teferi după traversare. O stare este sigură dacă toți misionarii sunt pe acelaşi mal (şi, deci, nu pot fi mai puțini decât canibalii) sau dacă este un număr egal de canibali şi misionari pe malul stâng (deci şi un număr egal pe malul drept).

În final, **condiția 7** garantează că programul nu se întoarce la o stare anterioară.

Dacă **solutie_canibal** ajunge la un moment de unde nu mai poate găsi o trecere în siguranță care va produce o situație care nu a fost încercată încă, Prolog se întoarce (pe căi dintre calea inițială şi calea curentă) la un punct unde este posibilă o nouă situație. Apoi căutarea continuă, din nou, înainte.

O posibilă implementare în Prolog a problemei misionarilor şi a canibalilor este următoarea (*fişierul Canibal.pl*):

```
%canibal.pl
%solutie la problema canibalilor si misionarilor
canibal :- solutie_canibal([etapa(3,3,s)],Solutie),
           reverse(Solutie,[],SolutieOrdonata),
           afisare_etape(SolutieOrdonata).

membru(X,[X|_]).
membru(X,[_|Y]) :- membru(X,Y).

reverse([],List,List).
reverse([X|Coada],Rezolvate,List) :-
        reverse(Coada,[X|Rezolvate],List).
```

```
solutie_canibal([etapa(0,0,d)|Etapele_anterioare],
                [etapa(0,0,d)|Etapele_anterioare]).
solutie_canibal([etapa(M1,C1,s)
                |Etapele_anterioare],Solutie) :-
        membru([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),
        M1>=M,C1>=C,
        M2 is M1-M, C2 is C1-C,
        membru([M2,C2],[[3,_],[0,_],[N,N]]),
        \+ membru(etapa(M2,C2,d),Etapele_anterioare),
        solutie_canibal([etapa(M2,C2,d),etapa(M1,C1,s)
                        |Etapele_anterioare],Solutie).
solutie_canibal([etapa(M1,C1,d)
                |Etapele_anterioare],Solutie) :-
        membru([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),
        3-M1>=M, 3-C1>=C,
        M2 is M1+M, C2 is C1+C,
        membru([M2,C2],[[3,_],[0,_],[N,N]]),
        \+membru(etapa(M2,C2,s),Etapele_anterioare),
        solutie_canibal([etapa(M2,C2,s),etapa(M1,C1,d)
                 |Etapele_anterioare],Solutie).

scrie_n_ori(_,0) :- !.
scrie_n_ori(Item,N) :-
        write(Item),
        M is N-1,
        scrie_n_ori(Item,M).

deseneaza_barca(s) :- write(' (_ _ _ _)         ').
deseneaza_barca(d) :- write('         (_ _ _ _) ').

afisare_etape([]).
afisare_etape([etapa(M,C,Locatie)|Etape_posterioare]) :-
          scrie_n_ori('M',M),
          scrie_n_ori('C',C),
          N is 6-M-C,
          scrie_n_ori(' ',N),
          deseneaza_barca(Locatie) ,
          MM is 3-M,
          CC is 3-C,
          scrie_n_ori('M',MM),
          scrie_n_ori('C',CC),
          nl,
          afisare_etape(Etape_posterioare).
```

Afişăm o soluție a problemei printr-o serie de imagini. Aceasta presupune inversarea listei soluției pentru ca stările să fie afişate în ordinea corectă generată. Predicatul **afisare_etape** şi auxiliarele lui fac acest lucru, desenând imagini cu ordinarele caractere ASCII aşa cum este prezentat în Figura 6.2.

```
9 ?- canibal.
MMMCCC (_ _ _ _)
MMCC           (_ _ _ _) MC
MMMCC  (_ _ _ _)         C
MMM            (_ _ _ _) CCC
MMMC   (_ _ _ _)         CC
MC             (_ _ _ _) MMCC
MMCC   (_ _ _ _)         MC
CC             (_ _ _ _) MMMC
CCC    (_ _ _ _)         MMM
C              (_ _ _ _) MMMCC
CC     (_ _ _ _)         MMMC
               (_ _ _ _) MMMCCC
true
```

Figura 6.2 *O soluţie la problema misionarilor şi a canibalilor*

3. Problema triunghiului ("triangle puzzle").

Puzzle-ul "Triunghiul", numit si "Puzzle-ul Pomului de Crăciun", este jucat cu cincisprezece bețe de culori diferite amplasate pe o machetă. Macheta are 15 găuri amplasate într-un format triunghiular (figura 6.3).

Figura 6.3 *Macheta de joc*

Inițial, există un băț în fiecare gaură. Jucătorul elimină un băț, apoi sare de la un băț la altul şi, la fiecare săritură, elimină bățul peste care a sărit. Un salt trebuie întotdeauna să se realizeze în linie dreaptă. La un moment dat un

singur băț poate fi sărit. Scopul jocului este de a termina cu un singur băț pe macheta de joc.

O soluție a puzzle-ului este reprezentată de o succesiune de panouri cu paisprezece bețe în panoul inițial şi un băț în panoul final.

Programul nostru are la început o procedură, **triunghi (N),** care elimină al N-lea băț din triunghi. Procedura denumită **rezolva_triunghi** găseşte soluția şi o afişează sub formă de triunghi.

```
triunghi(N):-
            elimina_bat(N,TriunghiStart),
            rezolva_triunghi(14,[TriunghiStart],Solutie),
            nl,nl,
            afis_triunghi(Solutie).
```

Procedura **rezolva_triunghi** utilizează, ca prim argument, un număr pentru a ține evidența numărului de bețe din triunghiul curent. O soluție a problemei a fost atinsă în cazul în care există un singur băț rămas pe macheta de joc. Dacă sunt mai multe bețe rămase, procedura **rezolva_triunghi** caută, printr-un salt legal, un triunghi ce poate fi generat din triunghiul curent şi îl adaugă la lista de triunghiuri. Procedura afişează numărul de bețe rămase în noul triunghi. În continuare procedura se repetă până când se obține un triunghi cu un singur băț.

```
rezolva_triunghi(1,Solutie,Solutie).
rezolva_triunghi(Contor,[TriunghiCurent],Solutie):-
    salt(TriunghiCurent,TriunghiUrmator),
    ContorNou is Contor-1,
    write(ContorNou),nl,
    rezolva_triunghi(ContorNou,[TriunghiUrmator],Solutie).
```

Înainte de a putea utiliza **rezolva_triunghi,** avem nevoie de o modalitate de a calcula triunghiurile ce rezultă din salturile realizate în triunghiul curent. Există mai multe căi de a realiza acest lucru, dar una dintre cele mai simple este prezentată în cele ce urmează.

În primul rând, vom reprezenta fiecare triunghi printr-o listă de cincisprezece găuri. Fiecare gaură este reprezentată de o variabilă:

```
[A,
 B, C,
D, E, F,
G, H, I, J,
K, L, M, N, P].
```

Reprezentând un salt ca o transformare de la un triunghi la altul, vom observa că doar trei găuri într-un triunghi sunt afectate la fiecare salt. Restul de găuri, rămân goale sau ocupate, așa cum au fost înainte de salt. Vom utiliza **1** pentru a reprezenta o gaură în care se află un băț şi **0** pentru a reprezenta o gaură goală (în care nu se află un băț).

```
    0
   1 1
  1 1 1
 1 1 1 1
1 1 1 1 1
```

```
    1
   0 1
  0 1 1
 1 1 1 1
1 1 1 1 1
```

Astfel vom putea defini un predicat binar al saltului (**salt**), folosind un set de clauze cum ar fi următoarele:

```
salt(triunghi(1,
              1,C,
             0,E,F,
            G,H,I,J,
           K,L,M,N,P),
     triunghi(0,
              0,C,
             1,E,F,
            G,H,I,J,
           K,L,M,N,P)).
```

Avem nevoie de o astfel de clauză pentru fiecare salt posibil. Tot ceea ce rămâne de făcut este de a defini o procedură care va elimina bățul inițial şi rutine pentru afişarea soluției finale.

Codul sursă SWI-Prolog pentru fişierul ***puzzleTriunghi.pl*** este următorul:

```
% Procedura responsabila de afisarea solutiei.
afis_triunghi([]):- !.
afis_triunghi([triunghi(A,B,C,D,E,F,G,H,I,J,K,L,M,N,P)]):-
     spatiu(4), write(A), nl,
     spatiu(3), write(B), spatiu(1), write(C), nl,
     spatiu(2), write(D), spatiu(1), write(E), spatiu(1),
                                     write(F), nl,
     spatiu(1), write(G), spatiu(1), write(H), spatiu(1),
                write(I), spatiu(1), write(J),nl,
     write(K), spatiu(1), write(L), spatiu(1), write(M),
               spatiu(1), write(N), spatiu(1), writeln(P),
     write('apasa orice tasta pt a continua').
```

```
% Procedura de calcul a numarului de spatii necesare
% pentru afisarea explicita a solutiei.
spatiu(0).
spatiu(N):- write(' '),
            M is N-1,
            spatiu(M).

%triunghi(N)
% gaseste si afiseaza o solutie la problema triunghiului
% unde al N-lea bat este eliminat primul
triunghi(N):-
             elimina_bat(N,TriunghiStart),
             rezolva_triunghi(14,[TriunghiStart],Solutie),
             nl,nl,
             afis_triunghi(Solutie).

%rezolva_triunghi(N,Pattern,Solutie).
% cauta o solutie la problema triunghiului, pornind de la
% o pozitie aleasa de utilizator cu N bete % asezate in
% formatul dat de Pattern
rezolva_triunghi(1,Solutie,Solutie).
rezolva_triunghi(Contor,[TriunghiCurent],Solutie):-
    salt(TriunghiCurent,TriunghiUrmator),
    ContorNou is Contor-1,
    write(ContorNou),nl,
    rezolva_triunghi(ContorNou,[TriunghiUrmator],Solutie).

%elimina_bat(N,Triunghi).
% produce un triunghi din care elimina batul de pe poz. N.
elimina_bat(1,triunghi(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1)).
elimina_bat(2,triunghi(1,0,1,1,1,1,1,1,1,1,1,1,1,1,1)).
elimina_bat(3,triunghi(1,1,0,1,1,1,1,1,1,1,1,1,1,1,1)).
elimina_bat(4,triunghi(1,1,1,0,1,1,1,1,1,1,1,1,1,1,1)).
elimina_bat(5,triunghi(1,1,1,1,0,1,1,1,1,1,1,1,1,1,1)).
elimina_bat(6,triunghi(1,1,1,1,1,0,1,1,1,1,1,1,1,1,1)).
elimina_bat(7,triunghi(1,1,1,1,1,1,0,1,1,1,1,1,1,1,1)).
elimina_bat(8,triunghi(1,1,1,1,1,1,1,0,1,1,1,1,1,1,1)).
elimina_bat(9,triunghi(1,1,1,1,1,1,1,1,0,1,1,1,1,1,1)).
elimina_bat(10,triunghi(1,1,1,1,1,1,1,1,1,0,1,1,1,1,1)).
elimina_bat(11,triunghi(1,1,1,1,1,1,1,1,1,1,0,1,1,1,1)).
elimina_bat(12,triunghi(1,1,1,1,1,1,1,1,1,1,1,0,1,1,1)).
elimina_bat(13,triunghi(1,1,1,1,1,1,1,1,1,1,1,1,0,1,1)).
elimina_bat(14,triunghi(1,1,1,1,1,1,1,1,1,1,1,1,1,0,1)).
elimina_bat(15,triunghi(1,1,1,1,1,1,1,1,1,1,1,1,1,1,0)).
```

```
%salt(TriunghiCurent, TriunghiUrmator).
% Cauta urmatorul triunghi in care se poate ajunge din
% triunghiul curent printr-un salt corect. Pentru
% reducerea spatiului numai prima clausa o afisam in
% format liniar.
salt(triunghi(1,
              1,C,
             0,E,F,
            G,H,I,J,
           K,L,M,N,P),
     triunghi(0,
              0,C,
             1,E,F,
            G,H,I,J,
           K,L,M,N,P)).

salt(triunghi(1,B,1,D,E,0,G,H,I,J,K,L,M,N,P),
     triunghi(0,B,0,D,E,1,G,H,I,J,K,L,M,N,P)).
salt(triunghi(A,1,C,1,E,F,0,H,I,J,K,L,M,N,P),
     triunghi(A,0,C,0,E,F,1,H,I,J,K,L,M,N,P)).
salt(triunghi(A,1,C,D,1,F,G,H,0,J,K,L,M,N,P),
     triunghi(A,0,C,D,0,F,G,H,1,J,K,L,M,N,P)).
salt(triunghi(A,B,1,D,1,F,G,0,I,J,K,L,M,N,P),
     triunghi(A,B,0,D,0,F,G,1,I,J,K,L,M,N,P)).
salt(triunghi(A,B,1,D,E,1,G,H,I,0,K,L,M,N,P),
     triunghi(A,B,0,D,E,0,G,H,I,1,K,L,M,N,P)).
salt(triunghi(0,1,C,1,E,F,G,H,I,J,K,L,M,N,P),
     triunghi(1,0,C,0,E,F,G,H,I,J,K,L,M,N,P)).
salt(triunghi(A,B,C,1,1,0,G,H,I,J,K,L,M,N,P),
     triunghi(A,B,C,0,0,1,G,H,I,J,K,L,M,N,P)).
salt(triunghi(A,B,C,1,E,F,G,1,I,J,K,L,0,N,P),
     triunghi(A,B,C,0,E,F,G,0,I,J,K,L,1,N,P)).
salt(triunghi(A,B,C,1,E,F,1,H,I,J,0,L,M,N,P),
     triunghi(A,B,C,0,E,F,0,H,I,J,1,L,M,N,P)).
salt(triunghi(A,B,C,D,1,F,G,1,I,J,K,0,M,N,P),
     triunghi(A,B,C,D,0,F,G,0,I,J,K,1,M,N,P)).
salt(triunghi(A,B,C,D,1,F,G,H,1,J,K,L,M,0,P),
     triunghi(A,B,C,D,0,F,G,H,0,J,K,L,M,1,P)).
salt(triunghi(0,B,1,D,E,1,G,H,I,J,K,L,M,N,P),
     triunghi(1,B,0,D,E,0,G,H,I,J,K,L,M,N,P)).
salt(triunghi(A,B,C,0,1,1,G,H,I,J,K,L,M,N,P),
     triunghi(A,B,C,1,0,0,G,H,I,J,K,L,M,N,P)).
```

```
salt(triunghi(A,B,C,D,E,1,G,H,1,J,K,L,0,N,P),
     triunghi(A,B,C,D,E,0,G,H,0,J,K,L,1,N,P)).
salt(triunghi(A,B,C,D,E,1,G,H,I,1,K,L,M,N,0),
     triunghi(A,B,C,D,E,0,G,H,I,0,K,L,M,N,1)).
salt(triunghi(A,0,C,1,E,F,1,H,I,J,K,L,M,N,P),
     triunghi(A,1,C,0,E,F,0,H,I,J,K,L,M,N,P)).
salt(triunghi(A,B,C,D,E,F,1,1,0,J,K,L,M,N,P),
     triunghi(A,B,C,D,E,F,0,0,1,J,K,L,M,N,P)).
salt(triunghi(A,B,0,D,1,F,G,1,I,J,K,L,M,N,P),
     triunghi(A,B,1,D,0,F,G,0,I,J,K,L,M,N,P)).
salt(triunghi(A,B,C,D,E,F,G,1,1,0,K,L,M,N,P),
     triunghi(A,B,C,D,E,F,G,0,0,1,K,L,M,N,P)).
salt(triunghi(A,0,C,D,1,F,G,H,1,J,K,L,M,N,P),
     triunghi(A,1,C,D,0,F,G,H,0,J,K,L,M,N,P)).
salt(triunghi(A,B,C,D,E,F,0,1,1,J,K,L,M,N,P),
     triunghi(A,B,C,D,E,F,1,0,0,J,K,L,M,N,P)).
salt(triunghi(A,B,0,D,E,1,G,H,I,1,K,L,M,N,P),
     triunghi(A,B,1,D,E,0,G,H,I,0,K,L,M,N,P)).
salt(triunghi(A,B,C,D,E,F,G,0,1,1,K,L,M,N,P),
     triunghi(A,B,C,D,E,F,G,1,0,0,K,L,M,N,P)).
salt(triunghi(A,B,C,0,E,F,1,H,I,J,1,L,M,N,P),
     triunghi(A,B,C,1,E,F,0,H,I,J,0,L,M,N,P)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,1,1,0,N,P),
     triunghi(A,B,C,D,E,F,G,H,I,J,0,0,1,N,P)).
salt(triunghi(A,B,C,D,0,F,G,1,I,J,K,1,M,N,P),
     triunghi(A,B,C,D,1,F,G,0,I,J,K,0,M,N,P)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,K,1,1,0,P),
     triunghi(A,B,C,D,E,F,G,H,I,J,K,0,0,1,P)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,0,1,1,N,P),
     triunghi(A,B,C,D,E,F,G,H,I,J,1,0,0,N,P)).
salt(triunghi(A,B,C,0,E,F,G,1,I,J,K,L,1,N,P),
     triunghi(A,B,C,1,E,F,G,0,I,J,K,L,0,N,P)).
salt(triunghi(A,B,C,D,E,0,G,H,1,J,K,L,1,N,P),
     triunghi(A,B,C,D,E,1,G,H,0,J,K,L,0,N,P)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,K,L,1,1,0),
     triunghi(A,B,C,D,E,F,G,H,I,J,K,L,0,0,1)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,K,0,1,1,P),
     triunghi(A,B,C,D,E,F,G,H,I,J,K,1,0,0,P)).
salt(triunghi(A,B,C,D,0,F,G,H,1,J,K,L,M,1,P),
     triunghi(A,B,C,D,1,F,G,H,0,J,K,L,M,0,P)).
salt(triunghi(A,B,C,D,E,F,G,H,I,J,K,L,0,1,1),
     triunghi(A,B,C,D,E,F,G,H,I,J,K,L,1,0,0)).
salt(triunghi(A,B,C,D,E,0,G,H,I,1,K,L,M,N,1),
     triunghi(A,B,C,D,E,1,G,H,I,0,K,L,M,N,0)).
```

Rezultate obținute la testare:

```
3 ?- triunghi(1).
13 au mai ramas 13 bete pe tabla
12 ............................................
11
10

                    0
                   0 0
                  0 0 0
                 0 0 0 0
                0 0 1 0 0
apasa orice tasta pt a continua
true

3 ?- triunghi(11).
13 au mai ramas 13 bete pe tabla
12 ............................................
11
10

                    0
                   0 0
                  0 0 1
                 0 0 0 0
                0 0 0 0 0
apasa orice tasta pt a continua
true
```

4. **Problema colorării hărților**.

Se consideră o hartă care cuprinde **n** țări din care unele au graniță comună. Să se coloreze harta utilizând **s** culori (**s < n**) astfel încât oricare două țări cu graniță comună să fie colorate diferit.

Problema constă deci, în colorarea fiecărei regiuni a hărții cu o culoare (dintr-un set de culori) astfel încât două țări vecine să fie colorate diferit (condițiile interne).

Pentru rezolvarea problemei vom utiliza două fișiere:

- Un fișier denumit **Europa.pl** ce conține baza de cunoștințe (numele țărilor și înlănțuirea acestora).
- Un al doilea fișier denumit **harta.pl** ce implementează algoritmul backtracking pentru colorarea hărții.

Codul sursă SWI-Prolog pentru fișierul ***harta.pl*** este următorul:

```
% Program colorare harta
% Gaseste si afiseaza o atribuire de culori regiunilor de
% pe o harta, astfel incat regiunile adiacente sa aiba
% culori diferite
harta :- reconsult('c:/Europa.pl'),
         colorare_harta([], Solutie), afisare(Solutie).

% colorare_harta(SolPartiala, Solutie)
%  Cauta o Solutie de atribuire a culorilor regiunilor
%  de pe harta potrivita, care include atribuirea
%  partiala SolPartiala.
colorare_harta(SolPartiala, Solutie) :-
         tara(Tara), \+ member([Tara,_], SolPartiala),
         culoare(Culoare),
         \+ interzis(Tara,Culoare,SolPartiala),
         write(Tara), nl,
         colorare_harta([[Tara,Culoare]
         |SolPartiala], Solutie).
colorare_harta(Solutie,Solutie).

% interzis(Tara,Culoare,SolPartiala)
%  Tara nu poate fi colorata cu culoarea Culoare daca
%  oricarei regiuni adicente ei ii este deja atribuita
%  culoarea Culoare, in SolPartiala.
interzis(Tara,Culoare,SolPartiala) :-
         granite(Tara,Vecin),
         member([Vecin,Culoare],SolPartiala).

%  granite(Tara,Vecin)
%    Reuseste daca Tara si Vecin au aceeasi granita.
granite(Tara,Vecin) :- alaturate(Tara,Vecin).
granite(Tara,Vecin) :- alaturate(Vecin,Tara).

afisare([]).
afisare( [X|Y] ) :- write(X), nl, afisare(Y).

%Numai patru culori sunt suficiente (intotdeauna).
%Este demonstrat matematic
culoare(rosu).
culoare(albastru).
culoare(verde).
culoare(galben).

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

Fişierul ***Europa.pl*** are următorul conținut:

```
%Date geografice pt. Europa de est, utilizate in harta.pl
tara(romania).          tara(belarus).
tara(moldova).          tara(polonia).
tara(ucraina).          tara(cehia).
tara(ungaria).          tara(slovacia).
tara(serbia).           tara(austria).
tara(bulgaria).         tara(slovenia).
tara(rusia).            tara(croatia).

alaturate(romania,moldova).   alaturate(cehia,polonia).
alaturate(romania,ucraina).   alaturate(cehia,slovacia).
alaturate(romania,ungaria).   alaturate(cehia,austria).
alaturate(romania,serbia).    alaturate(slovacia,ungaria).
alaturate(romania,bulgaria).  alaturate(slovacia,austria).
alaturate(moldova,ucraina).   alaturate(slovenia,austria).
alaturate(ucraina,rusia).     alaturate(slovenia,ungaria).
alaturate(ucraina,belarus).   alaturate(austria,ungaria).
alaturate(ucraina,polonia).   alaturate(croatia,slovenia).
alaturate(ucraina,slovacia).  alaturate(croatia,ungaria).
alaturate(ucraina,ungaria).   alaturate(serbia,ungaria).
alaturate(belarus,rusia).     alaturate(serbia,croatia).
alaturate(belarus,polonia).   alaturate(serbia,bulgaria).
```

Rezultate obținute la testare:

```
3 ?- harta.
[croatia, galben]
[rusia, albastru]
[slovenia, rosu]
[bulgaria, albastru]
[austria, verde]
[serbia, verde]
[slovacia, galben]
[ungaria, albastru]
[cehia, rosu]
[ucraina, verde]
[polonia, albastru]
[moldova, albastru]
[belarus, rosu]
[romania, rosu]
true
```

6.3 Probleme propuse

1. Se vor studia problemele rezolvate (problemele prezentate pe parcursul acestui laborator), încercând găsirea altor posibilități de soluționare a acestora. Utilizați şi alte scopuri (interogări) pentru a testa definițiile predicatelor introduse.
2. Se vor rezolva următoarele probleme propuse şi se va urmări execuția lor corectă.
   - Construiți un labirint care are mai multe soluții şi mai multe căi prin labirint. Definiți o procedură **ceaMaiScurtaCale**, care găseşte cea mai scurtă cale prin labirint.
   - Creați un listing similar celui din fişierul **Europa.pl**, care să se numească **Africa.pl**, în care să introduceți date geografice ale continentului african. Acesta va fi folosit cu aplicația **harta.pl**. Rulați aceasta aplicație cu ajutorul datelor din **Africa.pl** şi verificați corectitudinea rezultatelor obținute.
   - Puteți colora o hartă a Europei sau a Africii, folosind doar trei culori? Explicați. Cum puteți modifica programul **harta.pl**., astfel încât Prolog sa răspundă acestei întrebări?
   - Realizați o aplicație care să găsească soluția pentru un puzzle triunghi cu 10 găuri. Găurile sunt aranjate astfel:

     o
     o o
     o o o
     o o o o

     Precizați dacă există soluții pentru acest tip de machetă de joc.
   - Problema țăranului. Un țăran trebuie să transporte peste un râu *o vulpe, o gâscă* şi *o traistă cu grâu*. Țăranul poate traversa înot râul luând un singur element sau nici unul. Dacă vulpea rămâne pe acelaşi mal cu gâsca şi țăranul este pe celălalt mal, gâsca va fi mâncată de vulpe. Acelaşi lucru se întâmplă şi cu gâsca şi traista de grâu. Scrieți un program Prolog care să caute şi să afişeze toate soluțiile de rezolvare a acestui puzzle. Programul va fi apelat prin interogarea
     ```
     1 ?- taran.
     ```
   - Să se scrie un program în Prolog pentru a rezolva problema "Jocul Vieții". Într-o matrice, fiecare celulă este reprezentată cu 1, dacă este „vie”, şi cu 0, dacă este „moartă”. Regulile sunt următoarele:

- Dacă celula este vie la timpul t, va rămâne vie şi la timpul t+1, dacă şi numai dacă are 2 vecini vii sau 3 vecini vii.
- Dacă celula este moartă la timpul t, va învia la timpul t + 1, dacă şi numai dacă are 3 vecini vii.