Grafuri

1 Obiective

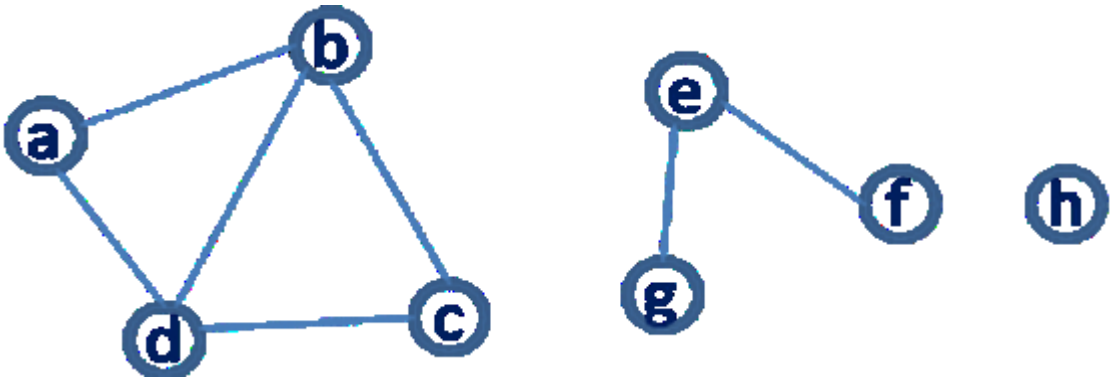
În lucrarea de față vom prezenta cum se poate reprezenta un graf în Prolog și cum putem căuta drumul între două noduri.

2 Considerații teoretice

2.1 Reprezentare

Un graf este reprezentat de două mulțimi: noduri și muchii. Dacă graful este orientat atunci vorbim despre: vârfuri și arce.

Exemplu de graf neorientat:



Există mai multe alternative de a reprezenta un graf în Prolog și le putem clasifica după următoarele criterii:

A. Componentele reprezentate

1. Noduri și muchii;
2. Noduri și lista asociată de vecini;

B. Locul unde sunt salvate

1. Într-o structură;
2. Într-o colecție de predicate de tip axiomă.

A1B2: O colecție de predicate *node* și *edge* (edge-clause)

```
node(a). node(b). %etc
edge(a,b). edge(b,a).
edge(b,c). edge(c,b).
%etc
```

Avem nevoie de predicatele de tip *node* pentru cazurile în care avem noduri izolate.

A2B2: O colecție de predicate *neighbor* (neighbor list-clause)

```
neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
neighbor(h, []).
%etc
```

A1B1: O pereche formată dintr-o listă de noduri și o listă de muchii (graph-term)

```
?- Graph = graph([a,b,c,d,e,f,g,h], [e(a,b), e(b,a), … ]).
```

A2B1: O listă de perechi: nod, listă asociată de vecini (neighbor list-list)

```
?- Graph = [n(a, [b,d]), n(b, [a,c,d]), n(c, [b,d]), n(d, [a,b,c]), n(e,
[f,g]), n(f, [e]), n(g, [e]), n(h, [])].
```

Reprezentarea care va fi folosită depinde de cerințele problemei. Mai jos este dat un exemplu de conversie între A2B2 în A1B2.

```
:-dynamic neighbor/2. % declarăm predicatul dinamic pentru a putea
folosi retract

neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
%etc.

neighb_to_edge:-retract(neighbor(Node,List)),!, %extrage un predicat
neighbor și apoi îl procesează
                process(Node,List),
                neighb_to_edge.
neighb_to_edge. % daca nu mai sunt predicate neighbor înseamnă că am
terminat

% procesarea presupune adăugare de predicate edge și node pentru un
predicat neighbor
process(Node, [H|T]):- assertz(edge(Node, H)), process(Node, T).
process(Node, []):- assertz(node(Node)).
```

2.2 Drumuri în graf

2.2.1 Simple

Presupunem că graful este reprezentat prin predicate *node* și *edge*. Următorul predicat caută un drum între două noduri și returnează lista de noduri parcurse.

```
% path(Source, Target, Path)
path(X,Y,Path):-path(X,Y,[X],Path).   % drumul parțial începe cu
punctul de pornire
path(Y,Y,PPath, PPath).               % când ținta este egală cu
sursa am terminat
path(X,Y,PPath, FPath):-
     edge(X,Z),                       % căutăm o muchie
     not(member(Z, PPath)),           % care nu a mai fost parcursă
     path(Z, Y, [Z|PPath], FPath).    % și o adăugăm în rezultatul
parțial
```

Urmărește execuția la:

```
?- path(a,c,R).
```

2.2.2 Restricționate

Drumul restricționat presupune trecerea printr-un anumit set de noduri în ordinea dată. Deoarece *path* returnează drumul în ordine inversă atunci vom aplica reverse.

```
% restricted_path(Source, Target, RestrictionsList, Path)
restricted_path(X,Y,LR,P):- path(X,Y,P), reverse(P,PR),
     check_restrictions(LR, PR).

% verificăm dacă se respectă restricția
check_restrictions([],_):- !.
check_restrictions([H|T], [H|R]):- !, check_restrictions(T,R).
check_restrictions(T, [H|L]):-check_restrictions(T,L).
```

Urmărește execuția la:

```
?- restricted_path(a, c, [a,c,d], R).
```

2.2.3 Optimale

Considerăm un drum optim între două noduri, acel drum care are lungimea minimă.

```
% optimal_path(Source, Target, Path)
optimal_path(X,Y,Path):-
     asserta(sol_part([],100)),       % distanța maximă inițială
     path(X,Y,[X],Path,1).
optimal_path(_,_,Path):-
     retract(sol_part(Path,_)).

path(Y,Y,Path,Path,LPath):-           % când ținta este egală cu
sursa, salvăm soluția curentă
     retract(sol_part(_,_)),!,
     asserta(sol_part(Path,LPath)),
fail.                                 % și căutăm o altă soluție
```

```
path(X,Y,PPath,FPath,LPath):-
      edge(X,Z),
      not(member(Z,PPath)),
      LPath1 is LPath+1,              % calculăm distanța parțială
      sol_part(_,Lopt),               % extragem distanța de la
soluția precedentă
      LPath1<Lopt,                    % dacă distanța curentă nu
depășește distanța precedentă mergem mai departe
      path(Z,Y,[Z|PPath],FPath,LPath1).
```

Urmărește execuția la:

```
?- optimal_path(a,c,R).
```

2.2.4 Ciclu Hamiltonian

Un ciclu Hamiltonian este un ciclu care trece prin toate nodurile o singură dată (cu excepția nodului de start care este începutul și sfârșitul acestui ciclu).

```
% hamilton(NumOfNodes, Source, Path)

hamilton(NN, X, Path):- NN1 is NN-1, hamilton_path(NN1,X, X,
[X],Path).
```

3 Exerciții

1. Scrieți un predicat care convertește din A1B2 (edge-clause) în A2B2 (neighbor list-clause).
2. Continuă implementarea la ciclul Hamiltonian.
3. Rescrie *optimal_path* folosind ponderea de pe muchie (predicatul *edge* va avea 3 parametrii).
4. Scrie un predicat care găsește un ciclu ce pornește din nodul X și pune rezultatul în R.

```
?- cycle(a, R).
R = [a,b,c,d,a] ;
R = [a,b,d,a] ;
false
```

5. (*) Scrie o serie de predicate care să rezolve problema „Lupul-Capra-Varza”: un fermier trebuie să mute în siguranță, de pe malul nordic pe malul sudic, un lup, o capră și o varză. În barcă încap maxim doi și unul dintre ei trebuie să fie fermierul. Dacă fermierul nu este pe același mal cu lupul și capra, lupul va mânca capra. Același lucru se întâmplă și cu varza și capra, capra va mânca varza.

Sugestii:

a. reprezentați lumea printr-o listă cu poziția celor 4 obiecte [Fermier, Lup, Capra, Varza];
b. starea inițială este [n,n,n,n] (toți sunt în nord) și starea finală este [s,s,s,s] (toți au ajuns în sud);
c. la fiecare trecere fermierul își schimba poziția (din „n” în „s” sau invers) și **cel mult** încă un participant;
d. problema poate fi văzută ca o problema de căutare a drumului într-un graf.