Grafuri

1 Obiective

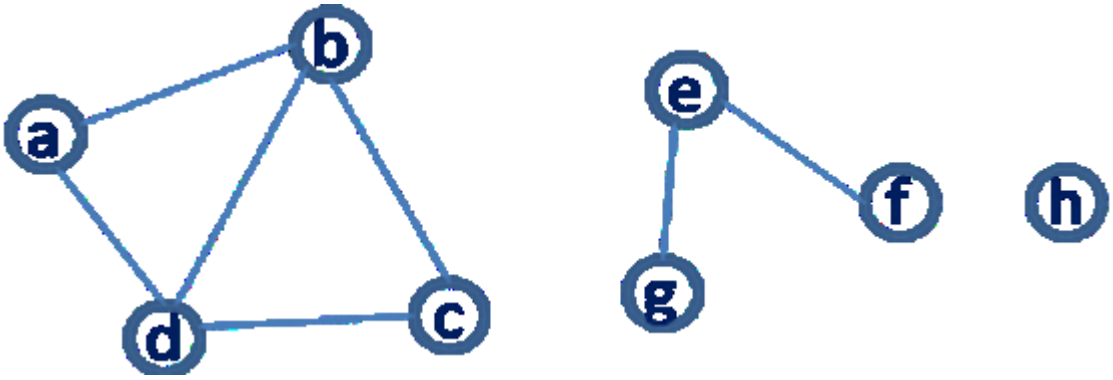
În lucrarea de față vom prezenta cum se poate reprezenta un graf în Prolog și cum putem căuta drumul între două noduri.

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. Tipul de reprezentare:

1. O colecție de noduri și o colecție de muchii
2. O colecție de noduri și lista asociată de vecini

B. Locul unde sunt salvate

In memoria principală, ca un data object (de ex. într-o structură)
In baza de cunoștințe, într-o colecție de predicate de tip axiomă/fapt

În consecință, patru reprezentări principale sunt posibile (alte abordări există, dar pentru scopul acestei lucrări, acestea sunt suficiente) :

A1B2: O colecție de predicate *node* și *edge,* stocate ca fapte (forma 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. Ca o alternativă, nodurile izolate trebuie să fie specificate ca având un nod intre ele si nil: edge(f, nil). Dacă graful este unul neorientat, putem scrie predicatul în modul următor (pentru a evita nevoia de a scrie edge-urile în ambele direcții):

```
is_edge(X,Y):- edge(X,Y); edge(Y,X).
```

A2B2: O colecție de noduri și listele asociate de vecini, stocate în faptele predicatului *neighbor/2* (forma 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, stocat ca un fapt (forma 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, stocat ca un data object (forma 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, [])].
```

Cea mai potrivită reprezentare depinde de problemă. Prin urmare, este convenient să știm modurile prin care putem să transformăm între diferite reprezentări de grafuri. Aici, furnizăm un exemplu de conversie din forma de *neighbor list-clause (A2B2)* în forma de *edge-clause (A1B2)*:

```
% declarăm predicatul dinamic pentru a putea folosi retract
:-dynamic neighbor/2.
% predicatul neighbor este considerat a fiind static deoarece este introdus
% în fișier, doar prin adăugarea declarației de dinamicitate ne este permis
% să folosim operația de retract asupra lui


% un exemplu de graf – prima componentă conexă a grafului
neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
%etc.

neighb_to_edge:-
   %extrage un predicat neighbor
   retract(neighbor(Node,List)),!,
   % și apoi îl procesează
   process(Node,List),
   neighb_to_edge.
neighb_to_edge. % dacă nu mai sunt predicate neighbor/2, ne oprim

% procesarea presupune adăugarea de predicate edge și node
% pentru un predicat neighbor, prima dată adăugăm muchiile
% până când lista devine vidă iar abia apoi predicatele de tip node
process(Node, [H|T]):- assertz(gen_edge(Node, H)), process(Node, T).
process(Node, []):- assertz(gen_node(Node)).
```

Inițial graful este salvat in baza de date a predicatelor. Predicatul *neighb_to_edge* citește o clauză a predicatului *neighbor* la un moment dat și proceseaza informația din fiecare clauză separat.

Predicatul *process/2* traversează lista de vecini (al doilea argument) a nodului curent (primul argument) și adaugă (prin assert) un nou fapt la predicatul *gen_edge* pentru fiecare vecin nou al nodului curent. La condiția de oprire(a doua clauză), ne oprim când lista de vecini devine vidă și facem assert la un nou fapt de tip *gen_node*.

Predicatul *neighb_to_edge* se folosește de recursivitate și retract pentru a traversa secvențial clauzele predicatului neighbor și se oprește la un moment dat când niciun neighbor nu mai poate fi găsit. Fără folosirea operației de retract, această implementare va rezulta într-o buclă infinită a primului fapt al

predicatului *neighbor*. În acest punct, *neighb_to_edge* ajunge la a doua clauză unde se oprește (predicatul neavând argumente, nici 'condiția de oprire' nu are – este nevoie de această clauză pentru a evalua predicatul ca true, altfel ultimul retract va da fail și va rezulta în false.

Pentru a pune întrebări acestui predicat, avem sintaxa următoare:
?- neighb_to_edge.
true;
false.

După cum putem observa, este insuficient să verificăm rezultatul. Două predicate predefinite adiționale trebuie folosite pentru a putea verifica. Primul predicat este cel de *retractall/1*, acesta ne garantează că predicatele stocate sunt doar din acestă execuție prin ștergerea tuturor clauzelor unui predicat dat. Al doilea predicat este cel de *listing/1*, care ne permite să afișăm toate clauzele unui predicat dat. Fiecare din acestea necesită propriul format al argumentului. Întrebarea devine:

?- retractall(gen_edge(_,_)), neighb_to_edge, listing(gen_edge/2).

```
:- dynamic gen_edge/2.


gen_edge(a, b).
gen_edge(a, d).
gen_edge(b, a).
gen_edge(b, c).
gen_edge(b, d).
gen_edge(c, b).
gen_edge(c, d).
```

true;
false.

2.1.1 Implementare alternativă

Predicatul anterior, *neighb_to_edge*, poate fi implementat într-un mod echivalent prin utilizarea unui failure-driven loop (laboratorul anterior, efecte laterale). Predicatul *process/2* va rămâne același.

```
neighb_to_edge_v2:-
   neighbor(Node,List), % accesăm faptul
   process(Node,List),
   fail.
neighb_to_edge_v2.
```

Parcugerea predicatului nu se schimbă și nu mai necesită declararea predicatului *neighbor* ca fiind dinamic, deoarece nu este necesară folosirea retract-ului. Citește argumentele fiecărei clauze a predicatului *neighbor* și continuă cu procesarea fiecăruia. Prin folosirea fail-ului, backtracking-ul este activat și o nouă clauză a neighbor este citit până când nu rămâne niciunul necitit. În acest punct, se ajunge la a doua clauză a *neighb_to_edge* și se oprește.

Cu toate acestea, prima implementare (folosind recursivitatea) distruge baza predicatului *neighbor/2* din cauza *retract*-ului. Astfel, a doua implementare este de preferat.

2.2 Drumuri în graf

Știind reprezentările grafurilor, vom trece la traversarea lor. Vom începe cu un drum simplu între două noduri și vom ajunge la drumuri restricționate, drumuri optime și, în final, la ciclul Hamiltonian al unui graf.

2.2.1 Drum Simplu

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)
% drumul parțial începe cu nodul sursă – acesta este un wrapper
path(X, Y, Path):-path(X, Y, [X], Path).

% când sursa (primul argument) este egal cu destinația (al doilea argument),
% atunci știm că drumul a ajuns la final
% și putem unifica drumul parțial cu cel final
path(Y, Y, PPath, PPath).
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).

Puteți folosi exemplul de graf sau chiar să adăugați niște muchii la acesta. Ce se întâmplă când repetăm întrebarea?

2.2.2 Drum Restricționat

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).
```

Predicatul `restricted_path/4` caută un drum între nodul sursă și destinație și verifică dacă acest drum satisface restricțiile specificate în `LR` (i.e. dacă trece printr-o secvență de noduri, specificată în `LR`), folosind predicatul `check_restrictions/2`, care face defapt verificarea.

Predicatul `check_restrictions/2` traversează lista de restricții (primul argument) și lista care reprezintă drumul (al doilea argument) simultan, cât timp primele elemente din cele două liste coincid (clauza 2). La momentul în care sunt diferite, vom continua doar cu a doua listă (clauza 3). Predicatul reușește când prima listă devine vidă (clauza 1).

Întrebare: Ce se întâmplă dacă mutăm condiția de oprire ca ultima clauză? Avem nevoie de tăierea de backtracking în condiția de oprire?

Urmărește execuția la:
?- check_restrictions([2,3], [1,2,3,4]).
?- check_restrictions([1,3], [1,2,3,4]).

```
?- check_restrictions([1,3], [1,2]).
?- restricted_path(a, c, [a,c,d], R).
```

2.2.3 Drum Optim

Considerăm un drum optim între două noduri într-un graf ca acel drum care are lungimea minimă. O abordare de a găsi drumul optim este de a genera toate drumurile prin backtracking și selecția drumului optim. Evident, această abordare este ineficientă. În loc să facem asta, în timpul procesului de backtracking, vom reține soluția optimă parțială folosind efecte laterale (în baza de predicate) și o vom actualiza când o soluție mai bună este găsită:

```
:- dynamic sol_part/2.

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

% path(Source, Target, PartialPath, FinalPath, PathLength)
% când ținta este egală cu sursa, salvăm soluția curentă
path(Y,Y,Path,Path,LPath):-
    % scoatem ultima soluție
    retract(sol_part(_,_)),!,
    % salvăm soluția curentă
    asserta(sol_part(Path,LPath)),
    % căutăm o altă soluție
    fail.
path(X,Y,PPath,FPath,LPath):-
    edge(X,Z),
    not(member(Z,PPath)),
    % calculăm distanța parțială
    LPath1 is LPath+1,
    % extragem distanța de la soluția precedentă
    sol_part(_,Lopt),
    % dacă distanța curentă nu depășește distanța precedentă
    LPath1<Lopt,
    % mergem mai departe
    path(Z,Y,[Z|PPath],FPath,LPath1).
```

Predicatul *path/4* generează, prin backtracking, toate drumurile care sunt mai bune decât soluția curentă parțială și actualizează soluția curentă parțială când un drum mai scurt este găsit. Când o soluție mai bună decât cea curentă este găsită, predicatul înlocuiește soluția optimă veche în baza de predicate (clauza 1) și după continuă căutarea, prin pornirea mecanismului de backtracking (folosind fail).

Urmărește execuția la:
?- optimal_path(a,c,R).

Rețineți. Când folosim assert/retract, trebuie să ne asigurăm că "curățăm", trebuie să verificăm că nicio clauză nedorită nu rămâne stocată în baza de predicate după execuția întrebărilor (nu sunt afectate de backtracking!).

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). Evident, nu toate grafurile conțin un astfel de ciclu. Predicatul *hamilton/3* este prezentat mai jos:

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

Predicatul *hamilton_path/5* vă rămâne de implementat. Predicatul acesta ar trebui să caute un drum închis începând din X, de lungime NN1 (numărul de noduri din graf, minus 1).

Urmăriți execuția predicatului *hamilton/3* folosind graful dat ca exemplu.

3 Exerciții

1. Scrieți un predicat care convertește din A1B2 (edge-clause) în A2B2 (neighbor list-clause).

```
edge_ex1(a,b).
edge_ex1(a,c).
edge_ex1(b,d).
```

?- retractall(gen_neighb_list(_,_)), edge_to_neighb, listing(gen_neighb_list/2).

```
:- dynamic gen_neighb_list/2.


gen_neighb_list(c, []).
gen_neighb_list(d, []).
gen_neighb_list(a, [c, b]).
gen_neighb_list(b, [d]).
```

true

2. Continuă implementarea la ciclul Hamiltonian folosind predicatul *hamilton/3*.

```
edge_ex2(a,b).
edge_ex2(b,c).
edge_ex2(a,c).
edge_ex2(c,d).
edge_ex2(b,d).
edge_ex2(d,e).
edge_ex2(e,a).
```

?- hamilton(5, a, P).
P = [a, e, d, c, b, a]

3. Predicatul *restricted_path/4* găsește un drum între nodul sursă și cel destinație și verifică dacă drumul găsit conține nodurile din lista de restricții. Acest predicat folosește recursivitate înainte, ordinea nodurilor trebuie inversată în ambele liste – lista de drum și de restricții. Motivați de ce această

strategie nu este eficientă (urmăriți ce se întâmplă). Scrieți un predicat mai eficient care caută un drum restricționat între nodul sursă și cel destinație.

```
edge_ex3(a,b).
edge_ex3(b,c).
edge_ex3(a,c).
edge_ex3(c,d).
edge_ex3(b,d).
edge_ex3(d,e).
edge_ex3(e,a).
```

```
? - restricted_path_efficient(a, e, [c,d], P2).
P = [a, b, c, d, e];
P = [a, c, d, e];
false
```

4. Rescrie *optimal_path/3* astfel încât să funcționeze pe grafuri ponderate: atașați o pondere pentru fiecare muchie din graf și calculați drumul de cost minim dintr-un nod sursă la un nod destinație.

   Predicatul *edge* va avea 3 parametrii.

```
edge_ex4(a,c,7).
edge_ex4(a,b,10).
edge_ex4(c,d,3).
edge_ex4(b,e,1).
edge_ex4(d,e,2).
```

```
?- optimal_weighted_path(a, e, P).
P = [e, b, a]
```

5. Scrie predicatul *cycle(X,R)* care găsește un ciclu ce pornește din nodul *X* dintr-un graf G (folosind reprezentarea *edge/2*) și pune rezultatul în *R*. Predicatul trebuie să returneze toate ciclurile prin backtracking.

```
edge_ex5(a,b).
edge_ex5(a,c).
edge_ex5(c,e).
edge_ex5(e,a).
```

```
edge_ex5(b,d).
edge_ex5(d,a).
```

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

6. Scrieți predicatul *cycle(X,R)* din exercițiul anterior folosind reprezentarea *neighbour/2*.

```
neighb_ex6(a, [b,c]).
neighb_ex6(b, [d]).
neighb_ex6(c, [e]).
neighb_ex6(d, [a]).
neighb_ex6(e, [a]).
```

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

7. Scrieți predicatul *euler/3* care poate să găsească drumuri Euleriane într-un graf dat de la un nod dat.

```
% euler(NE, S, R). – unde S este nodul sursă
% și NE este numărul de muchii din graf
edge_ex7(a,b).
edge_ex7(b,e).
edge_ex7(c,a).
edge_ex7(d,c).
edge_ex7(e,d).
```

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

8. 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:

- Puteți alege să encodați spațiul de stări ca instanțe a unei configurări ale celor 4 obiecte (Fermir, Lup, Capră, Varză): reprezentate ca o listă cu poziția celor 4 obiecte [Fermier, Lup, Capră, Varză] sau ca o structură complexă (ex. F-W-G-C, sau state(F,W,G,C)).
- starea inițială este [n,n,n,n] (toți sunt în nord) și starea finală este [s,s,s,s] (toți au ajuns în sud); pentru reprezentarea folosind liste a stărilor (ex. dacă Fermierul trece lupul -> [s,s,n,n], această stare nu ar trebui să fie validă deoarece capra mănâncă varza).
- la fiecare trecere Fermierul își schimba poziția (din „n” în „s” sau invers) și **cel mult** încă un participant (Lupul, Capra, Varza).
- problema poate fi văzută ca o problema de căutare a drumului într-un graf.