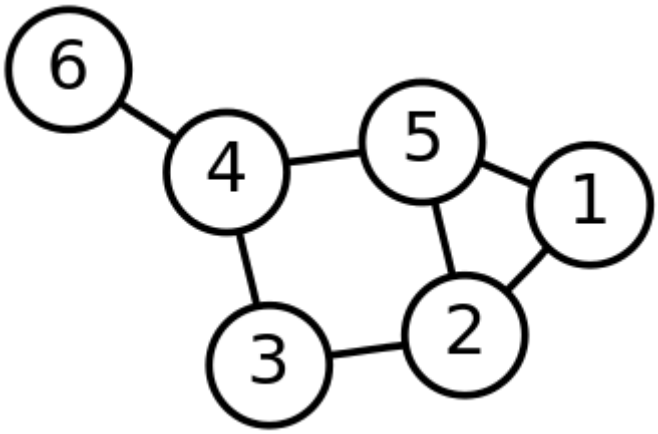
Algoritmi de traversare a grafurilor (DFS & BFS)

1 Obiective

În lucrarea de față vom prezenta cum se pot parcurge grafurile în Prolog. Grafurile le vom reprezenta prin predicate edge.

2 Considerații teoretice

Exemplu de graf neorientat:



În Prolog:
edge(1,5).
edge(1,2). %etc (va trebui să îl completăm)

2.1 Parcurgere în adâncime (DFS)

Ca în lucrarea anterioară, vom folosi forma *edge-clause* de prezentare a grafurilor. Deoarece parcurgerea în adâncime este un mecanism al Prolog-ului, toate predicatele drumurilor din lucrarea anterioară deja folosesc o strategie DFS de căutare. Mai jos aveți predicatul care implementează căutarea DFS dintr-un nod sursă (prin explorarea componentelor conexe a nodului sursă).

Ne vom folosi de un predicat auxiliar pentru a stoca nodurile deja vizitate.

```
:- dynamic nod_vizitat/1.

% d_search(Source, Path)
dfs(X,_) :- df_search(X). % parcurgerea nodurilor
```

```
% când parcurgerea se termină, începe colectarea
dfs(_,L) :- !, collect_reverse([], L). % colectarea rezultatelor

% predicatul de traversare
df_search(X):-
   % salvăm X ca nod vizitat
   asserta(nod_vizitat(X)),
   % luăm un prim edge de la X la un Y
   % restul le vom găsi prin backtracking
   edge(X,Y),
   % verificăm daca acest Y a fost deja vizitat
   not(nod_vizitat(Y)),
   % dacă nu a fost -  de aceea avem nevoie de negare –
   % atunci vom continua parcurgerea prin mutarea nodululi curent la Y
   df_search(Y).

% predicatul de colectare - colectarea se face în ordine inversă
collect_reverse(L, P):-
   % scoatem fiecare nod vizitat
   retract(nod_vizitat(X)), !,
   % îl adăugăm la lista ca primul element
   % astfel vor aparea în ordine inversă
   collect_reverse([X|L], P).
   % unificăm primul si al doilea argument,
   % rezultatul va fi în al doilea argument
collect_reverse(L,L).
```

Urmărește execuția la:
?- dfs(1,R).

2.2 Parcurgerea în lățime (BFS)

Strategia BFS se folosește de o coadă pentru a reține ordinea de expansiune a nodurilor. În fiecare pas, un nou nod este citit din coadă și expandat – toți vecinii nevizitați vor fi adăugați în coadă.

Ne vom folosi de un predicat dinamic pentru a reține nodurile de expandat. Coada este „implementată” prin modul în care inserăm și extragem acest noduri vizitate.

```
:- dynamic nod_vizitat/1.
:- dynamic coada/1.          % coada reține nodurile care trebuie expandate

% b_search(Source, Path)
bfs(X, _):- % parcurgerea nodurilor
   assertz(nod_vizitat(X)), % adăugăm sursa ca nod vizitat
   assertz(coada(X)), % adăugăm sursa în coadă
   bf_search.
bfs(_,R):- !, collect_reverse([],R). % colectarea rezultatelor

bf_search:-
   retract(coada(X)), % scoatem nodul care trebuie expandat
   expand(X), !, % apelăm predicatul de expansiune
   bf_search. % recursivitate

expand(X):-
   edge(X,Y), % găsim un nod Y cu o muchie la X-ul dat
   not(nod_vizitat(Y)), % verificăm daca Y a fost vizitat
   asserta(nod_vizitat(Y)), % adăugăm Y la nodurile vizitate
   assertz(coada(Y)), % adăugam Y în coadă pentru a fi expandat
   % la un moment dat
   fail. % fail-ul este necesar pentru a găsi un alt Y
expand(_).
```

Urmărește execuția la:
?- bfs(1,R).

2.3 Best-First Search

Algoritmul *Best-first search* este o strategie de căutare greedy, care se folosește de o euristică pentru a estima costul unui drum de la nodul curent la nodul target. În fiecare pas, algoritmul selecționează nodul cu distanța estimată cea mai mică până la nodul target (folosind funcția euristică).

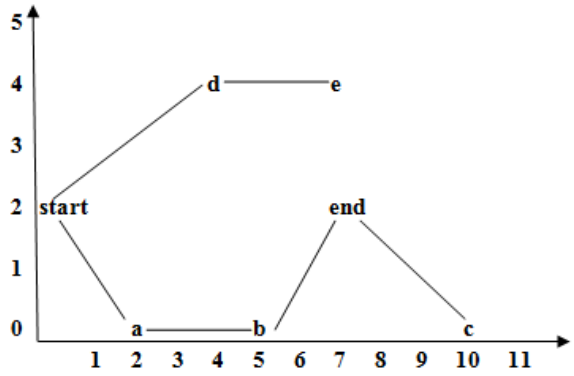


Figura 1: Un exemplu de graf pentru algoritmul best-first search

Graful de mai sus poate fi reprezentat folosind o variație a formei neighbor list-clause:

```
pos_vec(start,0,2,[a,d]).
pos_vec(a,2,0,[start,b]).
pos_vec(b,5,0,[a,c, end]).
pos_vec(c,10,0,[b, end]).
pos_vec(d,3,4,[start,e]).
pos_vec(e,7,4,[d]).
pos_vec(end,7,2,[b,c]).

is_target(end).
```

Nodul de final este specificat ca nodul target, folosind o clauză de predicat. Specificațiile predicatului sunt prezentate mai jos:

```
best([], []):-!.
best([[Target|Rest]|_], [Target|Rest]):- is_target(Target),!.
best([[H|T]|Rest], Best):-
        pos_vec(H,_,_, Neighb),
        expand(Neighb, [H|T], Rest, Exp),
        quick_sort(Exp, SortExp, []),
        best(SortExp, Best).

% Bazat pe calea curentă (al doilea argument), predicatul expand/4
% caută prin vecinii ultimului nod expandat (primul argument)
expand([],_,Exp,Exp):- !.
expand([H|T],Path,Rest,Exp):-
        \+(member(H,Path)), !, expand(T,Path,[[H|Path]|Rest],Exp).
expand([_|T],Path,Rest,Exp):- expand(T,Path,Rest,Exp).
```

```
% Predicatul quick_sort/3 utilizează liste diferență
quick_sort([H|T],S,E):-
        partition(H,T,A,B),
        quick_sort(A,S,[H|Y]),
        quick_sort(B,Y,E).
quick_sort([],S,S).

% În acest caz, predicatul partition/4 folosește un predicat auxiliar
% order/2 care definește modul de a partiționa ca fiind
% bazat pe distanțte
partition(H,[A|X],[A|Y],Z):- order(A,H), !, partition(H,X,Y,Z).
partition(H,[A|X],Y,[A|Z]):- partition(H,X,Y,Z).
partition(_,[],[],[]).

% predicat care calculează distanța între două noduri
dist(Node1,Node2,Dist):-
        pos_vec(Node1, X1, Y1, _),
        pos_vec(Node2, X2, Y2, _),
        Dist is (X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2).

% predicatul order/2 bazat pe distanțe folosit in partition/4
order([Node1|_],[Node2|_]):-
        is_target(Target),
        dist(Node1,Target,Dist1),
        dist(Node2,Target,Dist2),
        Dist1<Dist2.
```

Urmărește execuția la:

?- best([[start]], Best).

3 Exerciții

1. Modificați predicatul DFS astfel încât să caute noduri numai până la o anumită adâncime (DLS – Depth-Limited Search). Setați limita de adâncime printr-un predicat, depth_max(2). de exemplu.

```
edge_ex1(a,b).
edge_ex1(a,c).
edge_ex1(b,d).
edge_ex1(d,e).
edge_ex1(c,f).
edge_ex1(e,g).
edge_ex1(f,h).
```

?- d_search(a,DFS), dl_search(a, DLS).
DFS = [a, b, d, e, g, c, f, h],
DLS = [a, b, d, c, f].