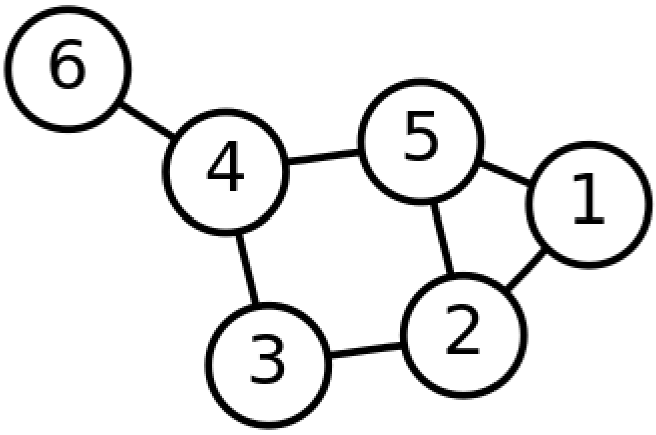
Graphs Search Algorithms (DFS & BFS)

1 Objectives

In this lesson, we will present how graph search algorithms can be implemented in Prolog. We will represent graphs using the edge-clause.

2 Theoretical Considerations

Example of undirected graph:



In Prolog:

Code
```
edge(1,2).
edge(1,5).
edge(2,3).
edge(2,5).
edge(3,4).
edge(4,5).
edge(4,6).

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

2.1 Depth-First Search (DFS)

As in the previous session, we shall employ the *edge-clause* form for graph representation. Since depth-first search is the mechanism employed by the Prolog engine, all path predicates from the previous session employed a DFS search strategy. Below you have the predicates which implement a DFS search from a source node (by exploring the connected component of the source node). This implementation is based upon the **Two-stage process** concepts discussed in previous laboratory sessions.

We will use an auxiliary predicate to store already visited nodes.

Code

```
:- dynamic visited_node/1.

% dfs(Source, Path)
dfs(G,X,_) :- df_search(G,X). % stage1. traversal of nodes
% when finished, collection starts
dfs(_,_,L) :- !, collect(L).       % stage2. collecting results

% traversal predicate
df_search(G,X):-
   % store X as visited node
   assertz(visited_node(X)),
   % take a first edge from X to Y, rest are found through backtracking
   Edge=..[G,X,Y], call(Edge),
   % check if this Y was already visited
   not(visited_node(Y)),
   % if it was not -this is why the negation is needed -
   % then we continue the traversal by moving the current node to Y
   df_search(G,Y).

% collecting predicate - collecting is done in reverse order
collect([X|R]):-
   % we retract each stored visited node
   retract(visited_node(X)), !,
   collect(R).
% the result is constructed backwards
collect([]).
```

Follow the execution of:

Query

```
?- trace, dfs(is_edge,1,R).
R = [1, 2, 3, 4, 5, 6].
```

2.2 Breadth-First Search (BFS)

The *BFS* strategy employs a queue to keep track of the expansion order of the nodes. In each step, a new node is read from the queue and is expanded – i.e. all its unvisited neighbors are added to the queue, in turn.

We will use a dynamic predicate to store the nodes to expand. The queue is „implemented" through the manner in which we insert (*assert*) and extract (*retract*) these visited nodes. This implementation is based upon the **Two-stage process** concepts discussed in previous laboratory sessions.

Code

```
:- dynamic visited_node/1.
:- dynamic queue/1.  % the queue stores nodes that need to be expanded

% bfs(Source, Path)
bfs(G,X,_):-  % stage1. traversal of nodes
   assertz(visited_node(X)), % add source as visited
   assertz(queue(X)), % add source as first element in queue
   bf_search(G).
bfs(_,_,R):- !, collect(R). % stage2. collecting results (same as previous)

bf_search(G):-
   retract(queue(X)), %retract node that needs to be expanded
   expand(G,X), !, % call expand predicate
   bf_search(G). % recursion

expand(G,X):-
   Edge=..[G,X,Y], call(Edge), % find a node Y linked to given X
   not(visited_node(Y)), % check if Y was already visited
   % if Y was not visited before
   assertz(visited_node(Y)), % add Y to visited nodes
   assertz(queue(Y)), % add Y to queue to be expanded
   % at some point
   fail. % fail required to find another Y
expand(_,_).
```

Follow the execution of:

Query

```
?- trace, bfs(is_edge,1,R).
R = [1, 2, 5, 3, 4, 6].
```

But the BFS algorithm can also be implemented without using side effects. In this implementation, we will use the *neighbor/2* representation for the graph and create the queue (Q) as a list, and we will keep a list of visited nodes (V).

Code

```
neighbor(1, [2,5]).
neighbor(2, [1,3,5]).
neighbor(3, [2,4]).
neighbor(4, [3,5,6]).
neighbor(5, [1,2,4]).
neighbor(6, [4]).

bfs1(G, X, R) :-
   bfs1(G, [X], [], R).

bfs1(_, [], _, []).
bfs1(G, [X|Q], V, [X|R]):-
   \+member(X, V),!,
   Neighb =.. [G,X,Ns], call(Neighb),        % Neighb = G(X, Ns)
   remove_visited(Ns, V, RemNs),
   append(Q, RemNs, NewQ),
   bfs1(G, NewQ, [X|V], R).
bfs1(G, [_|Q], V, R):-
   bfs1(G, Q, V, R).

remove_visited([], _, []).
remove_visited([H|T], V, [H|R]):- \+member(H, V), !, remove_visited(T, V, R).
remove_visited([_|T], V, R):- remove_visited(T, V, R).
```

Follow the execution of:

Query

```
?- trace, bfs1(neighbor,1,R).
R = [1, 2, 5, 3, 4, 6].
```

2.3 Best-First Search

The *Best-first search* algorithm is an informed greedy search strategy, in that it employs a heuristic to estimate the cost of the path from the current node to the target node. In each step, the algorithm selects the node having the smallest estimated distance to the target node (via the heuristic function).

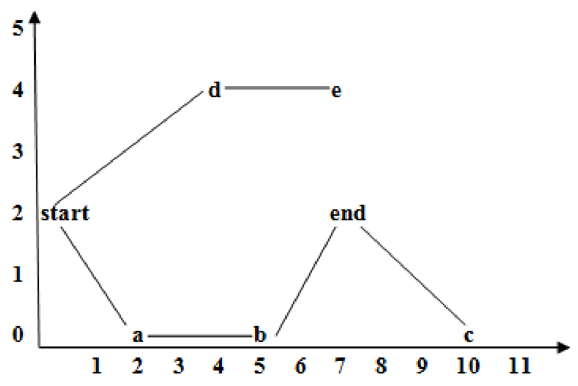


Figure 1: An example graph for the best-first search algorithm

The graph above can be represented using a variation of the neighbor list-clause form, as:

Code

```
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).
```

The end node is specified as being the target node, using a predicate clause. The predicate specifications are presented below:

Code

```
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).

% Based on the current path (second argument), the expand/4 predicate
% searches for neighbours of the last expansion (first 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).
```

```
% The quick_sort/3 predicate uses difference lists
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).

% In this case, the partition/4 predicate uses an auxiliary predicate
% order/2 that defines how the partition should be made
% based on distances
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(_,[],[],[]).

% predicate that calculates the distance between two nodes
dist(Node1,Node2,Dist):-
pos_vec(Node1, X1, Y1, _),
pos_vec(Node2, X2, Y2, _),
        Dist is (X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2).

% the order/2 predicate based on distances used in partition/4
order([Node1|_],[Node2|_]):-
is_target(Target),
        dist(Node1,Target,Dist1),
        dist(Node2,Target,Dist2),
        Dist1<Dist2.
```

Follow the execution of:

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

3 Exercises

1. Modify the DFS predicate such that it searches nodes only to a given depth (DLS – Depth-Limited Search). Set the depth limit via a predicate, `depth_max(2).` for example.

Code
```
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).
```

Query
```
?- dfs(edge_ex1,a,DFS), dls(edge_ex1,a,DLS).
DFS = [a, b, d, e, g, c, f, h],
DLS = [a, b, d, c, f].
```

2. Having the BFS algorithm implementation *without side effects*, modify it *without side effects* such that it works on the *edge* representation instead of the *neighbor* representation.

Query
```
?- bfs2(is_edge,a, R).
R = [1, 2, 5, 3, 4, 6].
```

3. Write the DFS algorithm implementations *without side effects* for the *edge* and *neighbor* representations.

Query
```
?- dfs1(neighbor,1,R).
R = [1, 2, 3, 4, 5, 6].

?- dfs2(is_edge,1,R).
R = [1, 2, 3, 4, 5, 6].
```