Graphs

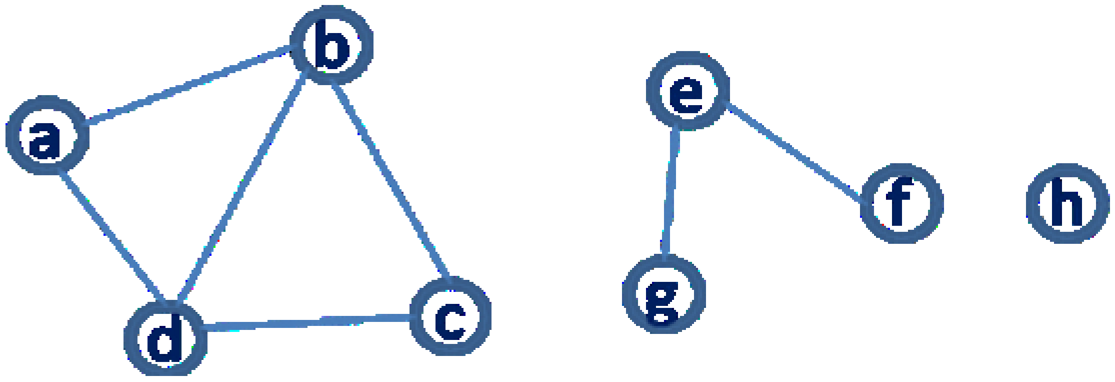
1 Theoretical Considerations

In this lesson, we will present how a graph can be represented in Prolog and how we can search for a path between two nodes.

1.1 Representation

A graph is given by a set of vertices and a set of edges (if the graph is undirected, or arcs, if the graph is directed):

Example of undirected graph:



Several alternatives are available for representing graphs in Prolog, and they can be categorized according to the following criteria:

A. Representation type:

1. As a collection of vertices and a collection of edges
2. As a collection of vertices and associated neighbor list

B. Where you store the graph:

1. In the main memory, as a data object (for example, a structure)
2. In the predicate base, as a set of predicate facts

Consequently, four main representations are possible (other approaches may exist, but for the purpose of the current course these are enough):

A1B2: As a set of nodes and edges, stored as predicate facts (edge-clause form):

```
Code
node(a).
node(b).
%etc
```

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

The node predicate is required for cases of isolated nodes. As an alternative, isolated nodes have to be specified as having an edge between them and nil: edge(f, nil). If your graph is undirected, you may write a predicate such as the following (to avoid having to write the edges in both directions):

Code

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

A2B2: As a set of vertices and associated neighbor lists, stored as *neighbour/2* predicate facts (neighbor list-clause form):

Code

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

A1B1: As the set of vertices and the set of edges, stored as a fact (graph-term form):

Query

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

A2B1: As a set of vertices and associated neighbor lists, stored as a data object (neighbor list-list form):

Query

```
?- 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, [])].
```

The most suitable representation to use is highly dependent on the problem at hand. Therefore, it is convenient to know how to perform conversions between different graph representations. Here, we provide an example conversion from the *neighbor list-clause (A2B2)* form to the *edge-clause (A1B2)* form:

Code

```
% we declare the predicate as dynamic to be able to use retract
:-dynamic neighbor/2.
% as the neighbour predicate is introduced in the file,
% it is considered static only through the introduction of the
% dynamic declaration can we use the retract operation on it

% an example graph - 1st connected component of the example graph
neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
%etc.

neighb_to_edge:-
   % extract the first neighbour predicate
   retract(neighbor(Node,List)),!,
   % and process it
   process(Node,List),
   neighb_to_edge.
neighb_to_edge. % if no more neighbor/2 predicates remain then we stop

% processing presumes the addition of edge & node predicates
% for each neighbour predicate, we first add the edges
% until the list is empty and then add the node predicate
process(Node, [H|T]):- assertz(gen_edge(Node, H)), process(Node, T).
process(Node, []):- assertz(gen_node(Node)).
```

The graph is initially stored in the predicate database. Predicate *neighb_to_edge* reads one clause of predicate *neighbor* at a time, and processes the information of each clause separately.

The *process/2* predicate traverses the neighbor list (second argument) of the current node (first argument), and asserts sequentially a new fact for predicate *gen_edge* for every new neighbor of the current node through the use of recursion. At the stopping condition (second clause of *process*), we stop when the list of neighbours has been emptied and we assert a new fact for the *gen_node* predicate.

The *neighb_to_edge* predicate uses recursion and retract to traverse the clauses of the neighbor predicate sequentially and it eventually stops when no more neighbours can be found. Without the use of retract, this implementation will result in an infinite loop of the first fact of the *neighbor* predicate. At this point, it enters the second clause where it stops (as the predicate has no arguments, neither does the 'stopping condition' – this clause is required to evaluate the predicate as true, otherwise the last retract will fail and it will return false).

The querying of this predicate is rather simple:

Query

```
?- neighb_to_edge.
true;
false.
```

But as we can observe, it is rather insufficient to verify the correctness of the result. Two additional built-in predicates must be used in the query in order to verify the result. The first is the *retractall* predicate, it guarantees that the predicates stored are only from this run by removing all previous clauses of a given predicate. The second is the listing *predicate*, that allows us to show all clauses of a given predicate. Each of these predicates requires its own format of the argument. The query will become:

Query

```
?- 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.
```

There is a disadvantage to this approach. Due to the use of *retract*, the *neighbor* predicate will be removed entirely from the predicate base and cannot be used in the same execution instance. Check the following query:

Query

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

1.1.1 Alternative implementations (Failure driven loop vs recursion)

The above predicate, *neighb_to_edge*, can be implemented in a second way through the use of a failure-driven loop (the previous laboratory, side effects). This implementation has the advantage of not removing the facts of the *neighbor* predicate during its execution. The *process/2* predicate remains unchanged.

Code

```
neighb_to_edge_v2:-
   neighbor(Node,List), % access the fact
   process(Node,List),
   fail.
neighb_to_edge_v2.
```

The overall flow of the predicate does not change and it does not require the dynamic declaration of the *neighbor* predicate anymore, as the retract operation is not needed. It reads the arguments of each clause of the neighbor predicate and continues by the processing of each of these. Through the use of the *fail*, backtracking is activated and a new *neighbor* clause is read until no unread neighbor clauses remain. At this point it enters the second clause of *neighb_to_edge* and stops.

The first implementation (using recursion) does destroy the predicate base of the *neighbor/2* predicate due to the *retract*. Thus, the second implementation is preferred. However, if we want to use recursion without *retract,* a third implementation can be used, but it requires a *seen* predicate to verify which facts of the *neighbor* have been traversed in order to not enter an infinite loop.

Code

```
:-dynamic seen/1.

neighb_to_edge_v3:-
   neighbor(Node,List),
   not(seen(Node)),!,
   assert(seen(Node)),
   process(Node,List),
   neighb_to_edge_v3.
neighb_to_edge_v3.
```

1.2 Paths in Graphs

Having covered the issue of graph representation, let us address the graph traversal problem. We shall start with the simple path between two nodes, and progress to the restricted path between two nodes, the optimal path between two nodes and, finally, the Hamiltonian cycle of a graph.

1.2.1 Simple Path

We assume the graph is represented in the edge-clause form. A predicate which searches for a path between two nodes in a graph and returns a list of traversed nodes is presented below.

Code

```
% path(Source, Target, Path)
% the partial path starts with the source node – it is a wrapper
path(X, Y, Path):-path(X, Y, [X], Path).

% when source (1st argument) is equal to target (2nd argument),
% we finished the path and we unify the partial and final paths.
```

```
path(Y, Y, PPath, PPath).
path(X, Y, PPath, FPath):-
    edge(X, Z),                     % search for an edge
    not(member(Z, PPath)),          % that was not traversed
    path(Z, Y, [Z|PPath], FPath).   % add to partial result
```

Follow the execution of:

Query

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

You may use the example graph, perhaps add some edges to it. What happens when you repeat the question?

1.2.2 Restricted Path

The retricted path must pass through certain nodes, in a certain order (these nodes are specified in a list). Because *path* returns the path in reverse order we will apply the *reverse* predicate.

Code

```
% 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|TR], [H|TP]):- !, check_restrictions(TR,TP).
check_restrictions(LR, [_|TP]):-check_restrictions(LR,TP).
```

The predicate restricted_path/4 searches for a path between the source and the destination nodes, and then checks if that path satisfies the restrictions specified in LR (i.e. passes through a certain sequence of nodes, specified in LR), using predicate check_restrictions/2, which performs the actual check.

The check_restrictions/2 predicate traverses the restrictions list (the first argument) and the list representing the path (the second argument) simultaneously, so long as their heads coincide (clause 2). When the heads do not match, we advance in the second list only (clause 3). The predicate succeeds when the first list becomes empty (clause 1).

Question: What happens if we move the stopping condition as last clause? Do we need the "!" in the stopping condition?

Follow the execution of:

Query

```
?- trace, check_restrictions([2,3], [1,2,3,4]).
?- trace, check_restrictions([1,3], [1,2,3,4]).
?- trace, check_restrictions([1,3], [1,2]).
?- trace, restricted_path(a, c, [a,c,d], R).
```

1.2.3 Optimal Path

We consider the optimal path between the source and the target node in a graph as the path containing the minimum number of nodes. One approach to find the optimal path is to generate all paths via backtracking and then select the optimal path. Of course, this is extremely inefficient. Instead, during the backtracking process, we will keep the partial optimal solution using lateral effects (i.e. in the predicate base), and update it whenever a better solution is found:

Code

```
:- dynamic sol_part/2.

% optimal_path(Source, Target, Path)
optimal_path(X,Y,Path):-
   asserta(sol_part([], 100)),     % 100 = max initial distance
   path(X, Y, [X], Path, 1).
optimal_path(_,_,Path):-
   retract(sol_part(Path,_)).

% path(Source, Target, PartialPath, FinalPath, PathLength)
% when target is equal to source, we save the current solution
path(Y,Y,Path,Path,LPath):-
   % we retract the last solution
   retract(sol_part(_,_)),!,
   % save current solution
   asserta(sol_part(Path,LPath)),
   % search for another solution
   fail.
path(X,Y,PPath,FPath,LPath):-
   edge(X,Z),
   not(member(Z,PPath)),
   % compute partial distance
   LPath1 is LPath+1,
   % extract distance from previous solution
   sol_part(_,Lopt),
   % if current distance is smaller than the previous distance,
   LPath1<Lopt,
   % we keep going
   path(Z,Y,[Z|PPath],FPath,LPath1).
```

The predicate *path/4* generates, via backtracking, all paths which are better than the current partial solution and updates the current partial solution whenever a shorter path is found. Once a better solution than the current optimal solution is found, the predicate replaces the old optimal in the predicate base (clause 1) and then continues the search, by launching the backtracking mechanism (using *fail*).

Follow the execution of:

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

Remember. When working with *assert/retract*, make sure you "clean up after yourselves", i.e. check that no unwanted clauses remain asserted on your predicate base after the execution of your queries (their effects are not affected by backtracking!).

1.2.4 Hamiltonian Cycle

A Hamiltonian cycle is a closed path in a graph which passes exactly once through all nodes (except for the first node, which is the source and the target of the path). Of course, not all graphs possess such a cycle. The predicate *hamilton/3* is provided below:

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

The predicate *hamilton_path/5* is left for you to implement. The predicate should search for a closed path from X, of length NN1 (number of nodes in the graph, minus 1).

Follow the execution of several queries of the *hamilton/3* predicate, using the example graph.

1.3 A better implementation of paths

Paths are saved in reverse, a first possible improvement is saving them in the correct order. As we add to the beginning of the third argument which works in a forward approach, the results are reversed. However, we can add directly to the output, while keeping the third argument only as a list of visited nodes.

Code
```
path1(X, Y, Path):- path1(X, Y, [X], Path).

path1(Y, Y, _, []).
path1(X, Y, PPath, [Z|FPath]):-
   edge(X, Z),
   not(member(Z, PPath)),
   path1(Z, Y, [Z|PPath], FPath).
```

We have reached a better implementation that does not require the path to be reversed.

However, it is not yet the best implementation we can make. Why? The *edge/2* predicate represents a single graph, thus, we can only run this path on a single graph, if we want to make it run on another graph, we have to change the implementation. One solution is the univ predicate discussed last laboratory session.

Code

```
path1(Graph, X, Y, Path):- path1(Graph, X, Y, [X], Path).

path1(_, Y, Y, _, []).
path1(G, X, Y, PPath, [Z|FPath]):-
    Edge=..[G, X, Z],                % Edge becomes → G(X, Z),
    call(Edge),
    not(member(Z, PPath)),
    path1(G, Z, Y, [Z|PPath], FPath).
```

Follow the execution of:

Query

```
?- trace, path1(edge, a, c, P).
```

2 Exercises

Each exercise has its own graph allocated to it. As such, in your implementation you should use the edges of that graph. For example, when solving Exercise 1, the predicate should call *edge_ex1/2*.

1. Rewrite the *optimal_path/3* predicate such that it operates on weighted graphs: attach a weight to each edge on the graph and compute the minimum cost path from a source node to a destination node.

The *edge* predicate will have 3 parameters.

Code

```
edge_ex1(a,c,7).
edge_ex1(a,b,10).
edge_ex1(c,d,3).
edge_ex1(b,e,1).
edge_ex1(d,e,2).
```

Query

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

2. Continue the implementation of the Hamiltonian Cycle using the *hamilton/3* predicate.

Code

```
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).
```

Query

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

3. Write the *euler/3* predicate that can find Eulerian paths in a given graf starting from a given source node.

Code

```
% euler(NE, S, R). – where S is the source node
% and NE the number of edges in the graph
edge_ex3(a,b).
edge_ex3(b,e).
edge_ex3(c,a).
edge_ex3(d,c).
edge_ex3(e,d).
```

Query

```
?- euler(5, a, R).
R = [[c, a], [d, c], [e, d], [b, e], [a, b]]
```

4. Write a predicate *cycle(X,R)* to find a closed path (cycle) starting at a given node *X* in the graph G (using the *edge/2* representation) and saves the result in R. The predicate should return all cycles via backtracking. Moreover, the predicate should receive the *edge* predicate defining the graph as an input (*suggestion*: univ predicate).

Code

```
edge_ex4(a,b).
edge_ex4(a,c).
edge_ex4(c,e).
edge_ex4(e,a).
edge_ex4(b,d).
edge_ex4(d,a).
```

Query

```
?- cycle(edge_ex4, a, R).
R = [a,d,b,a] ;
R = [a,e,c,a] ;
false
```

5. Write the *cycle(X,R)* predicate from the previous exercise using the *neighbour* representation. Moreover, the predicate should receive the *neighb* predicate defining the graph as an input (*suggestion*: univ predicate).

Code

```
neighb_ex5(a, [b,c]).
neighb_ex5(b, [d]).
neighb_ex5(c, [e]).
neighb_ex5(d, [a]).
neighb_ex5(e, [a]).
```

Query

```
?- cycle_neighb(neighb_ex5, a, R).
R = [a,d,b,a] ;
R = [a,e,c,a] ;
false
```

6. Write the predicate(s) which perform the conversion between the edge-clause representation (A1B2) to the neighbor list-list representation (A2B1).

Code

```
edge_ex6(a,b).
edge_ex6(a,c).
edge_ex6(b,d).
```

Query

```
?- retractall(gen_neighb(_,_)), edge_to_neighb, listing(gen_neighb/2).
    :- dynamic gen_neighb_list/2.

   gen_neighb(c, []).
   gen_neighb(d, []).
   gen_neighb(a, [c, b]).
   gen_neighb(b, [d]).
true
```

7. The *restricted_path/4* predicate computes a path between the source and the destination node, and then checks whether the path found contains the nodes in the restriction list. Since predicate path used forward recursion, the order of the nodes must be inversed in both lists – path and restrictions list. Try to motivate why this strategy is not efficient (use trace to see what happens). Write a more efficient predicate which searches for the restricted path between a source and a target node.

Code

```
edge_ex7(a,b).
edge_ex7(b,c).
edge_ex7(a,c).
edge_ex7(c,d).
edge_ex7(b,d).
edge_ex7(d,e).
edge_ex7(e,a).
```

Query

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

8. Write a set of Prolog predicates to solve the Wolf-Goat-Cabbage problem: "A farmer and his goat, wolf, and cabbage are on the North bank of a river. They need to cross to the South bank. They have a boat, with a capacity of two; the farmer is the only one that can row. If the goat and the cabbage are left alone without the farmer, the goat will eat the cabbage. Similarly, if the wolf and the goat are together without the farmer, the goat will be eaten."

Suggestions:

- you may choose to encode the state space as instances of the configuration of the 4 objects (Farmer, Wolf, Goat, Cabbage), represented either as a list (i.e. [F,W,G,C]), or as a complex structure (e.g. F-W-G-C, or state(F,W,G,C)).
- the initial state would be [n,n,n,n] (all north) and the final state [s,s,s,s] (all south); for the list representation of states (e.g. if Farmer takes Wolf across -> [s,s,n,n], this state should not be valid as the goat eats the cabbage).
- in each transition (or move), the Farmer can change its state (from n to s, or vice-versa) together with at most one other participant (Wolf, Goat, or Cabbage).
- this can be viewed as a path search problem in a graph.