9 Graphs. Paths in Graphs

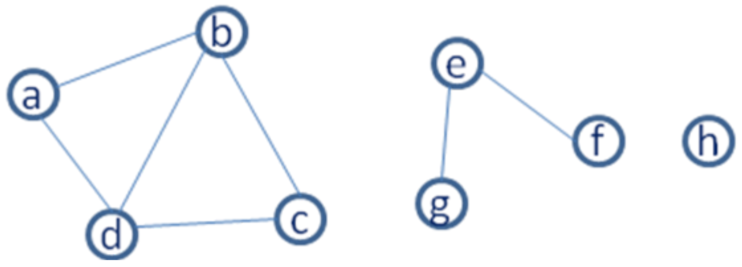
This session covers graph representation alternatives and several types of graph paths in Prolog.

9.1 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):

$$G=(V,E)$$

Let us consider an example of an 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 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
  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 edges**, stored as **predicate facts** (*edge-clause* form):

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

In this representation, 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):

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

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

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

- (A2B1) As a **set of vertices and associated neighbor lists**, stored as a **data object** (*neighbor list-list* form):
  ```
  ?- 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, [])].
  ```

- (A1B1) As **the set of vertices and the set of edges,** stored as a **data object** (*graph-term* form):
  ```
  ?- Graph = graph([a,b,c,d,e,f,g,h], [e(a,b), e(b,a), ... ]).
  ```

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* form to the *edge-clause* form:

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

neighb_to_edge:-neighbor(Node,List),
                process(Node,List),
                fail.
neighb_to_edge.

process(Node, [H|T]):- assertz(edge(Node, H)),
                       process(Node, T).
process(_, []).
```

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 in each clause separately; process traverses the neighbor list of the current node, and asserts a new fact for predicate edge for each new neighbor of the current node.

9.2 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.

9.2.1 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 is presented below:

```
% path(Source, Target, Path)

path(X,Y,Path):-path(X,Y,[X],Path).

path(X,Y,PPath, FPath):- is_edge(X,Z),
                 \+(member(Z, PPath)),
                 path(Z, Y, [Z|PPath], FPath).
path(X,X,PPath, PPath).
```

What type of recursion is used here?

Exercise 9.1: Represent a graph using the edge-clause form and trace the execution of the path predicate on different queries (you may use the example graph, perhaps add some edges to it). What happens when you repeat the question?

9.2.2 Restricted path

Let us now try to write a predicate which searches for a restricted path between two nodes in a graph, i.e. the path must pass through certain nodes, in a certain order (these nodes are specified in a list).

```
% restricted_path(Source, Target, RestrictionsList, Path)
% check_restrictions(RestrictionsList, Path)

restricted_path(X,Y,LR,P):- path(X,Y,P),
                        check_restrictions(LR, P).

check_restrictions([],_):- !.
check_restrictions([H|T], [H|R]):- !, check_restrictions(T,R).
check_restrictions(T, [H|L]):-check_restrictions(T,L).
```

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, performs the actual check. check_restrictions/2 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?

Exercise 9.2: Trace the execution of the following queries:

1. ?- check_restrictions([2,3], [1,2,3,4]).
2. ?- check_restrictions([1,3], [1,2,3,4]).
3. ?- check_restrictions([1,3], [1,2]).

Exercise 9.3: Trace the execution of several queries for the restricted_path/4 predicate, on your example graph. Which is the order in which you have to specify the nodes in the list of restrictions? Why?

9.2.3 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:

```
%optimal_path(Source, Target, Path)

:- dynamic sol_part/2.

optimal_path(X,Y,_):-asserta(sol_part([],100)),
                        path(X,Y,[X],1).
optimal_path(_,_,Path):-retract(sol_part(Path,_)).

path(X,X,Path,LPath):-retract(sol_part(_,_)),!,
                        asserta(sol_part(Path,LPath)),
                        fail.
path(X,Y,PPath,LPath):-is_edge(X,Z),
                        \+(member(Z,PPath)),
                        LPath1 is LPath+1,
                        sol_part(_,Lopt),
                        LPath1<Lopt,
                        path(Z,Y,[Z|PPath],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).

Exercise 9.3: Trace the execution of several queries for the optimal_path/3 predicate, using the example graph.

!!! When working with assert/retract, make sure you "clean 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!).

9.2.4 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:

```
%hamilton(NbNodes, 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).

Exercise 9.3: Trace the execution of several queries for the hamilton/3 predicate, using the example graph.

9.3 Quiz exercises

q9-1. Write the predicate(s) which perform the conversion between the *edge-clause* representation (A1B2) to the *neighbor list-list* representation (A2B1).

q9-2. The restricted_path 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.

q9-3. 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.

9.4 Problems

p9-1. Write a predicate cycle(A,P) to find a closed path (cycle) P starting at a given node A in the graph G (use any graph representation for G). The predicate should return all cycles via backtracking.

p9-2. (**) 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.”
Hints:
- 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], the final state [s,s,s,s], for the list representation of states (e.g. if Farmer takes Wolf across -> [s,s,n,n] (and the goat eats the cabbage), so this state should not be valid)

- 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 Enjoy!