Database Handling in Prolog

Database handling in Prolog

Two types of databases :- static and dynamic.

- ***Static database*** is a part of the program that is complied along with it. It does not change during execution of the program.
- ***Dynamic database*** can change dynamically at execution time and are of two types.

Type1: created at each execution. It grows, shrinks and is deleted at the end of program.

- This type of database is no longer available after program finishes its execution and is called **working memory**.

Cont…

Type2: Other type of dynamic databases are those which are stored in files and called database files.

- These are consulted in any program whenever required.
- These types of databases are not part of any particular program and are available for use in future by different programs using system defined predicates called **save** and **consult**.
- While executing a Prolog program one can load database file(s) using 'consult' predicate.
- These files can be updated dynamically at run time and saved using 'save' predicate.

Cont…

- The format of predicates 'save' and 'consult' are as follows:
  - **save(filename) -** succeeds after saving the contents of a file named 'filename'.
  - **consult(filename) -** succeeds after loading or adding all the clauses from a file stored in 'filename' in the current program being executed.
  - **reconsult(filename) -** succeeds after loading all the clauses by superseding all the existing clauses for the same predicate from a file 'filename'.
- Grouping of rules and facts into partitions can be easily.
- Example: Load a file named 'more' if predicates P and Q succeed,

```
R :- P, Q, consult('more').
```

Cont…

- Clauses can be added to a database at run time using following predicates.

asserta(X) & assertz(X) - succeed by adding fact X in the beginning & at the end of database of facts respectively.

  - For example, **asserta(father(mike, john))** adds fact **father(mike, john)** in the beginning of current database.

- Clauses can be constructed dynamically and asserted in a dynamic database as follows:

```
start :-  writeln('Input name of mother: '),  readln(M),
writeln('Input name of child: '), readln(C),
assert(parent(M, C)), assert(female(M)).
```

Cont…

- Similarly obsolete clauses can be deleted by using system defined predicate called ***retract*** from dynamic database at the run time.
- For example, **retract(father(mike, X))** deletes the first fact **father(mike, _)** from working memory.
- retractall(X) deletes all the clauses from a database whose head match with X.
- Example, **retractall(father(X, Y))** deletes all the facts **father(_ , _)** and **retractall(_)** deletes all the clauses from the working memory.

Graph Search

- Graph is a collection of nodes and edges. A predicate *edge(x, y)* is used to represent an edge from x to y.
- **Traversal Schemes**
  - *Depth first search:* It starts from start (initial) node deep down through one branch in depth first fashion until it reaches a final (goal) node.
  - *Breadth first search:* It basically allows traversal level wise. Search starts from the initial node and all the nodes reachable from current node are processed before the nodes at higher level.

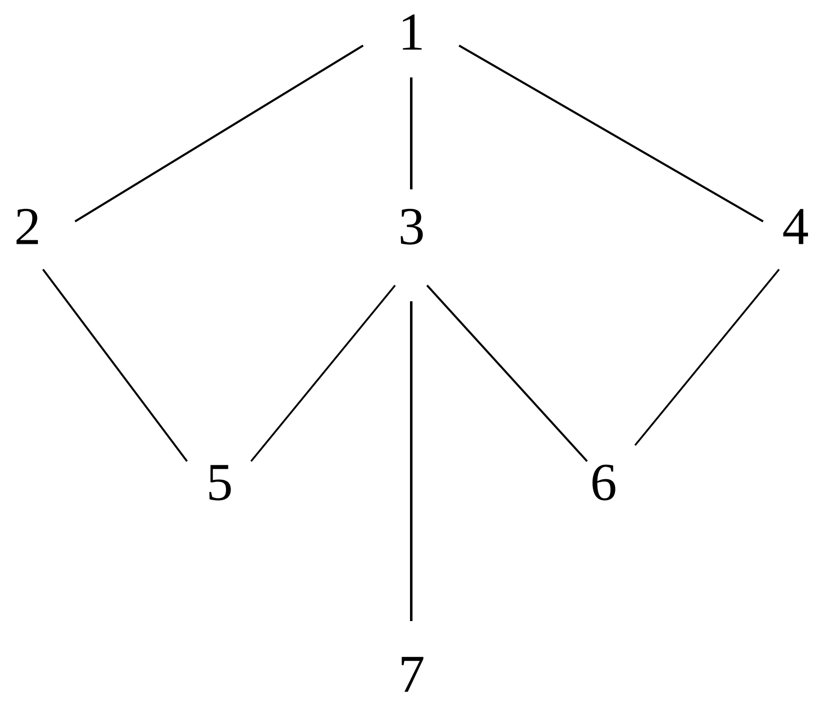
Graph Traversal - DFS

- dfs(S, G, L) - succeeds by initiating another predicate named dfs1 and binding L with the list of nodes visited from start node S to goal node G.
- dfs1(S, G, L1, L) - succeeds by unifying L with the reverse of L1, where L1 is a list of nodes visited so far in depth first fashion from S to G in reverse order.

```
dfs(S,G, L)   :-       dfs1(S,G, [S], L),!.
dfs1(S,G, L1, L):-     S = G, !, reverse(L1, L).          (1)
dfs1(X, G, L1, L):-    edge(X, Y), not(mem(Y, L1)),
                       dfs1(Y,G, [Y|L1], L).              (2)
reverse([ ],[ ]).
reverse([X|T], Y):-    reverse(T,Y1), append(Y1, [X], Y).
```

Example

- Consider the following undirected graph with initial node as 1 and goal node as 7



Cont…

- Above undirected graph is represented in Prolog as follows:

  edge(1, 2). edge(1, 3). edge(1, 4). edge(2, 1). edge(2, 5). edge(3, 1). edge(3, 5). edge(3, 7). edge(3, 6). edge(4, 1). edge(4, 6). edge(5, 2). edge(5, 3). edge(6, 3). edge(6, 4). edge(7, 3).

- Query: **?- dfs(1, 7, L).**
- Path obtained using depth first search from start node 1 to goal node 7 is: L = [1, 2, 5, 3, 7]

```
                ?- dfs1(1, 7, [1], L).
(1)                                         (2)
fails           ?- edge(1, Y), not(mem(Y, [1])), dfs1(Y,7, [Y|1], L).
                          {Y = 2}
                ?- dfs1(2,7, [2, 1], L).
(1)                       (2)
fails           ?- edge(2, Y), not(mem(Y, [2, 1])), dfs1(Y,7, [Y| 2, 1], L).
{Y =  1}                  (Y = 5}
fails           ?- dfs1( 5,7, [5,2,1], L)
(1)                       (2)
fails           ?- edge(5, Y), not(mem(Y, [5,2,1])), dfs1(Y,7,[Y| 5,2,1], L).
{Y=2}                     {Y = 3}
fails           ?- dfs1( 3, 7,[3,5,2,1], L)
(1)                         (2)
fails           ?- edge(3,Y),not(mem(Y,[3, 5,2,1])),dfs1(Y,7,[Y|3,5,2,1], L).
                          {Y = 7}
                ?- dfs1(7,7, [7| 3, 5, 2, 1], L).
                (1)
                ?- 7=7, !, reverse([7,3,5,2,1], L).
                          {L = [1,2,5,3,7]}
```

Graph Traversal - BFS

- Breadth first basically requires a data structure called queue (FIFO) and traverses graph level wise.
- Queue is implemented in Prolog using database concept.
- At any point in time, the first element is obtained from the queue database and its successors are found.
- Add them at the end of queue database by *assertz(queue(Succ, [Succ|L]))* if it is not already there.
- Check if this element is a goal node and if so, then reverse the list generated so far and delete all the elements from the queue database.

```
%   bfs(X, L)       -    X is a start node and L is the list of nodes to be
                         visited from start to goal nodes in order to find path.
%   f_succ(X, L1, Y)-    succeeds by finding Y as a successor of X and adding
                         queue(Y, [Y|L1]) at the end of queue database. On
                         backtracking it finds all the successors of X and fails if no
                         successor is available.
    bfs(S, L)       :-   start_node(S), asserta(queue(S, [S])), queue(X, L1),
                         f_succ(X, L1, Y), goal_node(Y), !, queue(Y, L2),
                         reverse(L2, L), retractall(queue( _, _ )).
    f_succ(X, L1, Y) :-  edge(X, Y), not(mem_list(Y, L1)),
                         assertz(queue(Y, [Y|L1])).
    f_succ(X, L1, Y) :-  retract(queue(X, L1)), fail.
```

Unification Program in Prolog

- Simple unification program in Prolog

% unification(T1, T2)- succeeds if terms T1 and T2 unifies

```
unification(T1, T2) :-      var(T1), var(T2), T1 = T2, !.
unification(T1, T2) :-      var(T1), nonvar(T2), T1 = T2, !.
unification(T1, T2) :-      nonvar(T1), var(T2), T2 = T1, !.
unification(T1, T2):-       nonvar(T1),  nonvar(T2),atomic(T1),
                            atomic(T2), T1 = T2, !.
unification(T1,T2):-        nonvar(T1), nonvar(T2),
                            compound(T1), compound(T2),
                            unify_term(T1, T2), !.
unify_term(T1, T2) :-       functor(T1, F, N), functor(T2, F, N),
                            unify_args(T1, T2, N).
unify_args(T1, T2, N) :-    N > 0, unify(T1, T2, N), N1 is N - 1,
                            unify_args(T1, T2, N1).
unify_args(T, T, 0).
unify(T1, T2, N)  :-        arg(N, T1, A1), arg(N, T2, A2),
                            unification(A1, A2).
```

Examples of Queries

Goals:

?- unification(3, X).

X = 3

?- unification(f(3, X), f(Y, 4)).

Y = 3, X = 4

?-unification(f(X), f(g(6))).

X = g(6);

?- unification(3, 4)

fails

Resolving two clauses and displaying the resolvant

- Program gets both the clauses in the form of list, for example, a clause (a V b V ~c V d) as [a, b, not(c), d] from the user and display its resolvant.
- The resolvant is obtained by taking union after deleting positive and negative literals from the parents.
- Example

  resolvant (a V ~ b V c, ~ a V c V d V e) = (~ b V c V d V e)

Resolvant Program

```
start   :- writeln('Input first clause in the form of list'), read(C1),
           writeln('Input second clause'), read(C2), res(C1, C2, R),
           writeln('Resolvant of two clauses is :'),writeln(R).
res(C1, C2, R)  :- matching(C1,C2,R).
matching(C1,C2,R) :-     literal_matching(C1, C2, C1literal, C2literal),
                         delete(C1,C1literal,C1new),
                         delete(C2,C2literal, C2new),
                         matching(C1new,C2new,R).
matching(C1,C2,R):-      union(C1, C2, R).
literal_matching(C1, C2, C1literal, C2literal) :-
    mem(C1literal, C1), mem(C2literal, C2), C2literal = not(C1literal).
literal_matching(C1, C2, C1literal, C2literal) :-
    mem(C1literal,C1), mem(C2literal, C2), C1literal = not(C2literal).
union([X | Xs], Y, U)  :-   mem(X, Y), union(Xs, Y, U).
union([X | Xs], Y, [X |U]) :- not(mem(X, Y),union(Xs, Y, U).
union([], Y, Y).
```