Programs

(available in ftp://ftp.essex.ac.uk/pub/csp/csp_programs.asc)

```
/*=========================================================
Program 5.1         :       bt.plg
Subject             :       Backtracking algorithm applied to the N-queens
                            problem
=========================================================*/

/*
        queens(N, Result)
        N is the number of queens to be placed.
        Result is a list of integers representing the solution.
        The i-th number in the list represents the column of queen on the i-th row.
        e.g. calling by ?- queens(8, Result)
        should get something like: Result = [5,7,2,6,3,1,4,8]
*/
queens(N, Result) :-
        range(N, Range),
        queens(Range, [], Result).

/*
        create a list of numbers from 1 to N
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

/*
        queens(Unlabelled, Labelled, Solution)
        Unlabelled is the list of rows with unlabelled queens;
        Labelled accumulates the labelled queens;
        Solution is the solution.
*/
queens([], Solution, Solution).
queens(UnlabelledQs, LabelledQs, Solution) :-
        delete(Q, UnlabelledQs, Rest),
        noattack(Q, LabelledQs, 1),
        queens(Rest, [Q|LabelledQs], Solution).

delete(A, [A|L], L).
delete(A, [B|L], [B|L1]) :- delete(A,L,L1).

noattack(_,[],_).
noattack(Y, [Y1| YL], XD) :-
        Y1-Y =\= XD,
        Y-Y1 =\= XD,
        D1 is XD + 1,
        noattack(Y, YL, D1).

/*=========================================================*/
```

```
/*=========================================================
Program 5.2     :     ib.plg
Subject         :     Iterative Broadening algorithm applied to the N-
                      queens problem
Note            :     This program needs Program 5.3: random.plg
=========================================================*/

queens( N, Result ) :-
        gen_domain( N, Domain ),
        gen_bound( N, Bound ),
        write('Bound = '), write(Bound), nl,
        ib_bt( N, Domain, Bound, [], Result ).

gen_domain( 0, [] ).
gen_domain( N, [N|L] ) :-
        N > 0, N1 is N - 1, gen_domain( N1, L ).

gen_bound( Max, Bound ) :-
        1 =< Max, gen_bound( Max, 1, Bound ).

gen_bound( Max, Bound, Bound ).
gen_bound( Max, N, Bound ) :-
        N < Max, N1 is N + 1, gen_bound( Max, N1, Bound ).

ib_bt( 0, _, _, Result, Result ).
ib_bt( X, Domain, Bound, Labelled, Result ) :-
        X1 is X - 1,
        random_N_times( Bound, Domain, V ),     /* defined in random.plg */
        noattack( X/V, Labelled ),
        delete( V, Domain, Rest ),
        ib_bt( X1, Rest, Bound, [X/V| Labelled], Result ).

delete( X, [X|Rest], Rest ).
delete( X, [H|L1], [H|L2] ) :- X \== H, delete( X, L1, L2 ).

noattack( _, [] ).
noattack( X0/V0, [X1/V1|Rest] ):-
        V0 =\= V1,
        V1-V0 =\= X1-X0,
        V1-V0 =\= X0-X1,
        noattack( X0/V0, Rest ).

/*=========================================================*/
```

```
/*=========================================================
Program 5.3     :     random.plg
Subject         :     Predicates for generating pseudo random numbers
Notes           :     random(L,U,R) takes three parameters : L and U
                      are the range of numbers you want; R is for the
                      result.
                      The random numbers it produces are integers in the
                      range L to R inclusive, so if you called random(1,
                      100, K) it would come out with K being bound to a
                      random number between 1 and 100.
=========================================================*/

random(L, U, R) :-
        retract(seed(Xi)),
        Xi1 is (371 * Xi) mod 3191,
        assert(seed(Xi1)),
        R is (Xi1 mod (U - L + 1)) + L, !.
random(L,U,R) :-
        X is ((U * (3137 * L) + 1) mod (U - L + 1)) + L,
        assert(seed(X)), random(L, U, R), !.

/*-------------------------------------------------------------------------*/

/*
        random_element( List, Element )
        Randomly pick an element from the given List
*/
random_element( [], _ ) :- !, fail.
random_element( [X], X ) :- !.
random_element( L, E ) :-
        P =.. [dummy |L],
        functor( P, _, Max ),
        random( 1, Max, Rand ),
        arg( Rand, P, E ), !.

/*
        random_N_times( N, List, X ) returns X as an element of List. It will suc-
        ceed a maximum of N times.
*/
random_N_times( N, List, X ) :-
        random_N_times( N, List, X, 1 ).

random_N_times( N, List, Result, I ) :-
        random_element( List, Y ),
        random_N_times_aux( N, List, Y, Result, I ).

random_N_times_aux( _, _, X, X, _ ).
random_N_times_aux( N, List, LastResult, Result, I ) :-
```

```
        I < N, I1 is I + 1,
        'random: delete'( LastResult, List, Rest ),
        random_N_times( N, Rest, Result, I1 ).

/*
        random_ordering( List, Result )
        Randomly order the elements in the List, giving Result
*/
random_ordering( [], [] ).
random_ordering( List, [E|Result] ) :-
        random_element( List, E ),
        'random: delete'( E, List, Rest ),
        random_ordering( Rest, Result ).

'random: delete'( E, [H|Rest], Rest ) :- E == H.
'random: delete'( E, [H|List], [H|Rest] ) :-
        E \== H, 'random: delete'( E, List, Rest ).

/*=======================================================*/
```

```
/*=========================================================
Program 5.4     :     fc.plg
Subject         :     Forward Checking algorithm applied to the N
                      queens problem
=========================================================*/

queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        look_ahead_search(C, R),
        report(R).

/*
        range(N, List)
        Given a number N, range creates the List:
                        [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1],
        return as the 3rd argument the Candidates:
                [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).


/*
        look_ahead_search(Candidates, Solution)
        The main clause for searching:
        The algorithm is: pick one value for one queen, propagate the constraints
        that it creates to other queens, then handle the next queen, untill all the
        queens are labelled.
*/
look_ahead_search([], []).
look_ahead_search([X/L| T], [X/V| R]) :-
        member(V, L),
        propagate(X/V, T, Temp),
        look_ahead_search(Temp, R).
```

```
/*      to propagate the constraints of a label to others:
        The label, input as the 1st argument, is propagated to one queen at a time,
        until all the queens are considered.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [],
        propagate(X/V, T, T1).

/*
        Given a choice (i.e. X/V), prop/3 restricts the domain of the Y-th queen
        (C) to an updated domain (R).
*/
prop(X/V, Y/C, R) :-
        del(V, C, C1),
        V1 is V-(X-Y),
        del(V1, C1, C2),
        V2 is V + (X-Y),
        del(V2, C2, R).

/*
        del(X, List, Result) deletes X from List and instantiates Result to the
        result. It succeeds whether X exists in List or not.
*/
del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

report([]) :- nl, nl.
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

/*=========================================================*/
```

```
/*=========================================================
Program 5.5     :     dac.lookahead.plg
Subject         :     Directional Arc-consistency Lookahead algorithm
                      applied to the N-queens problem
Notes           :     To be used with:
                            Program 5.6: ac.plg
                            Program 5.7: print.queens.plg
=========================================================*/

queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        sort_labels(C, accum([]), SortedC),
        dac_look_ahead_search(SortedC, R),
        print_queens(R).

range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).

/*
        dac_look_ahead_search(Candidates, Solution)
        This is the main predicate for the search. The algorithm is: pick one value
        for one queen, propagate the constraints by maintaining DAC, then han-
        dle the next queen, untill all the queens are labelled.
*/
dac_look_ahead_search([], []).
dac_look_ahead_search([X/L| T], [X/V| R]) :-
        member(V, L),
        propagate(X/V, T, Temp1),
        maintain_directed_arc_consistency(Temp1, DAC_Problem),
        sort_labels(DAC_Problem, accum([]), Sorted_DAC_Problem),
        dac_look_ahead_search(Sorted_DAC_Problem, R).

/*
        to propagate the constraints of a choice to others:
        The choice, input as the 1st argument, is propagated to one queen at a
        time, until all the queens are considered.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [],
        propagate(X/V, T, T1).
```

```
prop(X/V, Y/C, R) :-
        del(V, C, C1),
        V1 is V - (X - Y),
        del(V1, C1, C2),
        V2 is V + (X - Y),
        del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

/*=========================================================*/
```

```
/*=======================================================
Program 5.6     :     ac.plg
Subject         :     Predicates for maintaining AC and DAC in the N-
                      queens problem
Notes           :     Two external calls:
                      (1) maintain_arc_consistency(Unlabelled, Result)
                      (2) maintain_directed_arc_consistency(Unlabelled,
                      Result)
                      Data structure:
                      Given: is a problem, represented by a list of vari-
                      ables together with their domains, in the form "Var/
                      Values", for example:
                              [1/[1,4,8], 2/[2,4], 3/[4,7,8]]
                      Returned: a problem possibly with some values
                      removed from certain domain such that arc-consist-
                      ency/directional arc-consistency is maintained.
=======================================================*/

/*
        (1) maintain_arc_consistency(Unlabelled, NewUnlabelled)
        Given Unlabelled, which is a list of Variable/Domain, return NewUnla-
        belled where arc consistency is achieved.
*/
maintain_arc_consistency(Unlabelled, NewUnlabelled) :-
        maintain_ac(to_be_checked(Unlabelled), checked([]), NewUnlabelled).

maintain_ac(to_be_checked([]), checked(NewUnlabelled), NewUnlabelled).
maintain_ac(to_be_checked([X/Dx| U]), checked(Checked), NewUnlabelled) :-
        bagof( V, (ac_member(V,Dx), ac(X/V, U), ac(X/V, Checked)), NewDx ),
        /* if no such V exists, fail to achieve AC in the problem */
        maintain_ac_aux(X/Dx/NewDx, U, Checked, NewUnlabelled).


maintain_ac_aux(X/Dx/Dx, U, Checked, NewUnlabelled) :-
        maintain_ac(to_be_checked(U), checked([X/Dx|Checked]), NewUnla-
        belled).
maintain_ac_aux(X/Dx/NewDx, U, Checked, NewUnlabelled) :-
        Dx \== NewDx,
        ac_append(Checked, [X/NewDx], Temp),
        ac_append(U, Temp, ToBeChecked),
        maintain_ac(to_be_checked(ToBeChecked), checked([]), NewUnla-
        belled).

/*
        ac(X/Vx, Var_Dom)
        X/Vx is a variable X with a value Vx; Var_Dom is a list of Variable/
        Domain;
        ac/2 succeeds iff for each element in Var_Dom, there exists a label which
```

```
            is compatible with X/Vx.
*/
ac(_, []).
ac(X/Vx, [Y/Dy| L]) :-
            ac(X, Vx, Y, Dy),
            ac(X/Vx, L).

/*
            ac(X, Vx, Y, Dy)
            Dy is the legal domain of Y at present
            ac/4 succeeds iff there exists a value Vy in Dy such that <X,Vx> and
            <Y,Vy> are compatible.
*/
ac(X, Vx, Y, [Vy|_]) :-
            Vx \== Vy,
            X-Y =\= Vx-Vy,
            X-Y =\= Vy-Vx, !.
ac(X, Vx, Y, [_|Dy]) :-
            ac(X, Vx, Y, Dy).

/ac_member(X,[X|_]).
ac_member(X,[_|L]) :- 'ac_member'(X,L).

ac_append([], L, L) .
ac_append([H|L1], L2, [H|L3]) :- ac_append(L1, L2, L3) .

/*-------------------------------------------------------------------------*/

            (2) maintain_directed_arc_consistency(Unlabelled, NewUnlabelled)
            Given Unlabelled, which is a list of Variable/Domain, return NewUnla-
            belled where directed arc consistency is achieved.
*/
maintain_directed_arc_consistency([], []).
maintain_directed_arc_consistency([X/Dx|Unlabelled], [X/NewDx|NewUnla-
            belled]) :-
            maintain_directed_arc_consistency(Unlabelled, NewUnlabelled),
            bagof( V, ('ac_member'(V,Dx), ac(X/V, NewUnlabelled)), NewDx ).

/*=========================================================*/
```

```
/*=========================================================
Program 5.7          :        print.queens.plg
Subject              :        Predicates for printing result for the N-queens
                              problem
                              Given a list of labels in the form: [Var1/Val1, Var2/
                              Val2, ....], print the positions of the queens
=========================================================*/

print_queens(R) :-
        sort_labels(R, accum([]), SortedR),
        nl, write('** Solution:'), nl,
        report(SortedR), nl.

sort_labels([], accum(L), L).
sort_labels([X/Vx|L1], accum(L2), R) :-
        insert(X/Vx, L2, Temp),
        sort_labels(L1, accum(Temp), R).

insert(X/Vx, [], [X/Vx]).
insert(X/Vx, [Y/Vy|L], [X/Vx,Y/Vy|L]) :- X < Y.
insert(X/Vx, [Y/Vy|L], [Y/Vy|R]) :- X >= Y, insert(X/Vx, L, R).

report([]).
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

/*=========================================================*/
```

```
/*=========================================================
Program 5.8     :     ac.lookahead.plg
Subject         :     AC-Lookahead Algorithm applied to the N-queens
                      problem
Notes           :     To be used with:
                            Program 5.6: ac.plg
                            Program 5.7: print.queens.plg
=========================================================*/

queens(N, R) :-
        range(N, L), setup_candidate_lists(N, L, C),
        sort_labels(C, accum([]), SortedC),
        ac_look_ahead_search(SortedC, R), print_queens(R).

range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1, setup_candidate_lists(N1, L, R).

/*      This is the main clause for searching. The algorithm is: pick one value for
        one queen, propagate the constraints by maintaining AC, then handle the
        next queen, till all the queens are labelled.
*/
ac_look_ahead_search([], []).
ac_look_ahead_search([X/L| T], [X/V| R]) :-
        member(V, L), propagate(X/V, T, Temp1),
        maintain_arc_consistency(Temp1, AC_Problem),
        sort_labels(AC_Problem, accum([]), Sorted_AC_Problem),
        ac_look_ahead_search(Sorted_AC_Problem, R).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [], propagate(X/V, T, T1).

prop(X/V, Y/C, R) :-
        del(V, C, C1), V1 is V-(X-Y),
        del(V1, C1, C2), V2 is V + (X-Y), del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

/*=========================================================*/
```

```
/*=========================================================
Program 5.9     :     bj.plg
Subject         :     BackJumping (BJ) algorithm applied to the N-
                      queens problem.
=========================================================*/

queens(N, R) :-
        range(N, L),
        reverse(L, List_of_variables),
        setup_candidate_lists(N, List_of_variables, C),
        reverse(C, Variables_and_domains),
        bj_search(Variables_and_domains, R, [], -1),
        is_list(Result), report(R).

range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).

/*
        bj_search(Candidates, Solution, Committed, BT_Des)
        Candidates is a list:
                [X/Domain_of_X| Other_Variables_&_Domains];
        Solution is a variable for returning the output; Committed is a list of
        labels:
                [X/Value_for_X| Other_Labels]
        BT_Des is the variable to be backtracked to when needed
*/
bj_search([], R, R, _).
bj_search([X/[]| _], bt_to(BT_Des), _, BT_Des) :-
        writeln([‘domain for Q’,X,’ exhausted, bt to ‘,BT_Des,’ <<‘]).
bj_search([X/[V|L]| T], Result, Accum, BT_Destination) :-
        no_conflict( X/V, Accum ),
        writeln([‘>> looking at <‘,X,’,’,V,’>’]),
        bj_search(T, Temp, [X/V|Accum], -1),
/*      do not search for alternative results if no solution is found in the the above
        call. */
        (Temp = bt_to(_), !; true),
        bj( Temp, Result, [X/L| T], Accum, BT_Destination ).
bj_search([X/[V|L]| T], Result, Accum, BT_To ) :-
        \+ no_conflict( X/V, Accum ),
        find_earliest_conflict(X/V, Accum, Earliest_Conflict),
        max( Earliest_Conflict, BT_To, BT_Destination ),
        bj_search([X/L| T], Result, Accum, BT_Destination ).
/* the following clause is included for alternative results */
```

```
bj_search([X/[V|L]| T], Result, Accum, BT_To ) :-
        no_conflict( X/V, Accum ),
        bj_search([X/L| T], Result, Accum, BT_To ).


bj( bt_to(Y), bt_to(Y), [X/_|_], _, _ ) :-
        Y < X,
        writeln([‘BJ ignores all other values for variable ‘,X,’!’]).
bj( bt_to(Y), Result, [X/L| T], Accum, _ ) :-
        Y >= X,
        bj_search( [X/L| T], Result, Accum, X - 1).
bj( Result, Result, _, _, _ ) :- is_list(Result).

max( X, Y, X ) :- X >= Y.
max( X, Y, Y ) :- X < Y.

find_earliest_conflict( X/V, [_|L], EC ) :-
        find_earliest_conflict( X/V, L, EC ), !.
find_earliest_conflict( X/V, [Y/W|L], Y ) :-
        conflict( X/V, Y/W ).

no_conflict( X/V, [] ).
no_conflict( X/V, [Y/W|L] ) :-
        \+ conflict( X/V, Y/W ),
        no_conflict( X/V, L ).

conflict( _/V, _/V ) :- !.
conflict( X/V, Y/W ) :- X - Y =:= V - W, !.
conflict( X/V, Y/W ) :- X - Y =:= W - V, !.


report([]) :- nl, nl.
report([_/V | L]) :-
        Space is (V - 1) * 2, tab(Space), write(‘Q’), nl,
        report(L), !.

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).

reverse( List, Result ) :- reverse( List, Result, [] ).

reverse([], Result, Result).
reverse([H|L1], R, Accum) :- reverse( L1, R, [H|Accum] ).

is_list([]).
is_list([_|_]).

/*=======================================================*/
```

```
/*=========================================================
Program 5.10        :       lncl.plg
Subject             :       Learning Nogood Compound Labels algorithm
                            applied to the N-queens problem
Notes:              :       nogood(Compound_Label) is asserted to record
                            compound_labels which has been proved to be
                            unviable. Progress is reported to show the use of
                            nogoods
=========================================================*/

:- op( 100, yfx, [:]).

queens(N, R) :-
        range(N, L),
        reverse(L, RL),
        retract_all( nogood(_) ),
        lncl_search( domains:RL, unlabelled:RL, labelled:[], R ),
        report(R).

/*
        lncl_search( domains:D, unlabelled:U, labelled:L, R )
        D       Domain, list of all possible values
        U       list of Variables which are not yet labelled
        L       list of Variable/Value pairs already committed to
        R       Result, to be instantiated to list of Variable/Value
        lncl_search/4 behaves like chronological_backtracking, except that
        whenever backtracking is needed, culprit compound labels are identified
        and recorded as nogood. The program rejects any compound label which
        has the nogood sets in it in the future.
*/
lncl_search( _, unlabelled:[], labelled:R, R ).
lncl_search( domains:D, unlabelled:[H|U], labelled:L, R ) :-
        member_and_not_recorded_as_nogood( [H/V|L], D ),
        all_consistent( L, H/V ),
        sort( [H/V|L], L1 ),
        writeln( [‘>> Considering <‘,H,’,’,V,’> ...’] ),
        lncl_search( domains:D, unlabelled:U, labelled:L1, R ).
lncl_search( domains:D, unlabelled:[H|_], labelled:L, _ ) :-
        writeln( [‘Over-constrained: ‘,L,’, backtrack <<‘] ),
        record_nogoods( domains:D, H, L ),
        !, fail.

/*-----------------------------------------------------------------------*/

/*
        range(N, List)
        Given a number N, range creates the List: [N, N - 1, N - 2, ..., 3, 2, 1].
*/
```

```
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

reverse( L, R ) :- reverse( L, R, [] ).
reverse( [], R, R ).
reverse( [H|L], Result, Temp ) :- reverse( L, Result, [H|Temp] ).

/*
        member_and_not_recorded_as_nogood( Assignments, Domain )
        This predicate does two things at the same time. First, it takes an element
        from Domain and "assigns" it to the 1st element of the 1st argument,
        which is a list. Alternative elements (from Domain) will be used as long
        as L has not become nogood. Secondly, it checks whether the 1st argu-
        ment after assignment of the new value is recorded as nogood.
*/
member_and_not_recorded_as_nogood( [H/V|L], [V|_] ) :-
        not_recorded_as_nogood( [H/V|L] ).
member_and_not_recorded_as_nogood( [H/_|L], _ ) :-
        nogood(NG),
        sublist(NG, L),
        writeln( ['Queen-',H,' is rejected as ',NG, ' is recorded nogood.'] ),
        !, fail.
member_and_not_recorded_as_nogood( L, [_|Domain] ) :-
        member_and_not_recorded_as_nogood( L, Domain ).

/*
        not_recorded_as_nogood( L )
        L is not recorded as nogood
*/
not_recorded_as_nogood( L ) :-
        nogood( NG ),
        sublist( NG, L ),
        writeln([L,' is rejected as ',NG,' is recorded nogood...']),
        !, fail.
not_recorded_as_nogood( _ ) .

/*
        sublist( L1, L2 )
        L1 is a sublist of L2
*/
sublist( [], _ ) .
sublist( [H|L1], L2 ) :- member( H, L2 ), sublist( L1, L2 ).

/*
        all_consistent( L, H/V )
        L       List of Variable/Value
        H/V     a label <H,V>
        sublist/2 succeeds if H/V is consistent with all elements of L
```

```
*/
all_consistent( [], _ ) .
all_consistent( [X/Vx|L], Y/Vy ) :-
        \+ conflict( X/Vx, Y/Vy ),
        all_consistent( L, Y/Vy ).

/*-------------------------------------------------------------------------*/
/*
        record_nogoods( domain:D, X, L )
        D       list of values
        X       variable which has to take a value from D
        L       list of Variable/Value
        For each value V in the domain D, find all elements in L which are incon-
        sistent with X/V
*/
record_nogoods( domains:D, X, L ) :-
        identify_conflicts( D, X, L, Conflicts ),
        find_covering_set( Conflicts, NG, accumulator:[] ),
        sort( NG, SortedNG ),
        update_nogood_sets( nogood(SortedNG) ),
        fail.
record_nogoods( _, _, _ ) .

/*
        identify_conflicts( D, X, L, Conflicts ),
        D                       domain
        X                       Variable
        L                       list of [X1/V1, X2/V2, ...]
        Conflicts   list of list of labels to be returned, element-i is a list of labels
        which from L which have conflict with <X,i>. e.g.:
                                [[X1/V1,X2/V2], [X3/V2], ...]
        NB: the use of “bagof”, not “findall” in the 2nd clause is  important here.
        bagof will fail if X/Vx has no conflict label, whereas findall will instanti-
        ate C1 to [] under such situations.
*/
identify_conflicts( [], _, _, [] ) .
identify_conflicts( [Vx|Rest], X, L, [C1|Conflicts] ) :-
        bagof( Label, (member(Label,L), conflict(X/Vx, Label)), C1 ),
        identify_conflicts( Rest, X, L, Conflicts ) .

conflict( _/V, _/V ) :- !.
conflict( X/Vx, Y/Vy ) :- X-Y =:= Vx-Vy, !.
conflict( X/Vx, Y/Vy ) :- X-Y =:= Vy-Vx, !.

/*
        This is a very naive way to find covering sets from the given list. Identical
        sets could be re-discovered repeatedly. The efficiency of this predicate
        could be greatly improved. Finding covering sets is itself a Constraint
```

```
            Satisfaction
            Problem.
*/
find_covering_set( [], L, accumulator:L ).
find_covering_set( [C1|Cs], NG, accumulator:A ) :-
            member(X,C1),
            set_union( X, A, A1 ),
            find_covering_set( Cs, NG, accumulator:A1 ).

set_union( X, A, A ) :- member( X, A ) .
set_union( X, A, [X|A] ) :- \+ member( X, A ).

update_nogood_sets( nogood(L) ) :-
            nogood(NG), sublist(NG, L), !.
update_nogood_sets( nogood(L) ) :-
            nogood(NG), NG\==L, sublist(L, NG), retract(NG), fail.
update_nogood_sets( P ) :-
            asserta(P),
            writeln( [‘..... record ‘,P] ) .

/*-------------------------------------------------------------------------*/
*/
report([]) :- nl, nl.
report([_/V | L]) :- tab( (V - 1) * 2 ), write(‘Q’), nl, report(L).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

retract_all( P ) :- retract( P ), fail.
retract_all( P ) .

/*=========================================================*/
```

```
/*=========================================================
Program 6.1         :       mwo.plg
Subject             :       To find Minimal Width Ordering for input graphs
Note                :       A graph is assumed to be recorded in the database
                            in the following form:
                                    node(Node)
                                    edge(Node1, Node2)
=========================================================*/
:- op(100, yfx, in).

minimal_width_ordering( MWO ) :-
        bagof( Node, node(Node), Nodes ),
        mwo( Nodes, MWO, [] ).
minimal_width_ordering( [] ) :-         /* graph without nodes */
        \+node(_).

mwo( [], L, L ).
mwo( [N|L], Result, Accum ) :-
        setof( N1, adjacent(N, N1 in L), List ),
        length( List, Len ),
        least_connections( L, bsf(N,Len), [N|L], Node1, Rest ),
        mwo( Rest, Result, [Node1|Accum] ).
mwo( [Node1|L], Result, Accum ) :-
        \+ adjacent(Node1, _ in L),             /* Node1 is unadjacent */
        mwo( L, Result, [Node1|Accum] ).

/*      least_connections( Nodes, bsf(N1, Degree), NodesInG, Result, Rest )
        to find the node from [N1| Nodes] which is adjacent to the least number
        nodes in NodesInG, and return such node as Result. The rest of the nodes
        are returned as Rest. read bsf as “best so far”
*/
least_connections( [], bsf(Result,_), _, Result, [] ).
least_connections( [N1|L], bsf(N0,Len0), Nodes, Result, [N|Rest] ) :-
        setof( N2, adjacent(N1, N2 in Nodes), List ), length( List, Len1 ),
        (Len1 =< Len0, N = N0,
        least_connections( L, bsf(N1,Len1), Nodes, Result, Rest);
        Len1 > Len0, N = N1,
        least_connections( L, bsf(N0,Len0), Nodes, Result, Rest)).
least_connections( [N1|L], bsf(N0,_), Nodes, N1, [N0|L] ) :-
        \+ adjacent( N1, _ in Nodes ).          /* N1 is unadjacent */

adjacent( X, Y in List ) :- edge( X, Y ), in( Y, List ).
adjacent( X, Y in List ) :- edge( Y, X ), in( Y, List ).

in( X, [X|_] ).
in( X, [Y|L] ) :- X \= Y, in( X, L ).

/*=========================================================*/
```

```
/*=======================================================
Program 6.2     :       mbwo1.plg
Subject         :       To find Minimal Bandwidth Orderings for input
                        graphs, using the algorithm in (Gurari & Sudbor-
                        ough, 1984)
Notes           :       A graph is assumed to be recorded in the database
                        in the following form:
                                node(Node)
                                edge(Node1, Node2)
                        tried_already/2 is asserted into the database
=======================================================*/
/*
        minimal_bandwidth_ordering( MBWO, K )
        Given a graph represented in the above form, return one minimal band-
        width ordering (MBWO) at a time, together with the bandwidth. The
        search is complete, in the sense that it can find all the orderings with min-
        imal bandwidth.
*/
minimal_bandwidth_ordering( MBWO, K ) :-
        bagof( Node, node(Node), Nodes ),
        length( Nodes, Len ),
        Max_bandwidth is Len - 1,
        gen_num( Max_bandwidth, K ),      /* 1 =< K =< Max_bandwidth */
        retract_all( tried_already(_, _) ),
        bw( [([],[],[])], MBWO, K ).

gen_num( Len, K ) :- Len >= 1, gen_num( Len, 1, K ).

gen_num( Len, K, K ).
gen_num( Len, M, K ) :- M < Len, M1 is M + 1, gen_num( Len, M1, K ).

bw( [(C, [V1|R], D) | Q], Result, K ) :-
        length( [V1|R], K ),
        delete_edge( (V1,V2), D, D1 ),
        update( (C, [V1|R], D1 ), V2, (NewC, NewR, NewD) ),
        bw_aux( (NewC, NewR, NewD), Q, Result, K ).
bw( [(C, R, D) | Q], Result, K ) :-
        \+ length( R, K ),
        findall( V, unassigned( C, R, V ), U ),
        update_all( U, (C,R,D), Q, Result, K ).

bw_aux( (C, R, []), _, Result, _ ) :-
        append( C, R, Result ).
bw_aux( (C, R, D), Q, Result, K ) :-
        D \== [],
        plausible_n_untried( R, D, K ),
        append( Q, [(C,R,D)], Q1 ),
        bw( Q1, Result, K ).
```

```
bw_aux( (C, R, D), Q, Result, K ) :-
        D \== [],
        \+ plausible_n_untried( R, D, K ),
        bw( Q, Result, K ).

update_all( [], _, Q, Result, K ) :- bw( Q, Result, K ).
update_all( [V|L], (C,R,D), Q, Result, K ) :-
        update( (C,R,D), V, (C1,R1,D1) ),
        update_all_aux( L, (C,R,D), (C1,R1,D1), Q, Result, K ).

update_all_aux( _, _, (C,R,[]), _, Result, _ ) :-
        append( C, R, Result ).
update_all_aux( L, (C,R,D), (C1,R1,D1), Q, Result, K ) :-
        plausible_n_untried( R1, D1, K ),
        append( Q, [(C1,R1,D1)], Q1 ),
        update_all( L, (C,R,D), Q1, Result, K ).
update_all_aux( L, (C,R,D), (_,R1,D1), Q, Result, K ) :-
        \+ plausible_n_untried( R1, D1, K ),
        update_all( L, (C,R,D), Q, Result, K ).

update( (C,R,D), S, (C1,R1,D1) ) :-
        delete_all_edges( (S,_), D, Temp ),
        move_conquered_nodes( (C,R,Temp), (C1,TempR), [] ),
        append( TempR, [S], R1 ),
        findall( (S,X), adjacent_nodes( S, X, R ), List ),
        append( Temp, List, D1 ).

move_conquered_nodes( (C,[],_), (C1,[]), Accum ) :-
        append( C, Accum, C1 ).
move_conquered_nodes( (C,[H|R],D), (C1,R1), Accum ) :-
        \+ edge_member( (H,_), D ),
        move_conquered_nodes( (C,R,D), (C1,R1), [H|Accum] ).
move_conquered_nodes( (C,[H|R],D), (C1,[H|R]), Accum ) :-
        edge_member( (H,_), D ),
        append( C, Accum, C1 ).

adjacent_nodes( S, X, R ) :-
        (edge( S, X ); edge( X, S )),
        \+ member( X, R ).

plausible_n_untried( R, D, K ) :-
        plausible( R, D, K ),
        \+ tried( R, D ),
        sort( R, Sorted_R ),
        sort( D, Sorted_D ),
        assert( tried_already(Sorted_R, Sorted_D) ).
```

```
plausible( R, D, K ) :-
        length( R, LenR ),
        Limit is K - LenR + 1 ,
        limited_dangling_edges( R, D, Limit ).

limited_dangling_edges( [], _, _ ).
limited_dangling_edges( [X|L], D, Limit ) :-
        findall( Y, (member((X,Y),D); member((Y,X),D)), List ),
        length( List, Len ),
        Len =< Limit,
        limited_dangling_edges( L, D, Limit + 1 ).

tried( R, D ) :-
        sort( R, Sorted_R ),
        sort( D, Sorted_D ),
        tried_already( Sorted_R, Sorted_D ).

unassigned( C, R, V ) :- node(V), \+ member(V, C), \+ member(V, R).

delete_edge( _, [], [] ).
delete_edge( (X,Y), [(X,Y)|L], L ).
delete_edge( (X,Y), [(Y,X)|L], L ).
delete_edge( (X,Y), [(X1,Y1)|L1], [(X1,Y1)|L2] ) :-
        (X,Y) \= (X1,Y1), (X,Y) \= (Y1,X1),
        delete_edge( (X,Y), L1, L2 ).

delete_all_edges( _, [], [] ).
delete_all_edges( (X,Y), [(X,Y)|L], Result ) :-
        delete_all_edges( (X,Y), L, Result ).
delete_all_edges( (X,Y), [(Y,X)|L], Result ) :-
        delete_all_edges( (X,Y), L, Result ).
delete_all_edges( (X,Y), [(X1,Y1)|L1], [(X1,Y1)|L2] ) :-
        (X,Y) \= (X1,Y1), (X,Y) \= (Y1,X1),
        delete_all_edges( (X,Y), L1, L2 ).

edge_member( (X,Y), [(X,Y)|_] ).
edge_member( (X,Y), [(Y,X)|_] ).
edge_member( Edge, [_|L] ) :- edge_member( Edge, L ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

retract_all( P ) :- retract(P), fail.
retract_all( _ ).
/*=========================================================*/
```

```
/*=======================================================
Program 6.3     :       mbwo2.plg
Subject         :       Program to find Minimal Bandwidth Orderings
                        (compared with mbwo1.plg, this is an implementa-
                        tion of an algorithm which is more natural for Pro-
                        log)
Note            :       A graph is assumed to be recorded in the database
                        in the following form:
                                node(Node)
                                edge(Node1, Node2)
=======================================================*/
/*
        minimal_bandwidth_ordering( MBWO, K )
        Given a graph represented in the above form, return one minimal band-
        width ordering (MBWO) at a time, together with the bandwidth. The
        search is complete, in the sense that it can find all the orderings with min-
        imal bandwidth.
        "setof" is used to collect all the orderings with the minimal bandwidth,
        and the cut after it is used to disallow backtracking to generate greater
        Max_bandwidth.
*/
minimal_bandwidth_ordering( MBWO, K ) :-
        bagof( Node, node(Node), Nodes ),
        length( Nodes, Len ),
        Max_bandwidth is Len - 1,
        gen_num( Max_bandwidth, K ),    /* 1 =< K =< Max_bandwidth */
        setof( Ordering, mbwo(K,[],[],Nodes,Ordering), Solutions ), !,
        member( Solutions, MBWO, _ ).
minimal_bandwidth_ordering( [], _ ) :-          /* graph without nodes */
        \+node(_).


gen_num( Len, K ) :- Len >= 1, gen_num( Len, 1, K ).

gen_num( Len, K, K ).
gen_num( Len, M, K ) :- M < Len, M1 is M + 1, gen_num( Len, M1, K ).

/*
        mbwo( K, Passed, Active, Unlabelled, Result )
        Active has at most K elements.
        Invariance: (1) the bandwidth of Passed + Active =< K; (2) none of the
        nodes in Passed are adjacent to any of the nodes in Unlabelled;
*/
mbwo( _, Passed, Active, [], Result ) :-
        append( Passed, Active, Result ).
mbwo( K, Passed, Active, Unplaced, Result ) :-
        length( Active, LenActive ),
        LenActive < K,
```

```
        member( Unplaced, Node, Rest ),
        append( Active, [Node], NewActive ),
        mbwo( K, Passed, NewActive, Rest, Result ).
mbwo( K, Passed, [H|Active], Unplaced, Result ) :-
        length( Active, LenActive ), LenActive + 1 =:= K,
        member( Unplaced, Node, Rest ),
        no_connection( H, Rest ),
        append( Active, [Node], NewActive ),
        append( Passed, [H], NewPassed ),
        mbwo( K, NewPassed, NewActive, Rest, Result ).

member( [X|L], X, L ).
member( [H|L], X, [H|R] ) :- member( L, X, R ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

no_connection( _, [] ).
no_connection( X, [Y|List] ) :-
        \+adjacent( X, Y ), no_connection( X, List ).

adjacent( X, Y ) :- edge( X, Y ).
adjacent( X, Y ) :- edge( Y, X ).

/*=========================================================*/
```

```
/*=========================================================
Program 6.4     :     ffp-fc.plg
Subject         :     Forward Checking algorithm applied to the N-
                      queens problem; Fail-first Principle is used in
                      selecting the next variable
=========================================================*/

/*
        queens(N, R)
        N is the number of queens, and R is a solution
        The main clause. Called by, say, ?- (queens(8, Result).
*/
queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        forward_checking_with_ffp(C, R),
        report(R).


/*
        range(N, List)
        Given a number N, range creates the List:
                        [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1],
        return as the 3rd argument the Candidates:
                [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).


/*
        forward_checking_with_ffp( Unlabelled, Solution)
        This is the main clause for searching. Unlabelled is a list of X/Dx, where
        X is a variable and Dx is its domain. The algorithm is: pick one value for
        one queen, propagate the constraints that it creates to other queens, then
        handle the next queen, till all the queens are labelled. If the picked vari-
        ble cannot be labelled, the call will fail. For the picked variable, all values
```

```
        will be tried.
*/
forward_checking_with_ffp([], []).
forward_checking_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
        length(Dh, Len_H),
        select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
        select_value(X/Domain, V, Rest, Updated_Unlabelled),
        forward_checking_with_ffp(Updated_Unlabelled, R).


/*
        select_variable(Unlabelled, H/Dh, Len_H, X/Domain, Rest),
        Given a set of unlabelled variables and their domains (1st Arg), return the
        variable X which has the smallest Domain and the remaining unlabelled
        variable/domains (5th arg). H is the variable which has the smallest
        domain found so far, where Dh is the domain of H, and Len_H is the size
        of Dh.
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
        length(Dy, Ly),
        Ly < Lx,
        select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
        length(Dy, Ly),
        Ly >= Lx,
        select_variable(L, X/Dx, Lx, Result, Rest).


/*
        select_value( X/Dom, V, Unlabelled, Updated_Unlabelled)
        Given variable X and its domain (Dom) and a set of unlabelled variables
        and their domains (Unlabelled), return a value (V) in Dom and the
        updated domains for the unlabelled variables in Unlabelled. It fails if all
        the values will cause the situation to be over-constrained. In this imple-
        mentation, no heuristics is being used.
*/
select_value(X/[V|_], V, U, Updated_U) :-
        propagate(X/V, U, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :-
        select_value(X/L, V, U, Updated_U).

/*      propagate( Assignment, Unlabelled, Updated_Unlabelled )
        It propagates the effect of the Assignment to the Unlabelled variables.
        The Assignment is propagated to one queen at a time, until all the queens
        are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
```

```
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [],
        propagate(X/V, T, T1).

/*
        prop( X/Vx, Y/Dy, Updated_Dy )
        Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
        (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
        del(V, C, C1),
        V1 is V-(X-Y),
        del(V1, C1, C2),
        V2 is V + (X-Y),
        del(V2, C2, R).

/*
        del( Element, List, Result )
        delete an Element  from the input List, returning the Result as the 3rd
        argument. del/3 succeeds whether Element exists in List or not.
*/
del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

report([]) :- nl, nl.
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

/*=========================================================*/
```

```
/*=======================================================
Program 6.5     :     ffp-dac.plg
Subject         :     DAC-Lookahead algorithm applied to the N-queens
                      problem; Fail-first Principle is used in selecting the
                      next variable; DAC is maintained among unlabelled
                      variables.
Note            :     The following programs are required:
                            Program 5.6: ac.plg
                            Program 5.7: print.queens.plg
=======================================================*/

queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        dac_lookahead_with_ffp(C, R),
        print_queens(R).            /* defined in print.queens.plg */

range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).

/*
        dac_lookahead_with_ffp( Unlabelled, Solution )
        This is the main clause for searching. Unlabelled is a list of X/Dx, where
        X is a variable and Dx is its domain. The algorithm is: pick one value for
        one queen, propagate the constraints by maintaining directional arc-con-
        sistency, then handle the next queen, till all the queens have been labelled.
        If the picked variable cannot be labelled, the call will fail. For the picked
        variable, all values will be tried.
*/
dac_lookahead_with_ffp([], []).
dac_lookahead_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
        length(Dh, Len_H),
        select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
        select_value(X/Domain, V, Rest, Updated_Unlabelled),
        dac_lookahead_with_ffp(Updated_Unlabelled, R).

/*
        Given a set of unlabelled variables and their domains (1st Arg), return the
        variable (X) which has the smallest domain (Dom) and the remaining
        unlabelled variable/domains (5th arg).
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
```

```
        length(Dy, Ly), Ly < Lx,
        select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
        length(Dy, Ly), Ly >= Lx,
        select_variable(L, X/Dx, Lx, Result, Rest).

/*
        select_value(X/Dom, V, U, Updated_U)
        Given variable X and its domain (Dom) and a set of unlabelled variables
        and their domains (U), return a value (V) in Dom and the updated
        domains for the unlabelled variables in U. Fail if all the values cause the
        situation over-constrained. In this implementation, no heuristics is used.
        maintain_directed_arc_consistency/2 is defined in the program ac.plg.
*/
select_value(X/[V|_], V, U, Updated_U) :-
        propagate(X/V, U, Temp),
        maintain_directed_arc_consistency(Temp, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :-
        select_value(X/L, V, U, Updated_U).

/*      propagate( Assignment, Unlabelled, Updated_Unlabelled )
        It propagates the effect of the Assignment to the Unlabelled variables.
        The Assignment is propagated to one queen at a time, until all the queens
        are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [],
        propagate(X/V, T, T1).

/*
        prop( X/Vx, Y/Dy, Updated_Dy )
        Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
        (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
        del(V, C, C1),
        V1 is V-(X-Y),
        del(V1, C1, C2),
        V2 is V + (X-Y),
        del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

/*=========================================================*/
```

```
/*=======================================================
Program 6.6     :     ffp-ac.plg
Subject         :     AC-Lookahead algorithm applied to the N-queens
                      problem: Fail-first Principle is used in selecting the
                      next variable; Arc-consistency is maintained among
                      unlabelled variables.
Note            :     The following programs are required:
                            Program 5.6: ac.plg
                            Program 5.7: print.queens.plg
=======================================================*/
/*
        The main clause, the 1st argument is used to distinguish it from other def-
        initions of “queens” when more than one file is loaded.
*/
queens(N, R) :-
        range(N, L), setup_candidate_lists(N, L, C),
        label_with_ffp(C, R), print_queens(R).

/*      range(N, List)
        Given a number N, range/2 creates the List:
                        [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

/*      setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1], return as the
        3rd argument the Candidates: [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1, setup_candidate_lists(N1, L, R).

/*      label_with_ffp( Unlabelled, Solution )
        This is the main clause for searching. Unlabelled is a list of X/Dx, where
        X is a variable and Dx is its domain. The algorithm: pick one value for
        one queen, propagate the constraints by maintaining Arc-Consistency,
        then handle the next queen, till all the queens are labelled. If the picked
        variable cannot be labelled, the call will fail. For the picked variable, all
        values will be tried.
*/
label_with_ffp([], []).
label_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
        length(Dh, Len_H),
        select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
        select_value(X/Domain, V, Rest, Updated_Unlabelled),
        label_with_ffp(Updated_Unlabelled, R).
```

```
/*      Given a set of unlabelled variables and their domains (1st Arg), return the
        variable (X) which has the smallest domain (Dom) and the remaining
        unlabelled variable/domains (5th arg).
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
        length(Dy, Ly), Ly < Lx,
        select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
        length(Dy, Ly), Ly >= Lx,
        select_variable(L, X/Dx, Lx, Result, Rest).

/*      select_value( X/Dom, V, U, Updated_U)
        Given variable X and its domain (Dom) and a set of unlabelled variables
        and their domains (U), return a value (V) in Dom and the updated
        domains for the unlabelled variables in U. Fail if all the values cause the
        situation over-constrained.
*/
select_value(X/[V|_], V, U, Updated_U) :-
        propagate(X/V, U, Temp),
        maintain_arc_consistency(Temp, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :- select_value(X/L, V, U, Updated_U).

/*      propagate( Assignment, Unlabelled, Updated_Unlabelled )
        It propagates the effect of the Assignment to the Unlabelled variables.
        The Assignment is propagated to one queen at a time, until all the queens
        are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
        prop(X/V, Y/C, C1), C1 \== [], propagate(X/V, T, T1).

/*      prop( X/Vx, Y/Dy, Updated_Dy )
        Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
        (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
        del(V, C, C1), V1 is V - (X - Y),
        del(V1, C1, C2), V2 is V + (X - Y), del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

/*=======================================================*/
```

```
/*=========================================================
Program 6.7         :       inf_bt.plg
Subject             :       Solving the N-queens problem with backtracking,
                            using the Min-conflict Heuristic to order the values
Note                :       This program requires the following programs:
                                    Program 5.3:    random.plg
                                    Program 5.7:    print.queens.plg
=========================================================*/


:- op( 100, yfx, less ).                    /* for difference list */

/*
        Initial_Assignment should be a difference list
*/
queens(N, Result) :-
        generate_domain( N, Domain ),
        initialize_labels( N, Domain, Initial_labels ),
        informed_backtrack( Domain, Initial_labels, X less X, Temp, 0 ),
        retrieve_from_diff_list( Temp, Result ),
        print_queens( Result ).

/*-------------------------------------------------------------------------------------------*/

generate_domain( N, [] ) :- N =< 0.
generate_domain( N, [N|L] ) :-
        N > 0, N1 is N - 1, generate_domain(N1, L).

/*-------------------------------------------------------------------------------------------*/
/*
        initialize_labels( N, Domain, Assignments )
        It generates Assignments, which is a difference list representing a near-
        solution. initialize_labels/3 uses the min_conflicts heuristic.
*/
initialize_labels( N, Domain, Assignments ) :-
        init_labels( N, Domain, Assignments, X less X ).

init_labels( 0, _, Result, Result ).
init_labels( N, Domain, Result, L1 less L2 ) :-
        pick_one_value( N, Domain, L1 less L2, V, Remaining_Values ),
        N1 is N - 1,
        init_labels( N1, Remaining_Values, Result, [N/V|L1] less L2).

pick_one_value( _, [V], _, V, [] ).
pick_one_value( N, [V1|Vs], Labels less Tail, V, Rest ) :-
        Vs \== [],
        count_conflicts( N/V1, Labels less Tail, Bound ),
        find_min_conflict_value(Bound-N/[V1], Vs, Labels less Tail, V ),
```

```
        delete( V, [V1|Vs], Rest ).

/*
        find_min_conflict_value( Bound-N/V1, Vs, Labelled, V )
        given a label N/V1 and the number of conflicts that it has with the assign-
        ments in Labelled, pick from Vs a value V such that X/V has fewer con-
        flicts with Labelled. If no such V exists, instantiate V to V1. If the Bound
        is 0, then there is no chance of improvement. In this case, a random value
        is picked (this is done in the 1st clause).
*/
find_min_conflict_value( _-_/Vs, [], _, V ) :-
        random_element( Vs, V ).                    /* defined in random.plg */
find_min_conflict_value( Bound-X/V, [V1|Vs], Labelled, Result ) :-
        count_conflicts( X/V1, Labelled, Count, Bound ),
        fmcv( Bound-X/V, Count-X/V1, Vs, Labelled, Result ).

fmcv( Count-X/L, Count-X/V1, Vs, Labelled, R ) :-
        find_min_conflict_value( Count-X/[V1|L], Vs, Labelled, R).
fmcv( Bound-X/L, Count-_, Vs, Labelled, Result ) :-
        Bound < Count,
        find_min_conflict_value( Bound-X/L, Vs, Labelled, Result ).
fmcv( Bound-_, Count-X/V1, Vs, Labelled, R ) :-
        Bound > Count,
        find_min_conflict_value( Count-X/[V1], Vs, Labelled, R ).

/*-----------------------------------------------------------------------------------------*/
/*
        informed_backtrack(Domain, VarsLeft, VarsDone, Result, Count)
        Domain is the domain that each variable can take — in the N-queens
        problem, all variables have the same domain;
        VarsLeft is a difference list, which represents the labels which have not
        yet been fully examined;
        VarsDone is a difference list, which represents the labels which have been
        checked and guaranteed to have no conflict with each other;
        Result is a difference list, which will be instantiated to VarLeft + VarDone
        when no conflict is detected.
        Count counts the number of iterations needed to find the solution — used
        solely for reporting.
*/
informed_backtrack( Domain, VarsLeft, VarsDone, Result, Count ) :-
        pick_conflict_label( VarsLeft, VarsDone, X ),
        delete_from_diff_list( X/Old, VarsLeft, Rest ), !,
        order_values_by_conflicts( X, Domain, Rest, VarsDone, Ordered_Do-
        main ),
        member( _-V, Ordered_Domain ),
        add_to_diff_list( X/V, VarsDone, New_VarsDone ),
        delete( V, Domain, D1 ),
        Count1 is Count + 1,
```

```
        informed_backtrack( D1, Rest, New_VarsDone, Result, Count1 ).
informed_backtrack( _, X less Y, Y less Z, X less Z, Count ) :-
        write('Iterations needed: '), write(Count).

/*
        pick_conflict_label( VarsLeft, Labels_to_check, X )
*/
pick_conflict_label( L1 less L2, _, _ ) :-
        L1 == L2, !, fail.
pick_conflict_label( [X/V| Rest] less L1, L2 less L3, R ) :-
        no_conflicts( X/V, Rest less L1 ),
        no_conflicts( X/V, L2 less L3 ), !,
        pick_conflict_label( Rest less L1, [X/V| L2] less L3, R ).
pick_conflict_label( [X/_|_] less _, _, X ).


/*
        order_values_by_conflicts( X, D, VarsLeft, VarsDone, Result )
*/
order_values_by_conflicts( X, Domain, VarsLeft, VarsDone, Result ) :-
        bagof( Count-V, (member(V,Domain),
                        no_conflicts( X/V, VarsDone ),
                        count_conflicts( X/V, VarsLeft, Count )),
                Temp
         ),
        modified_qsort( Temp, Result ).


no_conflicts( _, L1 less L2 ) :- L1 == L2.
no_conflicts( X1/V1, [X2/V2| L1] less L2 ) :-
        [X2/V2| L1] \== L2,
        noattack( X1/V1, X2/V2 ),
        no_conflicts( X1/V1, L1 less L2 ).


modified_qsort( [], [] ).
modified_qsort( [Pivot-X|L], Result ) :-
        split( Pivot, L, Equal, Less, More ),
        modified_qsort( Less, Sorted_Less ),
        modified_qsort( More, Sorted_More ),
        random_ordering( [Pivot-X|Equal], Temp1 ),/* random.plg */
        append( Sorted_Less, Temp1, Temp2 ),
        append( Temp2, Sorted_More, Result ).


split( _, [], [], [], [] ).
split( V, [V-X|L1], [V-X|L2], L3, L4 ) :-
        split( V, L1, L2, L3, L4 ).
```

```
split( Pivot, [V-X|L1], L2, [V-X|L3], L4 ) :-
        V < Pivot, split( Pivot, L1, L2, L3, L4 ).
split( Pivot, [V-X|L1], L2, L3, [V-X|L4] ) :-
        V > Pivot, split( Pivot, L1, L2, L3, L4 ).

/*-------------------------------------------------------------------------------------------*/
/*
        count_conflicts ( X/V, Labelled, Count )
        count the number of conflicts between X/V and the Labelled variables,
        returning Count.
*/
count_conflicts( _, L1 less L2, 0 ) :- L1 == L2, !.
count_conflicts( X/V, [Y/W| L1] less L2, Count ) :-
        noattack( X/V, Y/W ), !,
        count_conflicts( X/V, L1 less L2, Count ).
count_conflicts( X/V, [_| L1] less L2, Count ) :-
        count_conflicts( X/V, L1 less L2, Count0 ),
        Count is Count0 + 1.


/*
        count_conflicts ( X/V, Labelled, Count, Max_Count )
        count the number of conflicts between X/V and the Labelled variables,
        returning Count. If Count > Max_Count, there is no need to continue. Just
        return 0.
*/
count_conflicts( _, L1 less L2, 0, _ ) :- L1 == L2, !.
count_conflicts( _, _, 0, N ) :- N < 0, !.
count_conflicts( X/V, [Y/W| L1] less L2, Count, Max ) :-
        noattack( X/V, Y/W ), !,
        count_conflicts( X/V, L1 less L2, Count, Max ).
count_conflicts( X/V, [_| L1] less L2, Count, Max ) :-
        Max1 is Max - 1,
        count_conflicts( X/V, L1 less L2, Count0, Max1 ),
        Count is Count0 + 1.


noattack(X0/V0, X1/V1):-
        V0 =\= V1,
        V1-V0 =\= X1-X0,
        V1-V0 =\= X0-X1.

/*-------------------------------------------------------------------------------------------*/

/*
        add_to_diff_list( X, Difference_List1, Result )
        to add X to a difference list, giving Result.
*/
```

```
add_to_diff_list( X, L1 less L2, [X|L1] less L2 ).

delete_from_diff_list( _, L1 less L2, L1 less L2 ) :-
        L1 == L2, !.
delete_from_diff_list( X, [X|L1] less L2, L1 less L2 ).
delete_from_diff_list( X, [H|L1] less L2, [H|L3] less L2 ) :-
        X \= H,
        delete_from_diff_list( X, L1 less L2, L3 less L2 ).

retrieve_from_diff_list( L1 less L2, [] ) :- L1 == L2.
retrieve_from_diff_list( [H|L1] less L2, [H|L3] ) :-
        [H|L1] \== L2,
        retrieve_from_diff_list( L1 less L2, L3 ).

reverse_diff_list( Diff_list, Reverse ) :-
        reverse_diff_list( Diff_list, Reverse, L less L ).

reverse_diff_list( L1 less L2, Result, Result ) :- L1 == L2.
reverse_diff_list( [H|L1] less L2, Result, L3 less L4 ) :-
        [H|L1] \== L2,
        reverse_diff_list( L1 less L2, Result, [H|L3] less L4 ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

/*
        delete(X,L1,L2)
        deletes the first occurence of X from L1, giving L2.
*/
delete( _, [], [] ).
delete( X, [X|L], L ).
delete( X, [H|L1], [H|L2] ) :- X \= H, delete( X, L1, L2 ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

/*=========================================================*/
```

```
/*=========================================================
Program 7.1         :       partition.plg
Subject             :       Program to partition the nodes in the given graph
                            into unconnected clusters
Notes               :       A graph is assumed to be recorded in the database
                            in the following form:
                                        node(Node)
                                        edge(Node1, Node2)
=========================================================*/
partition(Clusters) :-
        bagof(N, node(N), Nodes),
        delete(X, Nodes, RestOfNodes), !,
        partition([X], RestOfNodes, [], Clusters).
partition([]).

partition([], [], Cluster, [Cluster]).
partition(L, [], Accum, [Cluster]) :-
        append(L, Accum, Cluster).
partition([], Nodes, Cluster1, [Cluster1| Clusters]) :-
        Nodes \== [],                   /* start another cluster */
        delete(X, Nodes, RestOfNodes),
        partition([X], RestOfNodes, [], Clusters).
partition([H|L], Nodes, Accum, Clusters) :-
        Nodes \== [],
        findall(X, (member(X, Nodes), adjacent(H,X)), List),
        delete_list(List, Nodes, RestOfNodes),
        append(L, List, NewL),
        partition(NewL, RestOfNodes, [H|Accum], Clusters).

adjacent(X,Y) :- edge(X,Y).
adjacent(X,Y) :- edge(Y,X).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- X \== H, delete(X, L1, L2).

delete_list([], L, L).
delete_list([H|L1], L2, L3) :- delete(H, L2, Temp), delete_list(L1, Temp, L3).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).
/*=========================================================*/
```

```
/*=========================================================
Program 7.2         :       acyclic.plg
Subject             :       To check whether an undirected graph is Acyclic
Notes               :       A graph is assumed to be recorded in the database
                            in the following form:
                                    node(Node)
                                    edge(Node1, Node2)
=========================================================*/
acyclic :-
        bagof(N, node(N), Nodes), bagof((A,B), edge(A,B), Edges), node(X), !,
        acyclic([X], (Nodes, Edges)).

acyclic([], ([], _)).
acyclic([], (Nodes, Edges)) :- member(X, Nodes), acyclic([X], (Nodes, Edges)).
acyclic([H|L], (Nodes, Edges)) :-
        delete(H, Nodes, RestNodes),
        findall(Y, adjacent((H,Y), Edges), Connected),
        remove_connections( H, Connected, Edges, RestEdges),
        no_cycle( Connected, L ),
        append(Connected, L, L1),
        writeln([H,' removed, graph = (',Nodes,',',Edges,')']),
        acyclic(L1, (RestNodes, RestEdges)).

adjacent((X,Y), Edges) :- member((X,Y), Edges).
adjacent((X,Y), Edges) :- member((Y,X), Edges).

remove_connections(_, [], L, L).
remove_connections(X, [Y|L], Edges, RestEdges) :-
        delete((X,Y), Edges, Temp1),
        delete((Y,X), Temp1, Temp2),
        remove_connections(X, L, Temp2, RestEdges).

no_cycle([], _).
no_cycle([H|L], Visited) :- \+ member(H, Visited), no_cycle(L, Visited).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- X \== H, delete(X, L1, L2).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).
/*=========================================================*/
```

```
/*=======================================================
Program 7.3         :       max-clique.plg
Subject             :       To find all Maximum Cliques in a given graph
Notes               :       A graph is assumed to be recorded in the database
                            in the following form:
                                    node(Node)
                                    edge(Node1, Node2)
=======================================================*/
/*          max_cliques( MC )
            It instantiates MC to the set of all maximum cliques in the graph which is
            in the Prolog database.
*/
max_cliques( MC ) :- bagof(N, node(N), Nodes), mc(Nodes, [], [], MC).

mc([], _, _, []).
mc(Nodes, _, Excluded_nodes, []) :- no_clique(Excluded_nodes, Nodes), !.
mc(Nodes, _, _, [Nodes]) :- clique(Nodes), !.
mc(Nodes, Include_nodes, Excluded_nodes, MC) :-
            delete(X, Nodes, RestOfNodes), \+member(X, Include_nodes), !,
            findall(N, (member(N,RestOfNodes), adjacent(N,X)), Neighbours),
            mc([X|Neighbours], [X|Include_nodes], Excluded_nodes, MC1),
            mc(RestOfNodes, Include_nodes, [X|Excluded_nodes], MC2),
            append(MC1, MC2, MC).

/*          no_clique(N, C)
            no clique exists if any of the nodes in N is adjacent to all the nodes in C
*/
no_clique([H|_], C) :- all_adjacent(C, H).
no_clique([_|L], C) :- no_clique(L, C).

clique([]).
clique([H|L]) :- all_adjacent(L, H), clique(L).

all_adjacent([], _).
all_adjacent([H|L], X) :- adjacent(H,X), all_adjacent(L, X).

adjacent( X, Y ) :- edge( X, Y ).
adjacent( X, Y ) :- edge( Y, X ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- delete(X, L1, L2).
/*=======================================================*/
```

```
/*=======================================================
```

Program 7.4 : **alp.plg**
Subject : *AnalyzeLongestPath* algorithm
Note : A graph is assumed to be recorded in the database in the following form:

path(From, To, Length).

The predicate abs_*time(Point, Abs_Time)*, if present, states the absolute time of the Point. Abs_Time can either be an integer or a term *min(Time)*. All times are assumed to be possitive.
** NB : This program does not detect any untidiness of the database, e.g. duplicated clauses on the same path or *abs_time* constraint

```
=======================================================*/
/*
        analyse_longest_path
        analyse_longest_path succeeds if the temporal constraints in the given
        graph is satisfiable.
*/
analyse_longest_path :- analyse_longest_path(_).

/*
        analyse_longest_path(R)
        succeeds if the temporal constraints can be satisfied, in which case R is
        instantiated to the set of nodes in the database together with their earliest
        possible time
*/
analyse_longest_path(ResultList) :-
        setof( X, node_n_time(X), L ),
        setof( (Y,TimeY), (in(Y,L), alp_gets_time(Y,TimeY)), List ),
        (alp( to_be_processed(L), List, ResultList, visited([]) ), !,
         alp_satisfy_abs_time_constraint( ResultList ),
         writeln([‘AnalyzeLongestPath succeeds, result: ‘]);
         writeln([‘AnalyzeLongestPath fails: inconsistency detected ‘]),
         !, fail
        ).

/*
        alp_satisfy_abs_time_constraint( List )
        checks to see if all (Point,Time) pairs in the List satisfies all the abs_time
        constraints in the database. Checking is performed here instead of alp/4 in
        order to improve the clarity of alp/4.
*/
alp_satisfy_abs_time_constraint( [] ).
alp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
        abs_time( X, T ), integer(T),
        (T == TimeX;
```

```
            writeln([‘Time of ‘,X,’ violates abs_time constraint ‘,T]), !, fail
            ), !,
            alp_satisfy_abs_time_constraint( L ).
alp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
            abs_time( X, max(T) ),
            (TimeX =< T;
            writeln([‘Time of ‘,X,’ exceeds max. abs_time constraint ‘,T]), !, fail
            ), !,
            alp_satisfy_abs_time_constraint( L ).
alp_satisfy_abs_time_constraint( [_| L] ) :-
            alp_satisfy_abs_time_constraint( L ).

/*-------------------------------------------------------------------------*/

            Main predicates for analysing longest path

            alp( to_be_processed(L1), L2, Result, visited(V) )
            mutually recursive with alp_updates_time/5.
*/
alp(to_be_processed([]), Result, Result, _).            /* finished */
alp(to_be_processed([A|_]), _, _, visited(Visited_Nodes)) :-
            in(A, Visited_Nodes), !,
            writeln([‘  Loop with node ‘,A]),
            !, fail.                                    /* loop detected */
alp(to_be_processed([A| L]), List, Result, visited(V)) :-
            in((A,TimeA), List),
            bagof((P,Length), path(A, P, Length), U),
            (alp_updates_time(U, TimeA, List, Updated_List, visited([A|V]));
            writeln([‘ Loop with node ‘,A]), !, fail
            ), !,
            alp(to_be_processed(L), Updated_List, Result, visited(V)).

/*
            alp_updates_time( List1, Time, List2, List3, visited(V) )
            +           List1: list of (successor, distance) for updating
            +           Time: time at predecessor
            +           List2: most updated list of (point,time)
            -           List3: List 2 with time checked and possibly updated
            +           V: visited nodes, for checking loops
*/
alp_updates_time([], _, Result, Result, _).
alp_updates_time([(Y,Distance_X_Y)|U], TimeX, List, Result, Visited) :-
            delete((Y,TimeY), List, Rest),
            AltTimeY is TimeX + Distance_X_Y,
            AltTimeY > TimeY, !,
            alp(to_be_processed([Y]), [(Y,AltTimeY) |Rest], Temp, Visited), !,
            alp_updates_time(U, TimeX, Temp, Result, Visited), !.
alp_updates_time([_|U], T, List, Result, Visited) :-
```

```
        alp_updates_time(U, T, List, Result, Visited).

/*-------------------------------------------------------------------------*/

in( X, [X|_] ).
in( X, [_|L] ) :- in( X, L ).

delete( _, [], [] ).
delete( X, [X|Rest], Rest ).
delete( X, [Y|L], [Y|Rest] ) :- X\=Y, delete( X, L, Rest ).

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([H|L]) :- write(H), writeln(L).

/*-------------------------------------------------------------------------*/

                    PREDICATES RELATED TO THE DATABASE */
/*
        pick one node and find its time
*/
node_n_time(X) :- (path(X,_,_); path(_,X,_)).

alp_gets_time(A, TimeA) :- abs_time(A, TimeA), integer(TimeA), !.
alp_gets_time(A, TimeA) :- abs_time(A, min(TimeA)), !.
alp_gets_time(_, 0) :- !.

/*=========================================================*/
```

```
/*=======================================================
Program 7.5     :     asp.plg
Subject         :     AnalyzeShortestPath algorithm
Note            :     A graph is assumed to be recorded in the database
                      in the following form:
                            path(From, To, Length).
                      The predicate abs_time(Point, Abs_Time), if
                      present, states the absolute time of the Point.
                      Abs_Time can either be an integer or a term min(T-
                      ime). All times are assumed to be positive.
                      ** NB : This program does not detect any untidi-
                      ness of the database, e.g. duplicated clauses on the
                      same path or abs_time constraint
=======================================================*/

:- op(100, yfx, [less_than]).

/*
        analyse_shortest_path
        analyse_shortest_path succeeds if the temporal constraints in the
        given graph is satisfiable.
*/
analyse_shortest_path :- analyse_shortest_path(_).

/*
        analyse_shortest_path(R)
        it succeeds if the temporal constraints can be satisfied, in which case R is
        instantiated to the set of nodes in the database together with their earliest
        possible time
*/
analyse_shortest_path(ResultList) :-
        setof( X, node_n_time(X), L ),
        setof( (Y,TimeY), (in(Y,L), asp_gets_time(Y,TimeY)), List ),
        (asp( to_be_processed(L), List, ResultList, visited([]) ), !,
         asp_satisfy_abs_time_constraint( ResultList ),
         writeln([‘AnalyzeShortestPath succeeds, result: ‘]);
         writeln([‘AnalyseShortest Path fails: inconsistency detected ‘]),
         !, fail
        ).


/*
        asp_satisfy_abs_time_constraint( List )
        checks to see if the (Point,Time) pairs in the List satisfies all the abs_time
        constraints in the database. Checking is performed here instead of alp/4 in
        order to improve the clarity of alp/4.
*/
asp_satisfy_abs_time_constraint( [] ).
```

```
asp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
        abs_time( X, T ), integer(T),
        (T == TimeX;
         writeln([‘Time of ‘,X,’ violates abs_time constraint ‘,T]), !, fail
        ), !,
        asp_satisfy_abs_time_constraint( L ).
asp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
        abs_time( X, min(T) ),
        (\+(TimeX less_than T);
         writeln([‘Time of ‘,X,’ less than min. abs_time constraint ‘,T]), !, fail
        ), !,
        asp_satisfy_abs_time_constraint( L ).
asp_satisfy_abs_time_constraint( [_| L] ) :-
        asp_satisfy_abs_time_constraint( L ).

/*-------------------------------------------------------------------------*/

        Main predicates for analysing shortest path

        asp( to_be_processed(L1), L2, Result, visited(V) )
        mutually recursive with asp_updates_time/5.
*/
asp(to_be_processed([]), Result, Result, _).            /* finished */
asp(to_be_processed([A|_]), _, _, visited(Visited_Nodes)) :-
        in(A, Visited_Nodes), !,
        writeln([‘  Loop with node ‘,A]),
        !, fail.                                        /* loop detected */
asp(to_be_processed([A| L]), List, Result, visited(V)) :-
        in((A,TimeA), List), TimeA \= infinity,
        bagof((P,Length), path(P, A, Length), U),
        (asp_updates_time(U, TimeA, List, Updated_List, visited([A|V]));
         writeln([‘ Loop with node ‘,A]), !, fail
        ), !,
        asp(to_be_processed(L), Updated_List, Result, visited(V)).
asp(to_be_processed([A| L]), List, Result, visited(V)) :-
        in((A,infinity), List),
        asp(to_be_processed(L), List, Result, visited(V)).

/*
        asp_updates_time( List1, Time, List2, List3, visited(V) )
        +       List1: list of (predecessor, distance) for updating
        +       Time: time at successor
        +       List2: most updated list of (point,time)
        -       List3: List 2 with time checked and possibly updated
        +       V: visited nodes, for checking loops
*/
asp_updates_time([], _, Result, Result, _).
asp_updates_time([(X,Distance_X_Y)|U], TimeY, List, Result, Visited) :-
```

```
            delete((X,TimeX), List, Rest),
            difference(TimeY, Distance_X_Y, AltTimeX),
            AltTimeX less_than TimeX, !,
            asp(to_be_processed([X]), [(X,AltTimeX) |Rest], Temp, Visited), !,
            asp_updates_time(U, TimeY, Temp, Result, Visited), !.
asp_updates_time([_|U], T, List, Result, Visited) :-
            asp_updates_time(U, T, List, Result, Visited).

/*-------------------------------------------------------------------------*/

in( X, [X|_] ).
in( X, [_|L] ) :- in( X, L ).

delete( _, [], [] ).
delete( X, [X|Rest], Rest ).
delete( X, [Y|L], [Y|Rest] ) :- X\=Y, delete( X, L, Rest ).

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([H|L]) :- write(H), writeln(L).

difference( infinity, _, infinity ).
difference( X, infinity, -infinity ) :- X \== infinity.
difference( X, Y, Diff ) :- integer(X), integer(Y), Diff is X - Y.

_ less_than infinity.
X less_than Y :- integer(X), integer(Y), X < Y.

/*-------------------------------------------------------------------------*/

                    PREDICATES RELATED TO THE DATABASE */

/*
            pick one node and find its time
*/
node_n_time(X) :- (path(X,_,_); path(_,X,_)).

asp_gets_time(A, TimeA) :- abs_time(A, TimeA), integer(TimeA), !.
asp_gets_time(A, TimeA) :- abs_time(A, max(TimeA)), !.
asp_gets_time(_, infinity) :- !.

/*=========================================================*/
```

```
/*=========================================================
Program 8.1       :       hc.plg
Subject           :       Solving the N-queens problem using the Heuristic
                          Repair Method in Minton et al. [1990]
Note              :       The following programs are required:
                                  Program 5.3: random.plg
                                  Program 5.7: print.queens.plg
                          Search in this program is incomplete.
=========================================================*/

queens(N, Result) :-
        generate_domain( N, Domain ),
        initialize_labels( N, Domain, Initial_labels ),
        writeln([‘Initial labels: ‘, Initial_labels]),
        hill_climb( Initial_labels, Domain, Result ),
        print_queens( Result ).


/*-------------------------------------------------------------------------*/

generate_domain( N, [] ) :- N =< 0.
generate_domain( N, [N|L] ) :-
        N > 0, N1 is N - 1, generate_domain(N1, L).
/*
        initialize_labels( N, Domain, Assignments )
        it generates Assignments, which is a list representing an approximate
        solution.
        initialize_labels/3 uses the min_conflicts heuristic.
*/
initialize_labels( N, Domain, Assignments ) :-
        init_labels( N, Domain, Assignments, [] ).

init_labels( 0, _, Result, Result ).
init_labels( N, Domain, Result, Labelled ) :-
        N > 0,
        pick_value_with_min_conflict( N, Domain, Labelled, V ),
        N1 is N - 1,
        init_labels( N1, Domain, Result, [N/V| Labelled] ).

pick_value_with_min_conflict( N, [V1|Vs], Labels, V ) :-
        length(Vs, Len),
        count_conflicts( N/V1, Labels, Count, Len ),
        find_min_conflict_value(Count-N/V1, Vs, Labels, V ).

/*
        find_min_conflict_value( Bound-N/V1, Vs, Labelled, V )
        given a label N/V1 and the number of conflicts that it has with the
        Labelled, pick from Vs a value V such that X/V has less conflicts with
```

```
		Labelled. If no such V exists, instantiate V to V1. If the Bound is 0, then
		there is no chance to improve. This is handled by the 1st clause.
*/
find_min_conflict_value( _-_/V, [], _, V ).
find_min_conflict_value( 0-_/V, _, _, V ).
find_min_conflict_value( Bound-X/V, [V1|Vs], Labelled, Result ) :-
		count_conflicts( X/V1, Labelled, Count, Bound ),
		fmcv( Bound-X/V, Count-X/V1, Vs, Labelled, Result ).

fmcv( Count-X/V1, Count-X/V2, Vs, Labelled, R ) :-
		random_element( [V1,V2], V ),
		find_min_conflict_value( Count-X/V, Vs, Labelled, R).
fmcv( Bound-X/V, Count-_, Vs, Labelled, Result ) :-
		Bound < Count,
		find_min_conflict_value( Bound-X/V, Vs, Labelled, Result ).
fmcv( Bound-_, Count-X/V, Vs, Labelled, R ) :-
		Bound > Count,
		find_min_conflict_value( Count-X/V, Vs, Labelled, R ).

/*-------------------------------------------------------------------------*/
/*
		count_conflicts ( X/V, Labelled, Count, Max_Count )
		count the number of conflicts between X/V and the Labelled variables,
		returning Count. If Count is greater than Max_Count, there is no need to
		continue: just return 0.
*/
count_conflicts( _, [], 0, _ ).
count_conflicts( _, _, 0, N ) :- N < 0.
count_conflicts( X/V, [Y/W| L1], Count, Max ) :-
		Max >= 0,
		noattack( X/V, Y/W ),
		count_conflicts( X/V, L1, Count, Max ).
count_conflicts( X/V, [Y/W| L1], Count, Max ) :-
		Max >= 0, \+noattack( X/V, Y/W ),
		Max1 is Max - 1,
		count_conflicts( X/V, L1, Count0, Max1 ),
		Count is Count0 + 1.

noattack(X0/V0, X1/V1):-
		V0 =\= V1,
		V1-V0 =\= X1-X0,
		V1-V0 =\= X0-X1.

/*-------------------------------------------------------------------------*/

hill_climb( Config, Domain, Result ) :-
		setof( Label, conflict_element( Label, Config ), Conflict_list ),
		writeln([‘Conflict set: ‘, Conflict_list]),
```

```
            random_element( Conflict_list, Y/Vy ),      /* no backtrack */
            delete(Y/Vy, Config, Labelled),
            pick_value_with_min_conflict( Y, Domain, Labelled, Value ),
            writeln([‘Repair: ‘,Y,’=’,Vy,’ becomes ‘,Y,’=’,Value]),
            !,              /* no backtracking should be allowed */
            hill_climb( [Y/Value| Labelled], Domain, Result ).
hill_climb( Config, _, Config ).


conflict_element( Label, Config ) :-
            conflict_element( Label, Config, Config ).

conflict_element( X/V, [X/V| L], Config ) :- attack( X/V, Config ).
conflict_element( Label, [_| L], Config ) :- conflict_element( Label, L, Config ).

attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V0 == V1.
attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V1-V0 =:= X1-X0.
attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V1-V0 =:= X0-X1.
attack(Label, [_|L]) :- attack(Label, L).

/*
            delete(X,L1,L2)
            deletes the first occurence of X from L1, giving L2.
*/
delete( _, [], [] ).
delete( X, [X|L], L ).
delete( X, [H|L1], [H|L2] ) :- X \= H, delete( X, L1, L2 ).

writeln( [] ) :- nl.
writeln( [H|L] ) :- write(H), writeln( L ).

/*=========================================================*/
```

```
/*=========================================================
```

Program 9.1	:	**synthesis.plg**
Subject	:	*Freuder's Solution Synthesis* algorithm applied to the N-queens problem
Dynamic Clauses	:	Clauses of the following predicate will be asserted/ retracted: *node([X1,X2,...,Xn])* content([X1-V1, X2-V2, ..., Xn-Vn]) where Xi are variables and Vi are values
Note	:	This program reports the constraint propagation process

```
=========================================================*/
/*
        queens(N)
        N is the number of queens.
        queens(N) will report all the solutions to the N-queens problem.
*/
queens(N) :-
        retract_all( node(_) ), retract_all( content(_) ),
        build_nodes(1, N), report(N).

retract_all( P ) :- retract( P ), fail.
retract_all( _ ).

build_nodes(Order, N) :- Order > N.
build_nodes(Order, N) :-
        Order =< N,
        (combination( Order, N, Combination ),
         build_one_node( N, Combination ), fail;
         Order1 is Order + 1, build_nodes( Order1, N )
        ).

/*-------------------------------------------------------------------------*/

combination( M, N, Combination ) :-
        make_list(N, Variables),
        enumerate_combination( M, Variables, Combination ).

make_list(N, List) :- make_list( N, List, [] ).
make_list(0, L, L).
make_list(N, R, L) :- N > 0, N1 is N - 1, make_list(N1, R, [N|L]).

enumerate_combination( 0, _, [] ).
enumerate_combination( M, [H|Variables], [H|Combination] ) :-
        M > 0, M1 is M - 1,
        enumerate_combination( M1, Variables, Combination ).
enumerate_combination( M, [_|Variables], Combination ) :-
        M > 0, enumerate_combination( M, Variables, Combination ).
```

```
/*-------------------------------------------------------------------------*/

build_one_node( N, Combination ) :-
        assert(node(Combination)),
        writeln([‘** building node for ‘,Combination,’ **’]),
        make_list( N, Domain ),
        (one_assignment( Domain, Combination, Assignment ),
         compatible( Assignment ),
         supported( Combination, Assignment ),
         assert(content(Assignment)),
         writeln([‘>> content ‘,Assignment,’ is asserted.’]),
         fail;
         true
        ),
        downward_propagation( Combination ).

one_assignment( _, [], [] ).
one_assignment( Domain, [X1| Xs], [X1-V1| Assignments] ) :-
        member( V1, Domain ), one_assignment( Domain, Xs, Assignments ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

compatible( [X1-V1, X2-V2] ) :-
        !, V1 \== V2, X1 - X2 =\= V1 - V2, X1 - X2 =\= V2 - V1.
compatible( _ ).

supported( Vars, Assignment ) :-
        remove_one_variable( Vars, Assignment, V, A ),
        node( V ), \+content( A ), !, fail.
supported( _, _ ).

remove_one_variable( [_|V], [_|A], V, A ).
remove_one_variable( [V1|Vs], [A1|As], [V1|V], [A1|A] ) :-
        remove_one_variable( Vs, As, V, A ).

downward_propagation( C ) :-
        build_template( C, T, C1, T1 ), node( C1 ),
        assert( terminate_propagation ),
        d_propagate_all( T, T1 ), global_propagate( C1 ), fail.
downward_propagation( _ ).

/*
        build_template( C, T, C1, T1 )
        Given a list of variables, [X1, X2, ..., Xn], build: T = [X1-V1, X2-V2, ...,
        Xn-Vn]; C1 = C1 with one variable less; and T1 = T with one label less.
        Alternatively, given a C1, build C, T and T1.
```

```
*/
build_template( [X1|L1], [X1-_|L2], L3, L4 ) :-
        build_template_aux( L1, L2, L3, L4 ).
build_template( [X1|L1], [X1-V1|L2], [X1|L3], [X1-V1|L4] ) :-
        build_template( L1, L2, L3, L4 ).

build_template_aux( [], [], [], [] ).
build_template_aux( [X|L1], [X-V|L2], [X|L3], [X-V|L4] ) :-
        build_template_aux( L1, L2, L3, L4 ).

d_propagate_all( T, T1 ) :-
        content( T1 ),
        \+content( T ),
        retract( content(T1) ),
        writeln([T1, ' removed <<']),
        retract( terminate_propagation ),
        fail.
d_propagate_all( _, _ ).

global_propagate( _ ) :- retract( terminate_propagation ), !.
global_propagate( C1 ) :-
        writeln(['{ Global propagation from ',C1]),
        downward_propagation( C1 ), upward_propagation( C1 ),
        writeln(['} End of global propagation from ',C1]).

upward_propagation( C ) :-
        build_templates( C, T, C1, T1 ), node( C ),
        assert( terminate_propagation ),
        u_propagate_all( T, T1 ), global_propagate( C1 ),
        fail.
upward_propagation( _ ).

u_propagate_all( T, T1 ) :-
        content( T ), \+content( T1 ), retract( content(T) ),
        writeln([T, 'removed <<']), retract( terminate_propagation ), fail.
u_propagate_all( _, _ ).

/*-------------------------------------------------------------------------*/

report(N) :-
        write('Solutions:'), nl, functor( P, dummy, N ), P =.. [_|Solution],
        content( Solution ), write( Solution ), nl, fail.
report(_) :- write(****).

writeln([]) :- nl.
writeln([A|L]) :- write(A), writeln(L).

/*=========================================================*/
```

```
/*=======================================================
```

Program 9.2 : **invasion.plg**
Subject : *Seidel's Invasion* algorithm for solving CSPs
Dynamic Clauses : Clauses of the following predicates will be asserted:

```
        sg_node( N )
        sg_arc( X, Y, Label )
```

where *sg* stands for solution graph, *Label* in *sg_arc* is the label on the arc (X,Y).
Notes : This program assumes that:
(1) the constraint graph is a connected graph;
(2) the problem is specified with the following predicates:

```
        variable( X )
        domain( X, Domain )
        constraint( X, Y, Legal_pairs )
```

where Domain is a list of values; and
Legal_pairs = [Vx1/Vy1, Vx2/Vy2, ...]
where Vxi and Vyi are values for X and Y respectively. If constraint/3 is not defined between variables P and Q, then P and Q are not constrained.
An example problem is attached to the end of the program.

```
=======================================================*/
invasion :-
        retract_all( sg_node(_) ),
        retract_all( sg_arc(_, _) ),
        bagof( X, variable(X), Vars ),
        assert( sg_node( [] ) ),
        invade( [[]], Vars ),
        report.
invasion :- write('There are no variables in this problem.'), nl.

retract_all( P ) :- retract( P ), fail.
retract_all( _ ).

/*
        invade( S1, Vars )
        S1 stands for S(i - 1); Vars is the list of variables to be processed
*/
invade( _, [] ).
invade( S1, [X| Vars] ) :-
        domain( X, Dx ),
        update_sg( S1, X, Dx, Vars, NewNodes, [] ),
        NewNodes \== [],
        invade( NewNodes, Vars ).
invade( _, [X|_] ) :- write('Invasion fails in variable '), write(X), nl.
```

```
/*
        update_sg( OldNodes, X, Dx, Vars, NewNodes, TempNewNodes )
        OldNodes is the nodes in S(i - 1);
        X is the variable currently being processed;
        Dx is the domain of X;
        Vars is the set of variables yet to be processed; it is passed as a parameter
        for updating the Front;
        NewNodes is the nodes in Si (NewNodes is to be returned);
        TempNewNodes is set of NewNodes found so far.
        update_sg/6 processes one OldNode at a time.
*/
update_sg( [], _, _, _, NewNodes, NewNodes ).
update_sg( [CL1| CLs], X, Dx, Vars, NewNodes, Temp ) :-
        update_sg_aux( CL1, X, Dx, Vars, Temp1, Temp ),
        update_sg( CLs, X, Dx, Vars, NewNodes, Temp1 ).

/*
        update_sg_aux( CL, X, Dx, Vars, NewNodes, TempNewNodes )
        CL is the compound label being processed;
        X is the variable currently being processed;
        Dx is the domain of X;
        Vars is the set of variables yet to be processed; it is used here for updating
        the Front;
        NewNodes is the nodes in Si (NewNodes is to be returned);
        TempNewNodes is set of NewNodes found so far;
        update_sg_aux/6 processes one value in Dx at a time.
*/
update_sg_aux( _, _, [], _, NewNodes, NewNodes ).
update_sg_aux( CL, X, [V| Vs], Vars, NewNodes, Temp ) :-
        satisfy_constraints( CL, X-V ), !,
        find_new_front( [X-V| CL], FrontNode, Vars ),
        update_node( FrontNode, Temp, Temp1 ),
        assert( sg_arc(FrontNode, CL, X-V) ),
        update_sg_aux( CL, X, Vs, Vars, NewNodes, Temp1 ).
update_sg_aux( CL, X, [_| Vs], Vars, NewNodes, Temp ) :-
        update_sg_aux( CL, X, Vs, Vars, NewNodes, Temp ).

/*
        satisfy_constraints( CL, X-Vx )
        CL is a compound label;
        X-Vx is a label for variable X and value Vx;
        satisfy_constraints/2 succeeds if <X,Vx> is compatible with all the labels
        in CL.
*/
satisfy_constraints( [], _ ).
satisfy_constraints( [Y-Vy| CL], X-Vx ) :-
        constraint( X, Y, LegalPairs ),
        member( Vx/Vy, LegalPairs ),
```

```
                satisfy_constraints( CL, X-Vx ).
satisfy_constraints( [Y-_| CL], X-Vx ) :-
                \+constraint( X, Y, _ ),
                satisfy_constraints( CL, X-Vx ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

find_new_front( [], [], _ ).
find_new_front( [X-V| L], [X-V| R], Vars ) :-
                constraint( X, Y, _ ),
                member( Y, Vars ), !,
                find_new_front( L, R, Vars ).
find_new_front( [_| L], R, Vars ) :-
                find_new_front( L, R, Vars ).

update_node( Node, L, L ) :- sg_node( Node ).
update_node( Node, L, [Node|L] ) :-
                \+ sg_node( Node ), assert( sg_node(Node) ).

/*-------------------------------------------------------------------------*/

report :-       write('Solutions:'), nl,
                sg_arc( [], Node, Label ),
                trace_sg_arcs( Node, [Label], Solution ),
                write( Solution ), nl,
                fail.
report :- write('****'), nl.

trace_sg_arcs( [], Solution, Solution ).
trace_sg_arcs( Node, CL, Solution ) :-
                Node \== [],
                sg_arc( Node, Node1, Label ),
                trace_sg_arcs( Node1, [Label| CL], Solution ).

/*-------------------------------------------------------------------------*/
/*              An example problem:
variable( w ). variable( x ). variable( y ). variable( z ).
domain( w, [1,2,3] ). domain( x, [1,2,3] ). domain( y, [1,2,3] ). domain( z, [1,2,3] ).
constraint( w, x, [1/2,1/3,2/3] ).
constraint( w, y, [1/2,1/3,2/3] ).
constraint( x, z, [1/1,1/2,1/3,2/2,2/3,3/3] ).
constraint( y, z, [1/1,1/2,1/3,2/2,2/3,3/3] ).
constraint( x, w, [2/1,3/1,3/2] ).
constraint( y, w, [2/1,3/1,3/2] ).
constraint( z, x, [1/1,2/1,3/1,2/2,3/2,3/3] ).
constraint( z, y, [1/1,2/1,3/1,2/2,3/2,3/3] ).
/*=========================================================*/
```

```
/*=======================================================
Program 9.3         :       ab.plg
Subject             :       Essex Solution Synthesis algorithm AB applied to
                            the N-queens problem
Notes               :       The data structure used throughout the program is a
        list of:
                                    [Vars]-[[Val_1], [Val_2], ..., [Val_n]]
                            where Vars is a list of variable, each of Val_1,
                            Val_2, ..., Val_n is a list of values for the variables
                            in Vars.
=======================================================*/
/*
        queens(N, R)
        N               a number specifying how many queens to use
        R               a solution for the N-queens problem
        Problem         List of [Var]-[[Val_1], [Val_2], ..., [Val_n]]
*/
queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, Problem),
        syn(Problem, R),
        report(R).

/*-------------------------------------------------------------------------*/
/*
        range(N, List)
        Given a number N, range creates the List:
                [[1], [2], ..., [N - 1], [N]]
*/
range(N, R) :- range(N, R, []).

range(0, L, L).
range(N, R, L) :- N > 0, N1 is N - 1, range(N1, R, [[N]|L]).

/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L, return as the 3rd argument the Candi-
        dates:
                [[1]-L, [2]-L, ..., [N - 1]-L, [N]-L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(N, L, Result) :-
        setup_candidate_lists(N, L, Result, []).

setup_candidate_lists(0, _, R, R).
setup_candidate_lists(N, L, R, Temp) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R, [[N]-L| Temp]).
```

```
/*-------------------------------------------------------------------------*/
/*
        (predicates in this section are domain independent,
        except for “compatible_values/2”)

        syn(Nodes, Solution)
        Given: Nodes [Vars]-[CompoundLabels]
        where both Vars and CompoundLabels are lists.
        e.g. one of the nodes of order 2 in Nodes could be:
                [1,2]-[[1,2], [1,3], [2,2], [2,4]]
        if this list is combined with another node:
                [2,3]-[[1,2], [1,3], [2,2], [2,4]]
        in Nodes, one should get the following node of order 3:
                [1,2,3]-[[1,2,2], [1,2,4], [2,2,2], [2,2,4]]
*/
syn([Solution], Solution).
syn(Nodes, Solution) :-
        Nodes \= [_],
        Nodes = [Vars-_|_],
        length(Vars, Len),
        writeln([‘Nodes of order ‘,Len,’: ‘,nl,indented_list(Nodes)]),
        syn_nodes_of_current_order(Nodes, Temp),
        syn(Temp, Solution).

syn_nodes_of_current_order([N1,N2|L], [N3| Solution]) :-
        combine(N1, N2, N3), !,
        syn_nodes_of_current_order([N2|L], Solution).
syn_nodes_of_current_order(_, []).

combine([X|_]-Values1, X2-Values2, [X|X2]-CombinedValues) :-
        last(X2, Y),
        bagof(V, compatible_values(X, Y, Values1, Values2, V), CombinedVal-
        ues).
combine([X|L1]-Values1, X2-Values2, [X|X2]-[]) :-
        nl, writeln([‘** No value satisfies all variables ‘,[X|X2],’!!’]),
        writeln([‘Values for ‘,[X|L1],’ are: ‘,Values1]),
        writeln([‘Values for ‘,X2,’ are: ‘,Values2]).

compatible_values(X, Y, Values1, Values2, [Vx|V2]) :-
        member([Vx|V1], Values1),
        member(V2, Values2),
        append(V1, Tail, V2),
        last(Tail, Vy),
        compatible(X-Vx, Y-Vy).

compatible(X-Vx, Y-Vy):-
        Vx =\= Vy,
```

```
        Vy-Vx =\= Y-X,
        Vy-Vx =\= X-Y.

/*-------------------------------------------------------------------------*/
/*      Reporting -- not the core of this program
*/
report(_-[]).
report(Vars-[H| L]) :-
        write('Solution: ('),
        report_aux( Vars, H ),
        report(Vars-L).

report_aux( [], [] ) :- write(')'), nl.
report_aux( [X1|Xs], [V1|Vs] ) :-
        write(X1), write('/'), write(V1),
        (Xs == []; write(', ')), !,
        report_aux( Xs, Vs ).

/*-------------------------------------------------------------------------*/

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([indented_list(H)|L]) :-
        !, indented_list(H),
        writeln(L).
writeln([H|L]) :- write(H), writeln(L).

indented_list([]).
indented_list([H|L]) :- write(H), nl, indented_list(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X,L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

last([X], X).
last([_|L], X) :- last(L, X).

/*=========================================================*/
```

```
/*=========================================================
Program 9.4         :       ap.plg
Subject             :       Essex Solution Synthesis algorithm AP applied to
                            the N-queens problem. This program is modified
                            from ab.plg to allow local propagation
Notes               :       The following data structure is being used:
                                ListOfVariables-ListOfCompoundLabels
                            e.g.:       [1,2]-[[1,3],[1,4],[2,4],[3,1],[4,1],[4,2]]
=========================================================*/
/*
        queens(N, R)
        N               a number specifying how many queens to use
        R               a solution for the N-queens problem
*/
queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, Problem),
        syn(Problem, R),
        report(R).

/*-------------------------------------------------------------------------*/
/*
        range(N, List)
        Given a number N, range creates the List:
                [[1], [2], ..., [N - 1], [N]]
*/
range(N, R) :- range(N, R, []).

range(0, L, L).
range(N, R, L) :- N > 0, N1 is N - 1, range(N1, R, [[N]|L]).

/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N and a list L, return as the 3rd argument the Candidates:
                [[1]-L, [2]-L, ..., [N - 1]-L, [N]-L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(N, L, Result) :-
        setup_candidate_lists(N, L, Result, []).

setup_candidate_lists(0, _, R, R).
setup_candidate_lists(N, L, R, Temp) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R, [[N]-L| Temp]).

/*-------------------------------------------------------------------------*/

        syn(Nodes, Solution)
```

```
        (this predicate is domain independent, except for "allowed")
        Given: Nodes [Vars]-[CompoundLabels] where both Vars and Com-
        poundLabels are lists; e.g. one of the nodes of order 2 in Nodes could be:
                [1,2]-[[1,2], [1,3], [2,2], [2,4]]
        if this list is combined with another node:
                [2,3]-[[1,2], [1,3], [2,2], [2,4]]
        in Nodes, one should get the following node of order 3:
                [1,2,3]-[[1,2,2], [1,2,4], [2,2,2], [2,2,4]]
*/
syn([Solution], Solution).
syn(Nodes, Solution) :-
        Nodes = [Vars-_|L], L\==[],
        length(Vars, Len),
        writeln([‘Nodes of order ‘,Len,’: ‘,nl,indented_list(Nodes)]),
        syn_aux(Nodes, Temp),
        syn(Temp, Solution), !.


syn_aux([N1,N2|L], [N3| Solution]) :-
        combine(N1, N2, N3),
        (L==[], N2=NewN2; L\==[], downward_constrain(N2, N3, NewN2)),
        syn_aux([NewN2|L], Solution).
syn_aux(_, []) .

combine([X|_]-Values1, X2-Values2, [X|X2]-CombinedValues) :-
        last(X2, Y),
        bagof(V, allowed_values(X, Y, Values1, Values2, V), CombinedValues),
        !.
combine([X|L1]-Values1, X2-Values2, [X1|X2]-[]) :-
        nl, writeln([‘** No value satisfies all variables ‘,[X|X2],’!!’]),
        writeln([‘Values for ‘,[X|L1],’ are: ‘,Values1]),
        writeln([‘Values for ‘,X2,’ are: ‘,Values2]).

allowed_values(X, Y, Values1, Values2, [Vx|V2]) :-
        member([Vx|V1], Values1),
        member(V2, Values2),
        append(V1, Tail, V2),
        last(Tail, Vy),
        allowed(X-Vx, Y-Vy).

/*
        domain dependent predicates:
*/
allowed(X-Vx, Y-Vy):-
        Vx =\= Vy,
        Vy-Vx =\= Y-X,
        Vy-Vx =\= X-Y.
```

```
/*
        downward_constrain(X2-V2, X3-V3, X2-NewV2)
*/
downward_constrain(X2-V2, X3-V3, X2-NewV2) :-
        downward_constrain(V2, V3, NewV2),
        (V2 == NewV2;
         V2 \== NewV2, length(V2, M), length(NewV2,N), P is M - N,
         writeln([‘** Node ‘,X3,’-’,V3,’ reduces ‘,P,
         ‘ elements from node ‘, X2, nl,V2,’ --> ‘,NewV2,nl])
        ).

downward_constrain([], CombinedValues, []).
downward_constrain([H|L], CombinedValues, [H|R]) :-
        member_chk([_|H], CombinedValues), !,
        downward_constrain(L, CombinedValues, R).
downward_constrain([H|L], CombinedValues, R) :-
        downward_constrain(L, CombinedValues, R).

/*-------------------------------------------------------------------------*/

/*      Reporting -- not the core of the program
*/
report(_-[]).
report(Vars-[H| L]) :- writeln([‘Solution: ‘,Vars,’-’,H]), report(Vars-L).

writeln([]) :- nl.
writeln([nl|L]) :- nl, !, writeln(L).
writeln([indented_list(H)|L]) :-
        indented_list(H), !,
        writeln(L).
writeln([H|L]) :- write(H), writeln(L).

indented_list([]).
indented_list([H|L]) :- write(H), nl, indented_list(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X,L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

last([X], X).
last([_|L], X) :- L\==[], last(L, X).

member_chk(H, [H|_]).
member_chk(H, [A|L]) :- H \= A, member_chk(H, L).

/*=========================================================*/
```