Lecture 6 – Backtracking

1. Backtracking

Backtracking method:

- generally applicable to problems that have several solutions.
- allows the generation of all solutions to a problem
- depth limited search in the space of solutions to the problem
- exponential as execution time
- general method for solving problems in the class of Constraint Satisfaction Problems (CSP)
- Prolog is suitable for solving CSP class problems
    - the **control structure** used by the Prolog interpreter is based on backtracking

<u>Formalization</u>

- the problem solution is a list vector ($x1$, ..., xn), with $xi \in Di$
- the solution vector is incrementally generated
- we denote by *col* the list that collects a solution to the problem
- we denote by **final** the predicate verifiying wheter the *col* list is a solution to the problem
- we denote by **constraints** the predicate verifying whether the *col* list may lead to a solution of the problem

To determine a solution to the problem, the following cases are possible:

i. if *col* satisfies **final**, then there is a solution to the problem;
ii. otherwise, we extend *col* with an element *e* (which we will call **candidate**) such that col$\cup$*e* satisfies **constraints**

Remarks

- if the generation of the element *e* the *col* collector at point ii will be extended with is non-determinist, then all solutions to the problem will be generated
- there are problems with no final conditions imposed

The general recursive model for generating solutions is:

$$solution(col) =$$
$$\begin{array}{ll} 1.\ col & if\ col\ verifies\ the\ final\ conditions \\ 2.\ solution(col \cup e) & if\ e\ is\ a\ possible\ candidate\ and \\ & \quad col \cup e\ verifies\ the\ continuation\ conditions \end{array}$$

The simplified Prolog code structure follows:

```
% E – a candidate value for the current variable
candidate(E, …) :-
        …

% X – the candidate solution
final(X, …) :-
        …

% E – the candidate value for the current variable
% X – the past variables with assigned values
constraints(E, X, …) :-
        …

% S – the solution array – the solution to the problem
problem(…, S) :-
        solution([], S, …).

% X – the candidate solution – the complete array
solution(X, X, …) :-
        final(X, …).

% X – the past variables with assigned values
% S – the solution array
% E – the candidate value for the current variable
solution(X, S, …) :-
        candidate(E, …),
        constraints(E, X, …),
        solution([E | X], S, …).
```

EXAMPLE 1.1 Write a non-determinist predicate that generates combinations with k ≠ 1 elements of a non-empty set whose elements are non-zero natural numbers, so that the sum of the elements in the combination is a given value S.

? combSum ([3, 2, 7, 5, 1, 6], 3, 9, C). /* flow model (i, i, i, o) – non-determinist */
/* k = 3, S = 9 */
C = [2, 1, 6];
C = [3, 5, 1].

? allCombSum ([3, 2, 7, 5, 1, 6], 3, 9, LC).

LC = [[2, 1, 6], [3, 5, 1]].

For solving this problem, we will give three solutions, the last one based on the backtracking method described above.

Solution 1 We generate the solutions of the problem using direct recursion.

To determine the combinations of a list [E | L] (which has the head E and the tail L) taken K, of given sum S the following cases are possible:

i. if K = 1 and E is equal to S, then [E] is a combination
ii. determine a combination with K elements of the list L, having the sum S;
iii. place the element E on the first position in the combinations with K-1 elements of the list L, of sum S-E (if K> 1 and SE> 0).

The recursive model for generation is:

$combSum(l_1\ l_2 \ldots l_n, k, S) =$
1. (l_1) $\quad$ $if\ k = 1\ and\ l_1 = S$
2. $comb(l_2 \ldots l_n, k, S)$
3. $l_1 \oplus comb(l_2 \ldots l_n, k-1, S-l_1)$ $\quad$ $if\ k > 1\ and\ S - l_1 > 0$

We will use the non-determinist predicate combSum which will generate all combinations. If you want to collect combinations in a list, you can use the findall predicate.

The SWI-Prolog program is as follows:

```
% combSum (L: list, K: integer, S: integer, C: list)
% (i, i, I, o) – non-determinist
combSum ([H | _], 1, H, [H]).
combSum ([_ | T], K, S, C) :-
        combSum (T, K, S, C).
combSum ([H | T], K, S, [H | C]) :-
        K > 1,
        S1 is S-H,
        S1 > 0,
        K1 is K-1,
        combSum (T, K1, S1, C).
```

Solution 2 The combinations with k elements are first generated and then verified whether the sum of elements is S. This solution is not efficient, considering that combinations are generated that may not have sum S.

The following predicates will be used:

- the comb predicate for generating a combination with k elements from a list, predicate described in **lecture 5**
- the predicate sum (L: list of numbers, S: integer), flow model (i, i) that checks if the sum of the elements of the list L is equal to S.

```
% combSum (L: list, K: integer, S: integer, C: list)
% (i, i, i, o) – non-determinist
combSum (L, K, S, C) :-
        comb (L, K, C),
        sum (C, S).
```

Solution 3 We use a non-determinist predicate **candidate**(E: element, L: list), flow model (o, i), which generates all the elements of a list.

? candidate (E, [1, 2, 3]).
E = 1;
E = 2;
E = 3.

$candidate(l_1 l_2 \ldots l_n) =$
1. $l_1 \qquad if\ l\ is\ not\ empty$
2. $candidate(l_2 \ldots l_n)$

```
% candidate (E: item, L: list)
% (o, i) – non-determinist
candidate (E, [E | _]).
candidate (E, [_ | T]) :-
    candidate (E, T).
```

The main predicate will generate a candidate E and will start generating the solution with this element.

```
% combSum (L: list, K: integer, S: integer, C: list)
% (i, i, i, o) – non-determinist
combSum (L, K, S, C) :-
    candidate (E, L),
    combAux (L, K, S, C, 1, E, [E]).
```

The non-determinist auxiliary predicate combAux will generate combinations of a given sum.

```
% combAux (L: list, K: integer, S: integer, C: list, Lg: integer, Sum: integer, Col: list)
% (i, i, i, o, i, i, i) – non-determinist
combAux(_, K, S, C, K, S, C) :- !.
combAux(L, K, S, C, Lg, Sum, [H | T]) :-
    Lg < K,
    candidate(E, L),
    E < H,
    Sum1 is Sum + E,
    Sum1 =< S,
    Lg1 is Lg + 1,
    combAux(L, K, S, C, Lg1, Sum1, [E, H | T]).


% alternative for the second clause
% refactoring – an auxiliary predicate to verify the condition

% condition (L:list, K:integer, S:integer, Lg:integer, Sum:integer, H:integer, E:integer)
% (i, i, i, i, i, i, o) – non-determinist
% generates an E from L that verifies the constraints with the past variables

condition(L, K, S, Lg, Sum, H, E) :-
    Lg < K,
    candidate(E, L),
    E < H,
    Sum1 is Sum + E,
    Sum1 =< S.

% return the solution
combAux(_, K, S, C, K, S, C) :- !.

% get a legal candidate and add it to the partial solution found so far
combAux(L, K, S, C, Lg, Sum, [H|T]):-
    condition(L, K, S, Lg, Sum, H, E),
    Lg1 is Lg+1,
    Sum1 is Sum + E,
    combAux(L, K, S, C, Lg1, Sum1, [E,H|T]).
```

EXAMPLE 1.2. Given a set represented as list, the subsets of even sum formed by odd numbers only should be generated.

? submSP([1, 2, 3, 4, 5], S). /* flow model (i, o) – non-determinist */
S = [1, 3];
S = [1, 5] ;
S = [3, 5];

EXAMPLE 1.3 Given two natural values n (n> 2) and v (v non-zero), a Prolog predicate is required which returns the permutations of the elements 1, 2 ...., n with the property that any two consecutive elements have the difference in absolute value greater than or equal to v.

? permutations (4, 2, P) (*n* = 4, v = 2)
P = [2, 4, 1, 3];
P = [3, 1, 4, 2];
....

? candidate (4, I).
I = 4;
I = 3;
I = 2;
I = 1.

Recursive models

candidate (n) =
1. n
2. candidate (n-1) if n > 1

The following predicates will be used

- the non-determinist predicate **candidate** (N, I) (i, o) generates a solution candidate (a value between 1 and N);
- the non-determinist predicate **permutations_aux** (N, V, LResult, Length, LCollectors) (i, i, o, i, i) collects the elements of a permutation in LCollectors of length Length. Collection will stop when the number of collected items (Length) is n. In this case, LCollectors will contain a solution permutation, and LResult will be bound to LCollectors.
- the non-determinist predicate **permutations** (N, V, L) (i, i, o) generates a solution permutation;
- the predicate **member** (E, L) which tests the membership of an element in a list (to collect in solution only the distinct elements).

```
% candidate (N: integer, I: integer)
% (i, o) – non-determinist
candidate (N, N).
candidate (N, I) :-
        N> 1,
        N1 is N-1,
        candidate (N1, I).

% permutations (N: integer, V: integer, L: list)
% (i, i, o) – non-determinist
permutations (N, V, L) :-
        candidate (N, I),
        permutations_aux (N, V, L, 1, [I]).

% permutations_aux (N: integer, V: integer, L: list, Lg: integer, Col: list)
% (i, i, o, i, i) – non-determinist
permutations_aux (N, _, Col, N, Col) :- !.
permutations_aux(N, V, L, Lg, [H | T]) :-
        candidate (N, I),
        abs (H-I) >= V,
        \+ member (I, [H | T]), % (i, i)
        Lg1 is Lg + 1,
        permutations_aux (N, V, L, Lg1, [I, H | T]).
```

EXAMPLE 1.4 Consider a set of non-null natural numbers represented as a list. Determine all the possibilities to write a number N as a sum of elements from this list.

```
% list=integer*
% candidate(list, integer) (i, o) – non-determinist
% an element possibly to be added in the solution list
candidate([E|_],E).
candidate([_|T],E) :-
        candidate(T,E).

% solution(list,integer,list) (i,i,o) – non-determinist
solution(L, N, Rez) :-
        candidate(L, E),
        E =< N,
        solution_aux(L, N, Rez, [E], E).

% solution_aux(list,integer,list,list,integer) (i,i,o,i,i) – non-determinist
% the fourth parameter collects the solution
% the fifth parameter represents the sum of all elements in the collector
solution_aux(_, N, Rez, Rez, N) :- !.
solution_aux(L, N, Rez, [H | Col], S) :-
        candidate(L, E),
        E < H,
        S1 is S+E,
        S1 =< N,
        solution_aux(L, N, Rez, [E, H | Col], S1).
```

<u>EXAMPLE 1.5</u> THE PROBLEM OF THE THREE HOUSES.

1. The Englishman lives in the first house on the left.
2. In the house immediately to the right of the one where the wolf is, he smokesLucky Strike.
3. The Spaniard smokesKent.
4. The Russian has a horse.

Who smokes LM? Whose dog is it?

We remark that the problem has two solutions:

I.	English	dog	LM
	Spanish	wolf	Kent
	Russian	horse	LS
II.	English	wolf	LM
	Russian	horse	LS
	Spanish	dog	Kent

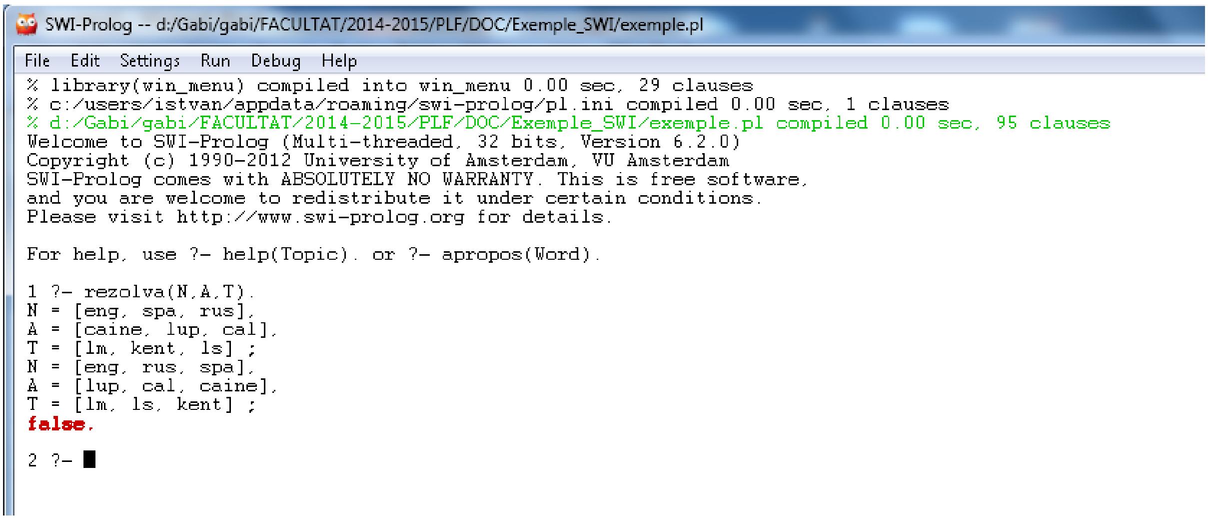
This is a typical constraint satisfaction problem.

We encode the data of the problem and remark that a solution consists of triplets of the form (N, A, T) where:

N belongs to the set **[eng, spa, rus]**
A belongs to the set **[dog, wolf, horse]**
T belongs to the set **[lm, ls, ken]**

We will use the following predicates:

- non-determinist predicate **solve** (N, A, T) (o, o, o) which generates a solution of the problem
- non-determinist predicate **candidates** (N, A, T) (o, o, o) that generates all candidates for solution
- determinist predicate **constraints** (N, A, T) (i, i, i) which verifies whether a candidate for solution satisfies the constraints imposed by the problem
- non-determinist predicate **perm** (L, L1) (i, o) which generates the permutations of the list L



```
% solve - (o, o, o)
solve (N, A, T) :-
        candidates(N, A, T),
        constraints(N, A, T).

% candidates - (o, o, o)
candidates(N, A, T) :-
        perm([eng, spa, rus], N),
        perm([dog, wolf, horse], A),
        perm([lm, kent, ls], T).

% constraints - (i, i, i)
constraints(N, A, T) :-
        aux (N, A, T, eng, _, _, 1),
        aux (N, A, T, _, wolf, _, Nr1),
        right (Nr1, Nr2),
        aux (N, A, T, _, _, ls, Nr2),
        aux (N, A, T, spa, _, kent, _),
        aux (N, A, T, rus, horse, _, _).

% right - (i, o)
right (I, J) :-
        J is I + 1.

% aux (i, i, i, o, o, o, o)
aux ([N1, _, _], [A1, _, _], [T1, _, _], N1, A1, T1,1).
aux ([_, N2, _], [_, A2, _], [_, T2, _], N2, A2, T2,2).
aux ([_, _, N3], [_, _, A3], [_, _, T3], N3, A3, T3,3).

% insert (s)
insert (E, L, [E | L]).
insert (E, [H | L], [H | T]) :-
        insert (E, L, T).

% perm (i, o)
perm ([], []).
perm ([H | T], L) :-
        perm (T, P),
        insert (H, P, L).
```

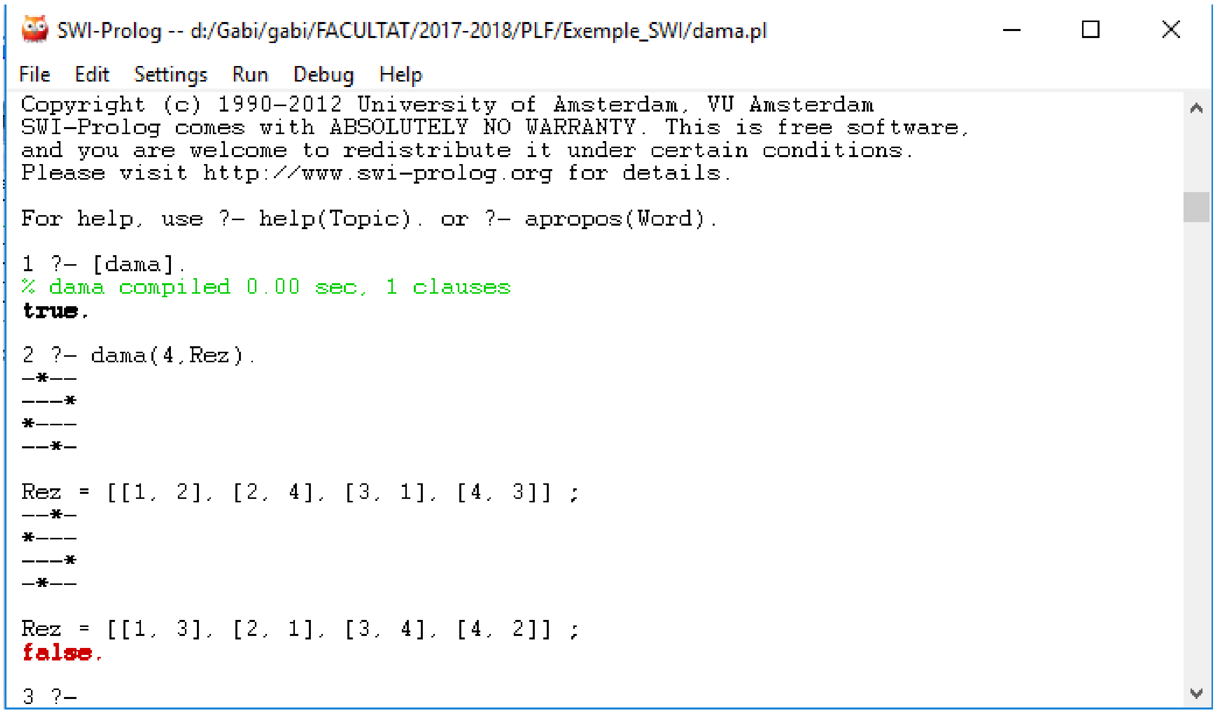
EXAMPLE 1.6 THE PROBLEM OF THE FIVE HOUSES.

Five people live in the five houses in a street. Each has a different profession, animal, favourite drink, and each house is painted in a different colour.

1. The Englishman lives in the red house
2. The Spaniard owns a dog
3. The Norwegian lives in the first house on the left
4. The Japanese is a painter
5. The green house is on the right of the white one
6. The Italian drinks tea
7. The fox is in a house next to the doctor
8. Milk is drunk in the middle house
9. The horse is in a house next to the diplomat
10. The violinist drinks fruit juice
11. The Norwegians house is next to the blue one
12. The sculptor breeds snails
13. The owner of the green house drinks coffee
14. The diplomat lives in the yellow house

Who owns the zebra? Who drinks water?

EXAMPLE 1.7 Arrange N queens on a NxN chessboard so that they do not attack each other.



```
SWI-Prolog -- d:/Gabi/gabi/FACULTAT/2017-2018/PLF/Exemple_SWI/dama.pl
File  Edit  Settings  Run  Debug  Help
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [dama].
% dama compiled 0.00 sec, 1 clauses
true.

2 ?- dama(4,Rez).
-*--
---*
*---
--*-

Rez = [[1, 2], [2, 4], [3, 1], [4, 3]] ;
--*-
*---
---*
-*--

Rez = [[1, 3], [2, 1], [3, 4], [4, 2]] ;
false.

3 ?-
```

```
% (integer, list *) - (i, o) non-determinist
queen (N, Rez) :-
        candidate (E, N),
        queen_aux (N, Rez, [[N, E]], N),
        printSolution (N, Rez).

% (integer, integer) - (o, i) non-determinist
candidate (N, N).
candidate (E, I) :-
        I > 1,
        I1 is I-1,
        candidate (E, I1).

% (integer, list *, list *, integer) - (i, o, i, i) non-determinist
queen_aux (_, Rez, Rez, 1) :- !.
queen_aux (N, Rez, C, Lin) :-
        candidate (Col1, N),
        Lin1 is Lin-1,
        valid (Lin1, Col1, C),
        queen_aux (N, Rez, [[Lin1, Col1] | C], Lin1).
```

```
% (integer, integer, list *) - (i, i, i) determinist
valid (_, _, []).
valid (Lin, Col, [[Lin1, Col1] | T]) :-
        Col =\= Col1,
        DLin is Col-Col1,
        DCol is Lin-Lin1,
        abs (DLin) =\= abs (DCol),
        valid (Lin, Col, T).

% (integer, list *) - (i, i) determinist
printSolution (_, []) :- nl.
printSolution (N, [[_, Col] | T]) :-
        printLine (N, Col),
        printSolution (N, T).

% (integer, char) - (i, o) determinist
character (1, '*') :- !.
character(_,'-').

% (integer, list *) - (i, i) determin.
printLine (0, _) :- nl, !.
printLine (N, Col) :-
        character (Col, C),
        write (C),
        N1 is N-1,
        Col1 is Col-1,
        printLine (N1, Col1).
```

HOMEWORK

1. The **chess knight path** problem: Find the path a knight has to follow on a chess board assuming: (1) legal chess knight moves, (2) the knight touches each cell exactly once, and (3) the knight touches all cells on the chess board.

2. A list of distinct integer elements is given. Generate all subsets with **elements in strictly ascending order**.

3. A list of distinct integer elements is given. Generate all subsets with **k elements in arithmetic progression**.