Logik für Informatiker: PROLOG
Programming Methodology
Generate & Test

Andreas Karwath

&

Wolfram Burgard

- Today‘s lecture:

- Nondeterministic Programs

- Generate and Test

Outline

- In imperative programming languages, the way a solution is obtained is typically encoded in the algorithm.
- Prolog offers an alternative way of solving problems by means of the generate and test paradigm.
- The key idea of such solutions is to describe the entire prolbem by means of a generator, that enumerates candidates for the solution and a test that verifies whether a generated candidate is in fact a proper solution.
- This approach is especially suited for logic puzzles such as n-queens, sudoku etc.
- The typical generate and test situation is

```
solution(X):-
      generate(X),   % generate potential candidates
      test(X).       % check whether it is a solution
```

- Example:

```
    S E N D
  + M O R E
= M O N E Y
```

Find an assignment of different values from {0,…,9} to the variables such that the equation holds.

```
solution(X):-
        generate(X),
        test(X).
```

- Generating potential candidates

- We are interested in an assignment of the different digits to the eight involved variables S, E, N, D, M, O, R, and Y, such that the equation holds.

- One popular way of generating such assignments is to use the remove predicate:

```
remove(X, [X|L], L).
remove(X, [H|T], [H|L]):-
    remove(X, T, L).
```

- Generating all potential assignments
- We start with the list of all assignments [1, 2, …, 9, 0]
- We then remove elements one after the other.

```
generate1([ S, E, N, D,
            M, O, R, E,
         M, O, N, E, Y]):-
    remove(S, [1, 2, 3, 4, 5, 6, 7, 8, 9, 0], L1),
    remove(E, L1, L2),
    remove(N, L2, L3),
    remove(D, L3, L4),
    remove(M, L4, L5),
    remove(O, L5, L6),
    remove(R, L6, L7),
    remove(Y, L7, L8).   % member(Y, L7).
```

- To implement the test condition we need to ensure that every “constraint” is met.

- Each digit in the third row is the sum of the elements in the previous row plus the carry bit modulo 10.

```
test1([ S, E, N, D,
        M, O, R, E,
     M, O, N, E, Y]):-
       Y is (D + E) mod 10,
       C1 is (D + E) // 10,
       E is (R + N + C1) mod 10,
       C2 is (R + N + C1) // 10,
       N is (O + E + C2) mod 10,
       C3 is (O + E + C2) // 10,
       O is (S + M + C3) mod 10,
       M is (S + M + C3) // 10.
```

- Finally, we put everything together:

```
solution1(L):-
   generate1(L),
   test1(L).
```

Answers obtained from the program:

```
?- solution1(L).
L = [2, 8, 1, 7, 0, 3, 6, 8, 0|...]
L = [2, 8, 1, 9, 0, 3, 6, 8, 0|...] ;
L = [3, 7, 1, 2, 0, 4, 6, 7, 0|...] ;
L = [3, 7, 1, 9, 0, 4, 5, 7, 0|...]
```

- If we want to make sure that neither M or S are 0:

```
test1([ S, E, N, D,
        M, O, R, E,
     M, O, N, E, Y]):-
       S =\= 0,
       M =\= 0,
       Y is (D + E) mod 10,
       C1 is (D + E) // 10,
       E is (R + N + C1) mod 10,
       C2 is (R + N + C1) // 10,
       N is (O + E + C2) mod 10,
       C3 is (O + E + C2) // 10,
       O is (S + M + C3) mod 10,
       M is (S + M + C3) // 10.
```

```
[9, 5, 6, 7, 1, 0, 8, 5, 1, 0, 6, 5, 2]
```

- Typical generators

- For permutations:
```
L = […]
permutation(L, L1).
```

- For assignments of different values:
```
L1 = […]
remove(X1, L1, L2),
remove(X2, L2, L3), …
```

- For arbitrary assignments:
```
L = […],
member(X1, L),
member(X2, L), …
```

- For sub-sequences:
```
L = […],
append(L1, L2, L),
append(L3, L4, L2).
```

- Problems with the generate and test paradigm

- As problems are solved by enumerating all potential combinations, the search space often becomes exponentially large.

- One common solution is to interleave the generate and test processes.

- This leads to more efficient programs that can handle substantially larger problems…

- Reordering the generate and test predicates in a proper way yields:

```
solution1Fast([ S, E, N, D, M, O, R, E, M, O, N, E, Y]):-
  remove(D, [1, 2, 3, 4, 5, 6, 7, 8, 9, 0], L1),
  remove(E, L1, L2),
  remove(Y, L2, L3),
  Y is (D + E) mod 10,
  C1 is (D + E) // 10,
  remove(N, L3, L4),
  remove(R, L4, L5),
  E is (R + N + C1) mod 10,
  C2 is (R + N + C1) // 10,
  remove(O, L5, L6),
  N is (O + E + C2) mod 10,
  C3 is (O + E + C2) // 10,
  remove(S, L6, L7),
  remove(M, L7, L8),
  O is (S + M + C3) mod 10,
  M is (S + M + C3) // 10.
```

- Note: Not necessarily optimal, when values of certain variables are known beforehand.

- The magic number:

- Find the 9-digit number containing all digits 1, …, 9 such that the sub-number consisting of the first n digits can be divided by n.

- Example: 123456789 is no solution, since
  - 1 can be divided by 1,
  - 12 can be divided by 2,
  - 123 is a multiple of 3, but
  - 1234 cannot be divided by 4.

- Generate and Test Solution:

```
magic_number([X1, X2, X3, X4, X5, X6, X7, X8, X9]):-
  permutation([1, 2, 3, 4, 5, 6, 7, 8, 9],
              [X1, X2, X3, X4, X5, X6, X7, X8, X9]),
  0 is X2 mod 2,
  0 is (X1 + X2 + X3) mod 3,
  0 is (X3 * 10 + X4) mod 4,
  0 is X5 mod 5,
  0 is X6 mod 2,
  0 is (X1 + X2 + X3 + X4 + X5 + X6) mod 3,
  0 is ((((((((X1 * 10) + X2) * 10 + X3) * 10)
      + X4) * 10 + X5) * 10 + X6)*10 + X7) mod 7,
  0 is ((X6 * 10 + X7) * 10 + X8) mod 8.
```

- Also here the interleaving of generate and test would substantially decrease the size of the search tree.

- Magic number implementation with early evaluations of tests:

```
magic_number_fast([X1, X2, X3, X4, X5,
                   X6, X7, X8, X9]):-
 L = [1, 2, 3, 4, 5, 6, 7, 8, 9],
 remove(X1, L, L1), remove(X2, L1, L2), 0 is X2 mod
 2,
 remove(X3, L2, L3), 0 is (X1 + X2 + X3) mod 3,
 remove(X4, L3, L4), 0 is (X3 * 10 + X4) mod 4,
 remove(X5, L4, L5), 0 is X5 mod 5,
 remove(X6, L5, L6), 0 is X6 mod 2,
 0 is (X1 + X2 + X3 + X4 + X5 + X6) mod 3,
 remove(X7, L6, L7),
 0 is ((((((((X1 * 10) + X2) * 10 + X3) * 10) + X4) *
 10 + X5) * 10 + X6)*10 + X7) mod 7,
 remove(X8, L7, [X9]),
 0 is ((X6 * 10 + X7) * 10 + X8) mod 8.
```

- Unfortunately, the point in time when a test can be executed may depend on the variables instantiated in the query.

- Is there a solution to the SEND+MORE+MONEY example, in which E is bound to 3?

```
:- solution([S, 3, N, D, M, O, R, E,
             M, O, N, E, Y]).
```

- Is there a magic number with digits 34 at positions 3 and 4?

- Eventually, some tests can be evaluated earlier.

- Co-routining mechanism in Prolog

- Some Prolog systems provide mechanism to delay the evaluation of literals until a given condition is met.

- SWI-Prolog, for example, includes the predicate when(Condition, Goal), which executes Goal when Condition is true.

- Example:

```
when(ground(D + E),Y is (D + E) mod 10)
```

- This Co-routining mechanism allows to mix the order of generate and test predicates and is useful in the context of numerical expressions:

```
:- when(ground(X), Y is X * 10), X=5.
```

- The when-declaration delays the Goal until X is ground.

- Accordingly this query succeeds with the answer substitutions X=5 and Y=10. The following query, in contrast, produces a runtime error.

```
:- Y is X * 10, X=5.
```

- The advantage of co-routining is that we can place the tests in front of the predicate generating potential solutions.
- Each test is then only evaluated when possible. All non-ground tests are delayed.
- Accordingly, if the user provides additional constraints, potentially available constraints are evaluated immediately.

- SEND+MORE=MONEY with co-routining:

```
solution1Delay(L):-
  test1Delay(L),
  generate1(L).

test1Delay([ S, E, N, D, M, O, R, E,
             M, O, N, E, Y]):-
  when(ground(D + E), (Y is (D + E) mod 10,
                               C1 is (D + E) // 10)),
  when(ground(R + N + C1), (E is (R + N + C1) mod 10,
                               C2 is (R + N + C1) // 10)),
  when(ground(O + E + C2), (N is (O + E + C2) mod 10,
                               C3 is (O + E + C2) // 10)),
  when(ground(S + M + C3), (O is (S + M + C3) mod 10,
                               M is (S + M + C3) // 10)).
```

- magic number with co-routining:

```
magic_number_delay([X1, X2, X3, X4, X5, X6, X7, X8, X9]):-
  when(ground(X2), 0 is X2 mod 2),
  when(ground(X1 + X2 + X3), 0 is (X1 + X2 + X3) mod 3),
  when(ground(X3+X4), 0 is (X3 * 10 + X4) mod 4),
  when(ground(X5), 0 is X5 mod 5),
  when(ground(X6), 0 is X6 mod 2),
  when(ground(X1 + X2 + X3 + X4 + X5 + X6),
              0 is (X1 + X2 + X3 + X4 + X5 + X6) mod 3),
  when(ground(X1 + X2 + X3 + X4 + X5 + X6 + X7),
              0 is ((((((((X1 * 10) + X2) * 10 + X3) * 10) +
              X4) * 10 + X5) * 10 + X6)*10 + X7) mod 7),
  when(ground(X6+X7+X8),
              0 is ((X6 * 10 + X7) * 10 + X8) mod 8),
  permutation([1, 2, 3, 4, 5, 6, 7, 8, 9],
  [X1, X2, X3, X4, X5, X6, X7, X8, X9]).
```

- Some timings:

```
?- time((magic_number(L),fail)).
% 3,190,996 inferences

?- time((magic_number_fast(L),fail)).
% 3,541 inferences

?- time((magic_number_delay(L),fail)).
% 40,325 inferences

?- time((magic_number([X1, X2, X3, X4, X5, 7, X7, X8, X9]),fail)).
% 461,622 inferences

?- time((magic_number_fast([X1, X2, X3, X4, X5, 7, X7, X8,
  X9]),fail)).
% 3,037 inferences

?- time((magic_number_delay([X1, X2, X3, X4, X5, 7, X7, X8,
  X9]),fail)).
% 45 inferences
```

- Application to Sudoku

- Find an assignment of the digits 1-9 to the free cells in a 9x9 grid such that every line and every column and every bold 3x3 sub-square contains all nine digits.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- generate

```
generate([L1, L2, L3, L4, L5, L6, L7, L8, L9]):-
    Numbers = [1,2,3,4,5,6,7,8,9],
    permutation(Numbers, L1),
    permutation(Numbers, L2),
    permutation(Numbers, L3),
    permutation(Numbers, L4),
    permutation(Numbers, L5),
    permutation(Numbers, L6),
    permutation(Numbers, L7),
    permutation(Numbers, L8),
    permutation(Numbers, L9).
```

- Test (1)

```
test([L1, L2, L3, L4, L5, L6, L7, L8, L9]):-
  test_columns(L1, L2, L3, L4, L5, L6, L7, L8, L9),
  test_blocks(L1,L2,L3),
  test_blocks(L4,L5,L6),
  test_blocks(L7,L8,L9).

test_columns([],[],[],[],[],[],[],[],[]).
test_columns([H1|T1],[H2|T2],[H3|T3],[H4|T4],[H5|T5],
             [H6|T6],[H7|T7],[H8|T8],[H9|T9]):-
  allunequal([H1,H2,H3,H4,H5,H6,H7,H8,H9]),
  test_columns(T1,T2,T3,T4,T5,T6,T7,T8,T9).

test_blocks(
  [X1, X2, X3, X4, X5, X6, X7, X8, X9],
  [Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9],
  [Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, Z9]):-
  allunequal([X1, X2, X3, Y1, Y2, Y3, Z1, Z2, Z3]),
  allunequal([X4, X5, X6, Y4, Y5, Y6, Z4, Z5, Z6]),
  allunequal([X7, X8, X9, Y7, Y8, Y9, Z7, Z8, Z9]).
```

- Test (2)

```
allunequal([]).
allunequal([H|T]):-
  not_contained(H, T),
  allunequal(T).

not_contained(_X, []).
not_contained(X, [H|T]) :-
  when(ground((X,H)), X =\= H),
  not_contained(X, T).
```

- The final program

```
sudoku(L):-
  test(L),
  generate(L).
```

- Example

```
my_sudoku([
  [_, 8, _, 7, _, _, 6, 3, _],
  [5, _, _, _, _, _, 1, _, _],
  [_, _, _, 8, 2, 3, _, _, _],
  [_, 1, _, _, 7, _, _, 9, _],
  [_, _, 3, _, _, _, 5, _, _],
  [_, 4, _, _, 9, _, _, 2, _],
  [_, _, _, 2, 3, 5, _, _, _],
  [_, _, 4, _, _, _, _, _, 3],
  [_, 5, 9, _, _, 8, _, 7, _]
    ]).
```

- Query: `:-my_sudoku(L), sudoku(L).`

- The hardest sudoku of the world (USA Today)

8	5				2	4		
7	2							9
		4						
			1		7			2
3		5				9		
	4							
				8			7	
	1	7						
				3	6		4	

- Summary:

- Generate & Test is a powerful programming methodology for solving constraint systems

- Care has to be taken to avoid overly large search trees

- This can be achieved by interleaving tests with the generation of candidates

- The co-routining mechanism allows to automatically interleave generate and test and performs tests “whenever possible.”