Pràctica 3: Codificació en SAT

Lògica en la Informàtica

FIB

Antoni Lozano

Q2 2023–2024

Objectius

Aquesta pràctica té com a objectius:

- Generar la codificació de problemes en SAT mitjançant un programa en Prolog
- Aprendre a utilitzar la llibreria de Prolog que es proporciona com a ajut per la codificació
- Fer ús d'un *SAT solver* (**kissat**) per resoldre problemes combinatoris

Referències

Com a guia d'estudi teniu

- Els fitxers **miSudoku.pl** i **miSudokuInput1**, que serveixen d'exemple per la resta de problemes
- El **Makefile** que es proporciona
- Aquestes transparències

Codificació en SAT

Codificació en SAT

El programa

1. genera i desa en un fitxer `infile.cnf` les clàusules corresponents a les restriccions del problema
2. crida al *SAT solver* kissat amb `infile.cnf` i desa el resultat en un fitxer `model`
3. obté un model simbòlic M, que conté les variables certes de la solució en format simbòlic
4. mostra la solució de forma llegible amb `displaySol`

Codificació en SAT

Per codificar un problema, només cal modificar tres punts de l'esquema:

1. Definir les *SAT variables*, que poden venir ja donades totalment o parcial
2. Generar les clàusules corresponents a les restriccions, és a dir, completar el predicat `writeClauses/0`
   (No confondre amb `writeOneClause/1`, que afegeix una clàusula)
3. Escriure o completar `displaySol(M)`, on `M` és la llista de variables simbòliques (*SAT variables*) que són certes en el model retornat per kissat

Codificació en SAT

Exemples

```
%% 2. Predicat between

%% between(L, U, X) és cert quan X és un enter entre els enters L i U.

%% ?- between(1, 3, 2).
%% true.

%% ?- between(1, 3, X), write(X), nl, fail.
%% 1
%% 2
%% 3
%% false.
```

Exemples

```
%% 3. Predicat findall

%% findall(X, G, L) és cert quan L és la llista del X per als que
%% (hi ha manera d'instanciar les altres variables, si n'hi ha)
%% l'objectiu G és cert

% f(a, b, c).
% f(a, b, d).
% f(b, c, e).
% f(b, c, f).
% f(c, c, g).

%% ?- findall(C, f(A, B, C), Cs).
%% Cs = [c, d, e, f, g].

%% ?- findall(C, f(a, B, C), Cs).
%% Cs = [c, d].
```

Exemples

```
%% Hi ha altres predicats semblants al findall: setof, bagof. Però tenen
%% el problema que, si no hi ha solucions per a G, aleshores fallen, en
%% lloc de considerar que L ha de ser la llista buida.
%% Això sol donar problemes, i no en recomanem l'ús.


%% ?- findall(A, f(A, b, C), As).
%% As = [a, a].

%% Pot ser que hi hagi solucions repetides!
%% Si no volem repeticions, podem fer servir el predicat sort,
%% que ordena *i elimina repeticions*:

%% ?- | sort([1,3,2], X).
%% X = [1, 2, 3].

%% ?- sort([1,3,1,3,2,2], X).
%% X = [1, 2, 3].
```

Exemples

```
%% Més exemples, combinant findall i between:

%% ?- findall(X, (between(1,7,X), 1 is X mod 2), L).
%% llista dels senars entre 1 i 7

%% L = [1, 3, 5, 7].

%% ?- findall(Y, (between(1,7,X), 1 is X mod 2, Y is X*X), L).
%% llista dels quadrats dels senars entre 1 i 7

%% L = [1, 9, 25, 49].
```

Codificació en SAT

Sudoku

```
symbolicOutput(0).  % set to 1 for DEBUGGING: to see symbolic output only; 0 otherwise.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% A sudoku is a logic-based, combinatorial number-placement puzzle that
%% uses a partially filled 9x9 grid. The objective is to fill the grid
%% with the digits 1 to 9, so that each column, each row, and each of the
%% nine 3x3 blocks contain only one of each digit.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

llista de llistes

solució única

```
%%%%%%% begin input example sudoku1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% entrada([ [-,4,-,  -,-,-,  -,1,-],      %   Solution shown      6 4 3  9 7 5  2 1 8
%%          [-,-,8,  -,3,-,  9,-,-],      %     for this          2 7 8  4 3 1  9 5 6
%%          [-,-,-,  6,8,2,  -,-,-],      %   input example:      5 1 9  6 8 2  3 4 7
%%                                        %
%%          [3,2,-,  -,6,-,  -,7,9],      %                       3 2 5  8 6 4  1 7 9
%%          [-,-,7,  -,-,-,  4,-,-],      %                       1 8 7  3 5 9  4 6 2
%%          [9,6,-,  -,1,-,  -,8,3],      %                       9 6 4  2 1 7  5 8 3
%%                                        %
%%          [-,-,-,  7,9,8,  -,-,-],      %                       4 5 2  7 9 8  6 3 1
%%          [-,-,1,  -,2,-,  7,-,-],      %                       8 3 1  5 2 6  7 9 4
%%          [-,9,-,  -,-,-,  -,2,-] ]).   %                       7 9 6  1 4 3  8 2 5
%%
%%%%%%% end input example sudoku1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Sudoku

```
%%%%%%% ========================================================================================
%
% miSudoku.pl:  simple example of our LI Prolog method for solving problems using a SAT solver.
%
% It generates the SAT clauses, calls the SAT solver, and shows the solution. Just specify:
%       1. SAT Variables
%       2. Clause generation
%       3. DisplaySol: show the solution.
%
%%%%%%% ========================================================================================


%%%%%%% Some helpful definitions to make the code cleaner: ==================================

row(I) :- between(1,9,I).
col(J) :- between(1,9,J).
val(K) :- between(1,9,K).

blockID(Iid,Jid) :- member(Iid,[0,1,2]), member(Jid,[0,1,2]).    %there are 9 blocks: 0-0 1-0 ...2-2
squareOfBlock( Iid,Jid, I,J ) :- row(I), col(J), Iid is (I-1) // 3,  Jid is (J-1) // 3.

%%%%%%% End helpful definitions ============================================================
```

Sudoku

```
%%%%%%% 1. SAT Variables: ------------------------------------------------------------

% x(I,J,K) means "square IJ gets value K",    1<=i<=9, 1<=j<=9, 1<=k<=9   9^3= 729 variables
satVariable( x(I,J,K) ) :- row(I), col(J), val(K).


%%%%%%% 2. Clause generation for the SAT solver: ----------------------------------------

writeClauses :-
    filledInputValues,          % for each filled-in value of the input, add a unit clause
    eachIJexactlyOneK,          % each square IJ gets exactly one value K
    eachJKexactlyOneI,          % each column J gets each value K in exactly one row I
    eachIKexactlyOneJ,          % each row    I gets each value K in exactly one column J
    eachBlockEachKexactlyOnce, % each 3x3 block gets each value K exactly once.
    true,!.                     % this way you can comment out ANY previous line of writeClauses
writeClauses :- told, nl, write('writeClauses failed!'), nl,nl, halt.
```

Sudoku

```
%% We can use writeClause to write any kind of clauses for the SAT solver, any list of positive and
%%     negative literals, such as:  writeOneClause([ -x(1,2,3),  x(2,2,3),  x(4,2,3), -x(5,5,6) ]).
%% We can also generate *constraints* for the SAT solver: exactly(K,Lits), atLeast(K,Lits), atMost(K,Lits).
%%     Look at the library below to see how these constraints generate the necessary SAT clauses to encode them.
%%     For example, exactly 1 of {x1,x2,...x9} is equivalent to:
%%           - at least 1 of {x1,x2,...x9} can be encoded by a single clause: x1 v...v x9
%%           - at most  1 of {x1,x2,...x9} by 36 binary clauses:   -x1 v -x2,   -x1 v -x3,   ...   -x8 v -x9.

%% The Prolog predicate nth1(I,L,E)  means:   "the Ith element of the list L is E"
%% Please understand the Prolog mechanism we use for generating all clauses: one line with "fail", and another one below:
filledInputValues :- entrada(Sud), nth1(I,Sud,Row), nth1(J,Row,K), integer(K), writeOneClause([ x(I,J,K) ]), fail.
filledInputValues.
```

Sudoku

```
%% The Prolog predicate  findall(X, Cond, L)   means:   "L = { X | Cond }"
eachIJexactlyOneK :- row(I), col(J), findall( x(I,J,K), val(K), Lits ), exactly(1,Lits), fail.
eachIJexactlyOneK.
%% this generates, for example,  exactly( 1, [ x(1,1,1), x(1,1,2), ... x(1,1,9) ])

eachJKexactlyOneI :- col(J), val(K), findall( x(I,J,K), row(I), Lits ), exactly(1,Lits), fail.
eachJKexactlyOneI.

eachIKexactlyOneJ :- row(I), val(K), findall( x(I,J,K), col(J), Lits ), exactly(1,Lits), fail.
eachIKexactlyOneJ.

eachBlockEachKexactlyOnce :- blockID(Iid,Jid),
        val(K), findall( x(I,J,K), squareOfBlock(Iid,Jid,I,J), Lits ), exactly(1,Lits), fail.
eachBlockEachKexactlyOnce.
```

Sudoku

```
%%%%%%  3. DisplaySol: show the solution. Here M contains the literals that are true in the model:

% displaySol(M) :- nl, write(M), nl, nl, fail.
displaySol(M) :- nl, row(I), nl, line(I), col(J), space(J), member(x(I,J,K), M ), write(K), write(' '), fail.
displaySol(_) :- nl,nl.

line(I) :-member(I,[4,7]), nl,!.
line(_).
space(J) :-member(J,[4,7]), write(' '),!.
space(_).

%%%%%% ================================================================================
```

Codificació en SAT

Ús de la llibreria

Des de l'intèrpret de Prolog, la generació de clàusules s'inicia amb la consulta

```
?- main.
```

o bé, si l'entrada es pot parametritzar,

```
?- main(<fitxer d'entrada>).
```

Per veure les clàusules generades en format simbòlic, canvieu al fitxer `symbolicOutput(0)` per `symbolicOutput(1)`.

(El predicat es troba normalment a l'inici)

Predicats de generació de clàusules

- `writeOneClause(L)`: genera la clàusula donada per la llista `L`
- Cardinalitat. Generació de clàusules equivalents a
  - `exactly(K,L)`: exactament `K` literals de `L` són certs
  - `atMost(K,L)`: com a màxim `K` literals de `L` són certs
  - `atLeast(K,L)`: com a mínim `K` literals de `L` són certs
- Equivalència. Genera clàusules que estableixen que la variable `V` equival a
  - `expressOr(V,L)`: la disjunció dels literals de `L`
  - `expressAnd(V,L)`: la conjunció dels literals de `L`

Predicat `exactly`

Està definit com:

```
exactly(K,Lits):-
        symbolicOutput(1),
        write(exactly(K,Lits)), nl, !.

exactly(K,Lits):-              % cas general
        atLeast(K,Lits),
        atMost(K,Lits), !.
```

És a dir, amb

- `symbolicOutput(1)`, s'escriu la restricció lògica que estem afegint com a entrada del *SAT solver* i
- `symbolicOutput(0)`, es generen clàusules que asseguren que almenys `K` literals de `Lits` són certs i que com a màxim `K` literals de `Lits` són certs.

Predicat `atMost`

```
atMost(K,Lits) :-
        symbolicOutput(1),
        write(atMost(K,Lits)), nl, !.

atMost(K,Lits) :-
        negateAll(Lits,NLits),
        K is K + 1,
        subsetOfSize(K1,NLits,Clause),
        writeOneClause(Clause), fail.

atMost(_,_).
```

Com abans, si tenim `symbolicOutput(0)`, s'assegura que

- en cada subconjunt de `K` literals, almenys un literal és fals.

Predicat `atLeast`

```
atLeast(K,Lits) :-
        symbolicOutput(1),
        write(atMost(K,Lits)), nl, !.

atLeast(K,Lits) :-
        length(Lits,N),
        K1 is N - K + 1,
        subsetOfSize(K1,Lits,Clause),
        writeOneClause(Clause), fail.

atLeast(_,_).
```

Es basa en la propietat següent:

- almenys `K` literals d'un total de `N` són certs si i només si en cada subconjunt de `N-K+1` literals hi ha almenys un literal cert.