UTF-8

Programación Lógica, Automata Celular

Raul Munoz Davila, C20M063

Table of Contents

Este programa es una implementación del autómata celular en Prolog. Un autmata celular esta formado por una cinta infinita (por ambos extremos) que contiene clulas, las cuales tienen siempre un color dado, y un conjunto de reglas que determinan cmo evoluciona el estado de la cinta: cmo cambian de color sus clulas. La evolucin del autmata consiste en aplicar las reglas a un estado de la cinta para llegar a un nuevo estado, moidficando los colores de las clulas. Por simplicidad, utilizaremos solo el blanco y el negro, representados por o y x , respectivamente.

```
color(o).
color(x).
```

El nuevo color de una clula depende solo de los colores (originales) de sus dos clulas vecinas, a izquierda y derecha. Gracias a esto las reglas se pueden definir a partir de solo tres clulas.

```
rule(o,o,o,_1,o).
rule(x,o,o,r(A,_1,_2,_3,_4,_5,_6),A) :-
    color(A).
rule(o,x,o,r(_1,B,_2,_3,_4,_5,_6),B) :-
    color(B).
rule(o,o,x,r(_1,_2,C,_3,_4,_5,_6),C) :-
    color(C).
rule(x,o,x,r(_1,_2,_3,D,_4,_5,_6),D) :-
    color(D).
rule(x,x,o,r(_1,_2,_3,_4,E,_5,_6),E) :-
    color(E).
rule(o,x,x,r(_1,_2,_3,_4,_5,F,_6),F) :-
    color(F).
rule(x,x,x,r(_1,_2,_3,_4,_5,_6,G),G) :-
    color(G).
```

Los estados deben seguir una serie de restricciones:

Los estados deben empezar y terminar por clulas blancas ([o,x,o,x,o] es un estado admitido mientras que [x,o,x] no lo debe ser).

El estado resultante debe contener siempre dos clulas ms que el estado inicial. Por simplicidad no consideramos representaciones minimizadas (es decir, si [o,x,o] evolucionara a [o,o,x,o,o] que es equivalente no debemos eliminar las celulas redundantes).

El predicado debe fallar si no se cumple la especificacin dada mostrada en los puntos anteriores. Para programar el predicado cells/3 que realiza un paso de evolucion dado un estado inicial y un conjunto de reglas. Se han creado las siguientes predicados auxiliares:

Definición de los números naturales:

```
natural(0).
natural(s(X)) :-
    natural(X).


?- natural(X).
X = 0 ? ;
X = s(0) ?
yes
?-
```

Definicón de la suma de dos números naturales:

```
suma(0,X,X) :-
    natural(X).
suma(s(Y),X,s(Z)) :-
    suma(Y,X,Z).
```

```
?- suma(0,s(0),Z).
Z = s(0) ?
yes
?-
```

Definición de los números pares:

```
par(0).
par(s(X)) :-
    \+par(X).

?- par(X).
X = 0 ? ;
X = s(s(0)) ?
yes
?-
```

Definición de los números impares:

```
impar(s(0)).
impar(s(X)) :-
    par(X).

?- impar(X).
X = s(0) ? ;
X = s(s(s(0))) ?
yes
?-
```

Calculo de la longitud de una lista:

```
?- longitud([o,x,o],X).
X = s(s(s(0))) ?
yes
?-
```

Comprobacion de que la diferencia de longitudes entre 2 listas es 2:

```
dif2(L1,L2) :-
    L1\=[],
    L1\=[_1],
    L1\=[_2,_3],
    longitud(L1,N1),
    longitud(L2,N2),
    suma(N1,s(s(0)),N2).

?- dif2([o,x,o],[o,x,o,x,o]).
yes
?- dif2([o,x,o],[o,x,o,x]).
no
?-
```

Comprobacion de que una lista termina en o:

```
termina([x|L]) :-
    termina(L).
termina([o|L]) :-
```

```
    termina(L).
termina([o]).

?- termina([o,x,o]).
yes
?- termina([o,x,x]).
no
?-
```

Comprobacion de que una lista empieza y termina por o:

```
empieza_termina([o|L]) :-
    termina(L).
empieza_termina([o]).

?- empieza_termina([o,x,o]).
yes
?- empieza_termina([x,o,x]).
no
?- empieza_termina([o,x,o,x,x]).
no
?- empieza_termina([x,o,x,o]).
no
?-
```

Predicado auxiliar para aplicar una regla a una lista:

```
aplicar_regla(o,[o,N|INICIAL],REGLA,[o,Y|FINAL]) :-
    rule(o,o,N,REGLA,Y),
    aplicar_regla(o,[N|INICIAL],REGLA,FINAL).
aplicar_regla(P,[E,N|INICIAL],REGLA,[Y|FINAL]) :-
    rule(P,E,N,REGLA,Y),
    aplicar_regla(E,[N|INICIAL],REGLA,FINAL).
aplicar_regla(P,[o],REGLA,[Y,o]) :-
    rule(P,o,o,REGLA,Y).

?- aplicar_regla(o,[o,x,o], r(x,x,x,o,o,x,o), Cells).
Cells = [o,x,x,x,o] ? ;
no
?-
```

De este modo cells/3 queda de la siguinte manera, donde primero se aplica la regla en cuestion y luego se comprueban las restricciones del formato:

```
cells(INICIAL,REGLA,FINAL) :-
    aplicar_regla(o,INICIAL,REGLA,FINAL),
    dif2(INICIAL,FINAL),
    empieza_termina(FINAL).

?- cells([o,x,o], r(x,x,x,o,o,x,o), Cells).
Cells = [o,x,x,x,o] ? ;
no
?-
```

Para el predicado evol/3 se realiza una llamada recursiva a si mismo, donde se ve el estado anterior del automata y una llamada a cells para comprobar que es el estado anterior del automata y una llamada a cells para comprobar que es el estado siguiente del automata:

```
evol(0,_1,[o,x,o]).
evol(s(N),REGLA,FINAL) :-
    evol(N,REGLA,INICIAL),
    cells(INICIAL,REGLA,FINAL).

?- evol(N, r(x,x,x,o,o,x,o), Cells).
Cells = [o,x,o],
N = 0 ? ;
Cells = [o,x,x,x,o],
N = s(0) ? ;
Cells = [o,x,x,o,o,x,o],
N = s(s(0)) ? ;
Cells = [o,x,x,o,x,x,x,x,o],
N = s(s(s(0))) ? ;
Cells = [o,x,x,o,o,x,o,o,o,x,o],
N = s(s(s(s(0)))) ?
yes
?-
```

Para el predicado auxiliar steps/2 se compreuba que el estado cells es correcto y luego se calcula si es posible el numero de pasos que se necesitan para llegar a el:

```
steps([o,x,o],0).
steps(FINAL,N) :-
    longitud(FINAL,L),
    impar(L),
    empieza_termina(FINAL),
    evol(N,_1,FINAL).
steps(Goal,_1) :-
    call(Goal).

?- steps([o,x,x,o,o,x,o],N).
N = s(s(0)) ?
yes
?-
```

Este predicado se usa en el predicado ruleset para comprobar que el conjunto de reglas es correcto:

```
ruleset(RuleSet,Cells) :-
    steps(Cells,N),
    evol(N,RuleSet,Cells).

?- ruleset(RuleSet , [o,x,x,o,o,x,o,o,o,o,x,o,o,x,o]).
RuleSet = r(x,x,x,o,o,x,o) ?
yes
?- ruleset(RuleSet , [o,x,x,o,o,x,o,o,o,o,x,o,x,x,o]).
no
?-
```

0.1 Usage and interface

- **Library usage:**
  `:- use_module(/mnt/c/Users/UsuarioPC/Desktop/GITHUB_REP/RAUL/prolog/automataCelular/code.pl).`
- **Exports:**
  - *Predicates:*
    `author_data/4`, `cells/3`, `evol/3`, `steps/2`, `ruleset/2`.
  - *Properties:*
    `color/1`, `rule/5`, `natural/1`, `suma/3`, `par/1`, `impar/1`, `longitud/2`, `dif2/2`, `termina/1`, `empieza_termina/1`, `aplicar_regla/4`.

0.2 Documentation on exports

author_data/4: PREDICATE

No further documentation available for this predicate.

color/1: PROPERTY

Usage: `color(X)`

es el color de la celula (x u o).

```
color(o).
color(x).
```

rule/5: PROPERTY

Usage: `rule(X,Y,Z,R,A)`

La regla es una regla de la forma r(A,B,C,D,E,F,G) donde A,B,C,D,E,F,G son colores.

```
rule(o,o,o,_1,o).
rule(x,o,o,r(A,_1,_2,_3,_4,_5,_6),A) :-
    color(A).
rule(o,x,o,r(_1,B,_2,_3,_4,_5,_6),B) :-
    color(B).
rule(o,o,x,r(_1,_2,C,_3,_4,_5,_6),C) :-
    color(C).
rule(x,o,x,r(_1,_2,_3,D,_4,_5,_6),D) :-
    color(D).
rule(x,x,o,r(_1,_2,_3,_4,E,_5,_6),E) :-
    color(E).
rule(o,x,x,r(_1,_2,_3,_4,_5,F,_6),F) :-
    color(F).
rule(x,x,x,r(_1,_2,_3,_4,_5,_6,G),G) :-
    color(G).
```

natural/1: PROPERTY

Usage: `natural(X)`

Los numeros naturales son 0 o s(X) donde X es natural.

```
natural(0).
natural(s(X)) :-
    natural(X).
```

suma/3: PROPERTY

Usage: `suma(X,Y,Z)`

La suma de dos numeros naturales es un numero natural Z = X + Y.

```
suma(0,X,X) :-
    natural(X).
suma(s(Y),X,s(Z)) :-
    suma(Y,X,Z).
```

par/1: PROPERTY

Usage: `par(X)`

Los numeros pares son 0 o s(s(X)) donde X es un numero par.

```
par(0).
par(s(X)) :-
    \+par(X).
```

impar/1: PROPERTY

Usage: `impar(X)`

Los numeros impares son s(0) o s(s(X)) donde X es un numero impar.

```
impar(s(0)).
impar(s(X)) :-
    par(X).
```

longitud/2: PROPERTY

Usage: `longitud(X,Y)`

La longitud de una lista es 0 o s(N) donde N es la longitud de la lista sin el primer elemento.

```
longitud([],0).
longitud([_1|L],s(N)) :-
    longitud(L,N).
```

dif2/2: PROPERTY

Usage: `dif2(X,Y)`

La longitud de la lista X es la longitud de la lista Y mas 2.

```
dif2(L1,L2) :-
    L1\=[],
    L1\=[_1],
    L1\=[_2,_3],
    longitud(L1,N1),
    longitud(L2,N2),
    suma(N1,s(s(0)),N2).
```

termina/1: PROPERTY

Usage: `termina(X)`

La lista X acaba por una celula o.

```
termina([x|L]) :-
    termina(L).
termina([o|L]) :-
    termina(L).
termina([o]).
```

empieza_termina/1: PROPERTY

Usage: `empieza_termina(X)`

La lista X empieza y acaba por o.

```
empieza_termina([o|L]) :-
    termina(L).
empieza_termina([o]).
```

aplicar_regla/4: PROPERTY

Usage: `aplicar_regla(X,Y,Z,A)`

La lista A es el resultado de aplicar la regla Z a la lista Y, X es el elmento de la izquierda.

```
aplicar_regla(o,[o,N|INICIAL],REGLA,[o,Y|FINAL]) :-
    rule(o,o,N,REGLA,Y),
    aplicar_regla(o,[N|INICIAL],REGLA,FINAL).
aplicar_regla(P,[E,N|INICIAL],REGLA,[Y|FINAL]) :-
    rule(P,E,N,REGLA,Y),
    aplicar_regla(E,[N|INICIAL],REGLA,FINAL).
aplicar_regla(P,[o],REGLA,[Y,o]) :-
    rule(P,o,o,REGLA,Y).
```

cells/3: PREDICATE

Usage: `cells(X,Y,Z)`

La lista Z es el resultado de aplicar la regla Y a la lista X.

```
cells(INICIAL,REGLA,FINAL) :-
    aplicar_regla(o,INICIAL,REGLA,FINAL),
    dif2(INICIAL,FINAL),
    empieza_termina(FINAL).
```

evol/3: PREDICATE

Usage: `evol(X,Y,Z)`

La lista Z es el resultado de aplicar la regla Y a la lista [o,x,o] en el paso X.

```
evol(0,_1,[o,x,o]).
evol(s(N),REGLA,FINAL) :-
    evol(N,REGLA,INICIAL),
    cells(INICIAL,REGLA,FINAL).
```

steps/2: PREDICATE

Usage: `steps(X,Y)`

La lista X es el resultado de aplicar la regla Y a la lista X en el paso Y.

```
steps([o,x,o],0).
steps(FINAL,N) :-
    longitud(FINAL,L),
    impar(L),
    empieza_termina(FINAL),
    evol(N,_1,FINAL).
steps(Goal,_1) :-
    call(Goal).
```

ruleset/2: PREDICATE

Usage: `ruleset(X,Y)`

La regla X es la regla que se aplica a la lista Y.

```
ruleset(RuleSet,Cells) :-
    steps(Cells,N),
    evol(N,RuleSet,Cells).
```

0.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.