Pràctica de Prolog

Laura Galera Alfaro Aleix Fernández Salavedra

6 de juny de 2021

Índex

1 Descripció del problema

El problema a resoldre és la col·locació de tendes en una quadrícula seguint una sèrie de restriccions. En primer lloc sabem el nombre de tendes que hi ha d'haver a cada fila i

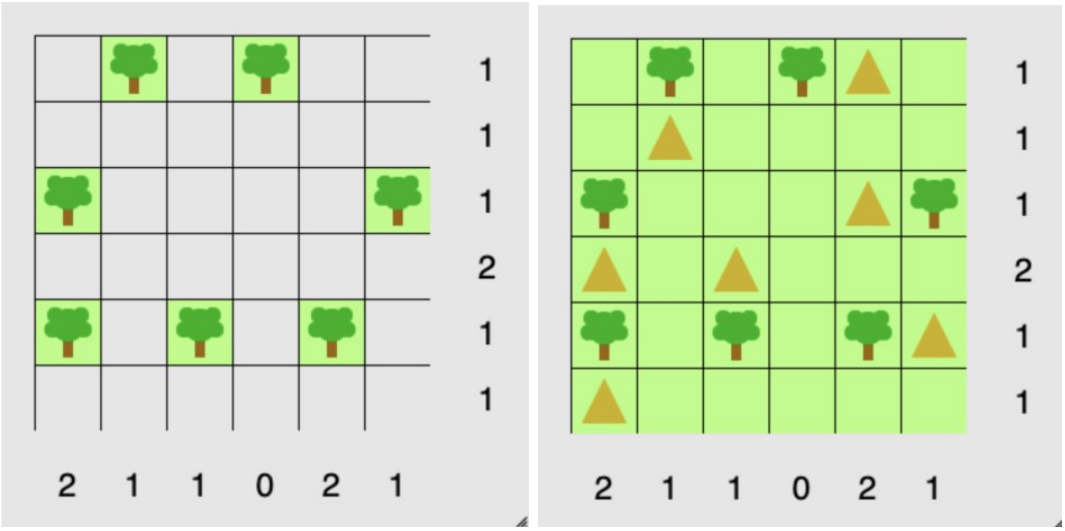

Figura 1: Exemple del problema amb una possible solució.

Figura 2: Exemple on el programa simplificat donaria correcte (esquerra). El problema original necessitaria una configuració com la de la dreta, en canvi.

a cada columna de la quadrícula. A més, la quadrícula conté arbres, i es vol que cada arbre tingui almenys una tenda al costat (sense comptar les diagonals). Finalment, les tendes no es poden tocar entre sí (en aquest cas incloent les diagonals). Es pot veure un exemple del problema i una possible solució a la figura 1.

En el problema original cada arbre ha de tenir la seva pròpia tenda, de manera que hi haurà sempre el mateix nombre de tendes que nombre d'arbres. Com que això complica la col·locació, s'ha simplificat el problema de forma que dos arbres podrien arribar a compartir una tenda, tal com es pot veure a la figura 2. Això generarà més solucions que el problema original, però les solucions del problema original també es trobaran. Podem estar segurs, doncs, que si el programa no ens troba solució, el problema original tampoc en té. Es podria donar el cas, però, en què el programa original no tingui solució i la versió simplificada sí. De tot això en veurem exemples més endavant.

Pel que fa al funcionament del programa, el dividim en 3 apartats, que corresponen també amb 3 predicats. En primer lloc veurem el predicat per decidir la satisfactibilitat de fórmules Booleanes (SAT). Seguidament veurem el codificador que ens passarà de la

descripció del problema de les tendes a una fórmula CNF. Finalment veurem el predicat que mostra la solució a partir del model obtingut amb el SAT solver. Així doncs, el predicat `resol` es basa en fer la conjunció de codificar, SAT solver i mostrar.

```
% resol(+N,+LA,+LF,+LC) => mostra la solució al problema de
↪  tendes amb quadricula NxN, LA les posicions dels arbres, LF
↪  les tendes per fila i LC les tendes per columna
resol(N,LA,LF,LC):-codifica(N,LA,LF,LC,CNF),sat(CNF,[],M),
                   mostra(N,LA,LF,LC,M).
```

2 SAT solver

Aquest predicat ens decideix la satisfactibilitat de fórmules Booleanes en CNF. Es basa en l'algorisme de *backtracking* de Davis Putnam Logemann Loveland (DPLL).

El seu funcionament a grans trets consisteix a escollir una clàusula unitària amb un sol literal —si no n'hi ha cap, s'escull el literal d'una clàusula qualsevol— i assignar aquest com a cert. Llavors es propaga actualitzant la CNF de la següent manera: (i) esborrant les clàusules que tenen aquest literal, (ii) esborrant el seu literal negat de totes les clàusules. Si en algun moment apareix la clàusula buida, cal fer *backtracking* i buscar una assignació diferent, ja que la interpretació actual fa la CNF insatisfactible. Si s'aconsegueix assignar totes les variables sense conflicte, llavors la interpretació és model de la fórmula.

El predicat `sat` és el següent:

```
% sat(F,I,M) => donada una fórmula F en CNF i una interpretació
↪  I, es satisfà quan M sigui un model de F construit extenent I
sat([],I,I):- write('SAT:'), write(I),nl,nl,!.
sat(CNF,I,M):-tria(CNF,L),simplif(L,CNF,CNFS),append(I,[L],IR),
              sat(CNFS,IR,M).
sat(CNF,I,M):-tria(CNF,L),N is -L,simplif(N,CNF,CNFS),
              append(I,[N],IR),sat(CNFS,IR,M).
```

El primer cas talla la recursivitat perquè la interpretació és model de la fórmula. Els altres dos casos serveixen per construir la interpretació de la CNF. Seguint l'algorisme, la interpretació sorgeix d'escollir un literal, el qual s'assigna a cert o fals, d'aquí els dos casos, i actualitzar la CNF amb l'assignació. A partir d'aquí es continua amb la construcció de la interpretació fins a arribar al model o deduir que cap interpretació és model de la CNF, en altres paraules, la CNF és insatisfactible.

La tria no té secret perquè els dos `append` actuen d'igual manera que el `member`. En el primer cas, Lit és el literal de la primera clàusula unitària de la CNF i el tall evita buscar altres models: quan trobem el literal, és aquest i prou. En el segon cas, el literal és el primer d'una clàusula qualsevol, sigui unitària o no, però no serà el d'una clàusula unitària pel tall d'abans i l'ordre del predicat. El tall de nou evita el backtracking per tractar altres literals de la CNF.

```
% tria(CNF,Lit) => Lit és el primer literal d'una clàsula
↪  unitària de CNF, un literal qualsevol de la CNF si no n'hi ha
↪  cap
tria(CNF,Lit):-append(_,[[Lit]|_],CNF),!.
tria(CNF,Lit):-append(_,[[Lit|_]|_],CNF),!.
```

El simplifica serveix per actualitzar la CNF eliminant les clàusules ja satisfetes mentre en quedin i no aparegui la clàusula buida.

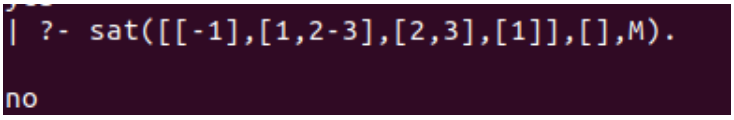

Figura 3: CNF insatisfactible.

```
% simplif(+Lit,CNF,CNFS) => CNFS és la CNF sense les clàusules
↪  amb el literal Lit i sense el literal -Lit
simplif(_,[],[]).
simplif(X,[Y|YS],REC):-append(_,[X|_],Y),!,simplif(X,YS,REC).
simplif(X,[[E]|_],_):-E is -X, !,fail.
simplif(X,[Y|YS],[Z|REC]):-append(L,[E|ES],Y),E is -X, !,
                           append(L,ES,Z), simplif(X,YS,REC).
simplif(X,[Y|YS],[Y|REC]):-simplif(X,YS,REC).
```

El primer predicat és una CNF buida, llavors la CNFS també és buida. Fixem-nos en el segon predicat, aquest simplement construeix la CNFS eliminant les clàusules on apareix el literal. El tercer té com a particularitat el *fail* i és que acabem de trobar que aquesta assignació fa la CNF insatisfactible perquè apareix una clàusula avaluada a fals que contamina tota la CNF. Aquí és on es fa el *backtracking* per cercar una assignació diferent de literals per la interpretació. En el següent, es porta a terme l'eliminació del literal negat de la primera clàusula i, per últim, es dóna el cas que la clàusula de la CNF no conté ni el literal ni el literal negat, per això aquesta també ha d'aparèixer a la nova CNF simplificada.

Si el provem amb algunes CNF, veiem resultats com els de les figures 3, 4, 5, en què els literals positius signifiquen que són certs i els negatius falsos. En cas de no aparèixer, significa que tant poden ser falsos com certs, no importa pel model.

3 Codificació

La codificació consisteix a traduir el problema a una CNF que contempli totes les restriccions que cal complir per a ser una solució d'aquest. D'aquesta manera resoldre el puzle

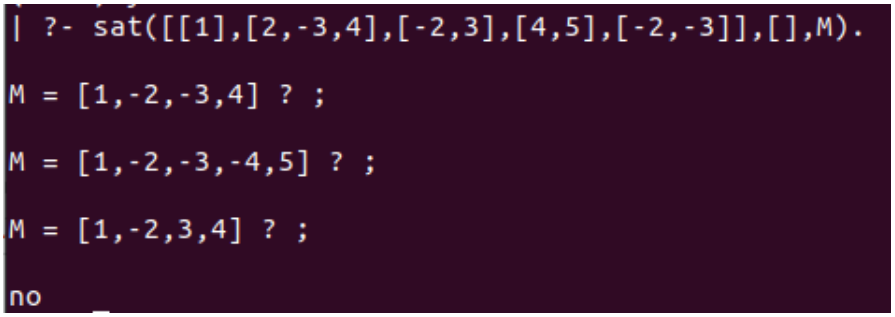

Figura 4: CNF satisfactible. Els models que es mostren depenen de l'ordre dels literals però sempre es mostren tots els possibles models. Passa que n'hi ha de més genèrics que d'altres. Per exemple, el model [1,-2,4] és un model que ja inclou [1,-2,-3,4] i [1,-2,3,4].

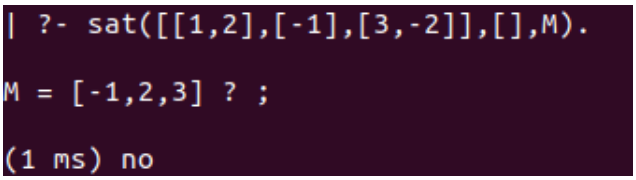

Figura 5: CNF satisfactible amb un únic model.

de tendes quedarà reduït a decidir si una CNF és satisfactible o no, essent el model trobat la solució al nostre joc. Les restriccions o regles ja s'han descrit a l'inici del document, així com la informació que es proporciona sobre el puzle.

La CNF del predicat `codifica` és la concatenació de les CNFs resultants per cada restricció del problema, a partir de la informació de les mides del tauler, les posicions on hi ha arbres i el nombre de tendes per fila i columna.

```
% codifica(+N,+LA,+LF,+LC,CNF) => CNF és la concatenació de CNFs
↪  resultants de cada restricció del problema per un puzle NxN,
↪  amb arbres a les posicions LA i tantes tendes requerides per
↪  fila i columna com marca LF i LC
codifica(N,LA,LF,LC,CNF):-generarmatriu(1,N,N,M),veins(M,N,LA,CNF1),
                          tendesfilacolumna(M,LF,LC,CNF2),
                          noveins(M,N,CNF3),noarbres(LA,M,CNF4),
                          append(CNF1,CNF2,L1),append(L1,CNF3,L2),
                          append(L2,CNF4,CNF).
```

Veiem a continuació cadascun d'aquests predicats.

3.1 Generar Matriu

Abans de procedir a veure la construcció de cada CNF, és necessari conèixer la relació entre el tauler i les variables. Per fer la reducció, cada casella de la matriu es representa amb una variable. Com que els literals són enters, cada casella serà representada per un nombre enter. Per fer-ho simple, l'enter 1 correspon a la primera casella i el valor es va incrementant una unitat en recorre la fila d'esquerra a dreta. Quan s'arriba al final de la fila, es continua amb la següent fila. Per tant, la casella $X_{n,n}$ té la variable NxN.

```
% generarmatriu(+F,+N1,+N2,M):- M és la matriu d'enters de mida
↪  N1xN2 amb la primera casella a F i la resta incrementada una
↪  unitat respecte la casella de l'esquerra
generarmatriu(_,0,_,[]):-!.
generarmatriu(F,I,N,[X|M]):- N2 is F+N-1, generarfila(F,N2,X),
                             F2 is N2+1, I2 is I-1,
                             generarmatriu(F2,I2,N,M).

% generarfila(+F,+N,M) :- M és la llista de números des de F fins
↪  N
generarfila(N,N,[N]):-!.
generarfila(F,N,[F|M]):-F < N, F2 is F+1, generarfila(F2,N,M).
```

Pel predicat `generarmatriu` es podria haver fet un pas d'immersió per incloure com a valor el comptador de files, però ho hem fet directament des del `codifica`. D'aquesta manera és més genèric perquè es pot demanar una matriu de NxN que comenci per un valor qualsevol, no cal que sigui 1. Així doncs, aquest va unificant amb resolució les files de la matriu fins que n'hi ha N. Sabent el primer element i l'últim de la fila, generar la llista de valors és només qüestió d'anar unificant-la amb el valor anterior incrementat una unitat.

3.2 Veïns

```
% veins(LLV,+N,+LA,CNF) => CNF es satisfà quan tots els arbres de
↪  LA tenen almenys una variable certa al costat
veins(_,_,[],[]).
veins(M,N,[A|LA],CNF):-veinselem(M,N,A,L),unCert(L,CNF1),
                        veins(M,N,LA,CNF2),append(CNF1,CNF2,CNF).
```

El predicat `veins` genera una CNF que se satisfà quan cada arbre del tauler té almenys una casella veïna certa, és a dir, almenys una variable del conjunt que representa les caselles adjacents[1] ha de ser certa. Primer és necessari saber quines són les variables veïnes coneixent la posició de l'arbre. Per fer-ho, tenim el predicat `veinselem` i d'aquesta manera disposar de la llista de variables veïnes. Amb això ja es pot construir la CNF descrita fa un moment i seguir fent el mateix amb la resta d'arbres. La CNF resultant és la concatenació de totes les CNFs.

[1] Ens referim a caselles adjacents en la línia vertical i horitzontal, diagonals no compten.

```
% veinselem(M,+N,+A,L) => L són les variables veines de l'arbre A
↪  de la matriu M de mida N
veinselem(M,N,(F,C),L):-F1 is F-1, F2 is F+1, C1 is C-1, C2 is C+1,
                        trobarelem((F1,C),M,N,E1),
                        trobarelem((F,C1),M,N,E2),
                        trobarelem((F,C2),M,N,E3),
                        trobarelem((F2,C),M,N,E4),
                        append(E1,E2,L1),append(L1,E3,L2),
                        append(L2,E4,L).

% trobarelem(+P,M,+N,E) => E és una llista amb l'element de la
↪  matriu M de mida N a la posició P. Si P està fora la llista E
↪  és buida
trobarelem((F,C),M,N,[E]):- F>0,C>0,F=<N,C=<N,!,
                            elem(M,F,E2),elem(E2,C,E).
                            trobarelem(_,_,_,[]).

% elem(L,+N,E) => E és l'element de L a la posició N
elem([X|_],1,X):-!.
elem([_|XS],Y,R):-Y1 is Y-1,elem(XS,Y1,R).
```

El predicat `veinselem` es basa en construir una llista amb els quatre elements que hi ha a les caselles de la dreta, de l'esquerra, damunt i de sota. Per controlar els elements fora de rang, hem definit el predicat `trobarelem`, que en comptes d'unificar un element, unifica una llista. Si la posició és dins el rang, el valor és la llista amb l'element, i si està fora de rang, és una llista buida.

```
% unCert(Xs,CNF) => CNF satisfà quan almenys 1 variable de Xs és
↪  certa
unCert(X,CNF):-atleastK(X,1,CNF).

% atleastK(Xs,+K,CNF) => CNF satisfactible quan K variables de Xs
↪  són certes
atleastK(Xs,K,CNF):- allargada(Xs,N), Np is N-K+1, Np > 0,
                     findall(C, comb(Np,Xs,C),CNF).

% comb(+N,X,Comb) => Comb es un subset del set X. L'ordre es
↪  irrellevant
comb(0,_,[]).
comb(N,[X|T],[X|Comb]):-N>0,N1 is N-1,comb(N1,T,Comb).
comb(N,[_|T],Comb):-N>0,comb(N,T,Comb).

% allargada(L,+N) => N es l'allargada de L.
allargada([],0).
allargada([_|Xs],N):-allargada(Xs,Np), N is Np+1.
```

El predicat `atleastK` consisteix en una CNF formada per la conjunció de totes les combinacions[2] de clàusules amb Z literals. Si volem que com a mínim K literals siguin certs, Z ha de ser el *total de literals - K + 1*. D'aquesta manera es força que com a mínim K literals siguin certs o hi hauria clàusules on no apareixeria cap d'aquests literals i serien falses. Un parell d'exemples a la figura 6.

És en aquest predicat on trobem la simplificació del problema original, tal com s'ha comentat a la descripció del problema. Mirant alguns exemples d'execució es pot veure (figura 7).

[2]El predicat *comb* és d'aquesta pàgina

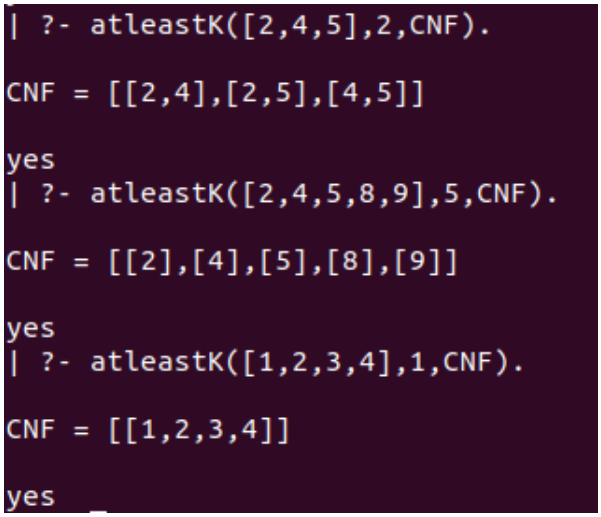

Figura 6: En el primer exemple, les combinacions són de dos elements. D'aquesta manera, si només un literal fos cert, alguna de les tres clàusules seria falsa i per tant força que 2 o més siguin certs per fer-la satisfactible. En el segon cas, tots els literals han de ser certs, per tant clàusules d'un sol literal. En l'últim, mínim un literal cert, llavors amb una sola clàusula és suficient per fer-la satisfactible.

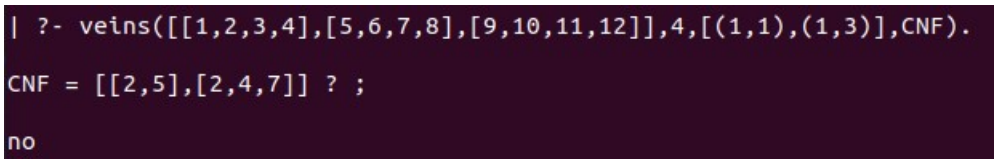

Figura 7: Exemple del predicat veïns per una matriu de mida 4. Podem veure que una sola tenda a la posició 2 és model de la CNF tot i tenir 2 arbres, cosa que en el problema original no passaria.

3.3 Tendes per files i columnes

```
% tendesfilacolumna(M,+LF,+LC,CNF) => CNF satisfactible quan cada
↪ fila de M té Xi literals certs i cada columna té Xj literals
↪ certs, essent Xi el valor i de LF, Xj el valor j de LC.
tendesfilacolumna(M,LF,LC,CNF):-tendesfila(M,LF,CNF1),transpose(M,T),
                                tendesfila(T,LC,CNF2),
                                append(CNF1,CNF2,CNF).
```

El predicat `tendesfilacolumna` genera una CNF que se satisfà quan hi ha exactament un cert nombre k de variables certes per cada fila i columna. S'utilitza el predicat `tendesfila` que genera la CNF de totes les files. En el cas de les columnes, es pot utilitzar el mateix predicat, però abans transposant[3] la matriu.

```
% tendesfila(M,+L,CNF) => CNF satisfactible quan cada fila de M
↪ té Xi literals certs, essent Xi el valor i de L
tendesfila([],[],[]).
tendesfila([F|M],[N|NS],CNF):-exactlyK(F,N,CNF1),tendesfila(M,NS,CNF2)
                              ,append(CNF1,CNF2,CNF).

% exactlyK(L,+K,CNF) => CNF satisfactible quan té exactament K
↪ literals de L certs
exactlyK(L,K,CNF):- atleastK(L,K,CNF1), atmostK(L,K,CNF2),
                    append(CNF1,CNF2,CNF).

% atmostK(Xs,+K,CNF) => CNF satisfactible quan té com a molt K
↪ literals de Xs certs
atmostK(Xs,K,CNF):- Np is K+1, Np > 0, findall(CN, (comb(Np,Xs,C),
                    negate(C,CN)), CNF).
```

[3] El predicat *transpose* és d'aquesta pàgina

```
| ?- atmostK([1,2,3,4],1,M).

M = [[-1,-2],[-1,-3],[-1,-4],[-2,-3],[-2,-4],[-3,-4]]

yes
| ?- atmostK([1,2,3,4],3,M).

M = [[-1,-2,-3,-4]]

yes
| ?- atmostK([1,2,3,4],0,M).

M = [[-1],[-2],[-3],[-4]]

yes
| ?- atmostK([1,2,3,4],4,M).

M = []
```

Figura 8: 4 exemples amb diferents K. Amb K = 0, la CNF és satisfactible quan tots els literals són falsos. A diferència de l'últim cas en què qualsevol interpretació satisfà la CNF.

```
| ?- tendesfilacolumna([[1,2],[3,4]],[1,1],[2,1],CNF).

CNF = [[1,2],[-1,-2],[3,4],[-3,-4],[1],[3],[2,4],[-2,-4]] ? ;

no
```

Figura 9: Aquí es veu un exemple d'una matriu 2x2. Les quatre primeres clàusules i les dues últimes representen els atleastk i atmostk amb 1. En el cas de K = 2 per la primera columna, només apareixen els casos de atleastk en què ambdós literals han de ser certs per força, però en el cas de atmostk, la CNF' és una llista buida perquè tota interpretació és model.

El predicat `tendesfila` construeix una CNF per cada fila i la resultant és la concatenació de totes elles. Per construir la CNF de la fila s'utilitza el predicat `exactlyK`, fent que K sigui el nombre X_i de L. Aquest predicat es pot fer reutilitzant el `atleastK` que ja hem vist i un nou predicat `atmostK`.

En el cas de `atmostK`, la CNF és satisfactible si com a molt K variables són certes. De nou es tracta de fer una conjunció de totes les combinacions d'un cert nombre de literals, però aquest cop de negats. Si tenim 4 literals, per exemple, i com a molt 2 poden ser certs, això significa que no podem tenir 3 que siguin certs (no tenir-ne quatre ja s'inclou en no tenir-ne tres), i evidentment cal fer totes les combinacions de 3 elements. En veiem exemples a la figura 8 i també un exemple complet a la figura 9.

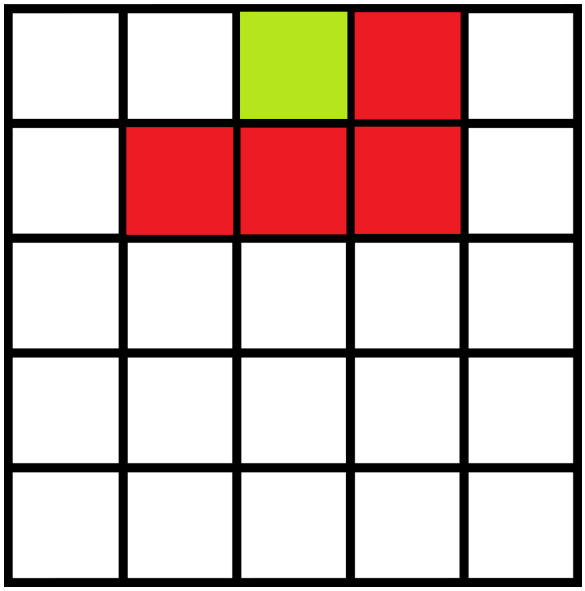

Figura 10: Si la casella en verd és certa, comprova que les files en vermell siguin falses.

3.4 No Veïns

```
% noveins(LLV,N,CNF) => CNF es satisfà quan a la matriu LLV de
↪  mida N no hi ha dues variables certes que es toquin
noveins([],_,[]).
noveins([X|XS],N,CNF):- tractarfila([X|XS],1,N,CNF1),
                        noveins(XS,N,CNF2), append(CNF1,CNF2,CNF).
```

El predicat `noveins` genera una CNF que se satisfà quan no hi ha dues variables veïnes certes que es toquin. Ho comprova fila per fila de forma recursiva, comprovant el predicat de `tractarfila`, que comprova la primera fila de la matriu.

El predicat `tractarfila` genera una CNF que se satisfà quan les variables certes de la primera fila de la matriu no tinguin altres variables certes al costat. Per fer-ho es crida de forma recursiva passant-li un índex. Necessita tota la matriu perquè també comprova que les variables certes de la 2a fila es toquin amb la de la primera. Per cada casella, doncs, mira la variable de la dreta i les 3 inferiors (figura 10).

```
% tractarfila(LLV,+Y,N,CNF) => CNF es satisfà quan les variables
↪  certes de la primera fila de LLV a partir de la posició Y no
↪  tenen cap variable certa al costat
tractarfila([_],N,N,[]):-!.
tractarfila([X],Y,N,[[E1,E2]|CNF]):-nelem(X,Y,E1),Y2 is Y+1,
                                    nelem(X,Y2,E2),
                                    tractarfila([X],Y2,N,CNF).
tractarfila([X,Y|XS],1,N,[[E1,E2],[E1,E3],[E1,E4]|CNF]):-!,
                                    nelem(X,1,E1), nelem(X,2,E2),
                                    nelem(Y,1,E3), nelem(Y,2,E4),
                                    tractarfila([X,Y|XS],2,N,CNF).
tractarfila([X,Y|_],N,N,[[E1,E2],[E1,E3]]):-!,nelem(X,N,E1),
                                    nelem(Y,N,E2), N2 is N-1,
                                    nelem(Y,N2,E3).
tractarfila([X,Y|XS],Z,N,[[E1,E2],[E1,E3],[E1,E4],[E1,E5]|CNF]):-
                                    nelem(X,Z,E1), Z2 is Z+1,
                                    nelem(X,Z2,E2), nelem(Y,Z,E3),
                                    nelem(Y,Z2,E4), Z3 is Z-1,
                                    nelem(Y,Z3,E5),
                                    tractarfila([X,Y|XS],Z2,N,CNF).
```

Tenim un predicat per cada cas, ja que quan s'arriba al final de la fila no s'han de comprovar els elements de la dreta, i per l'última fila no s'han de comprovar els elements de la fila inferior.

Si avaluem el predicat per una matriu de mida 3, per exemple, veiem com la CNF està formada de parelles d'elements que corresponen a les caselles que es toquen, i de cada parella un dels dos ha de ser fals (figura 11).

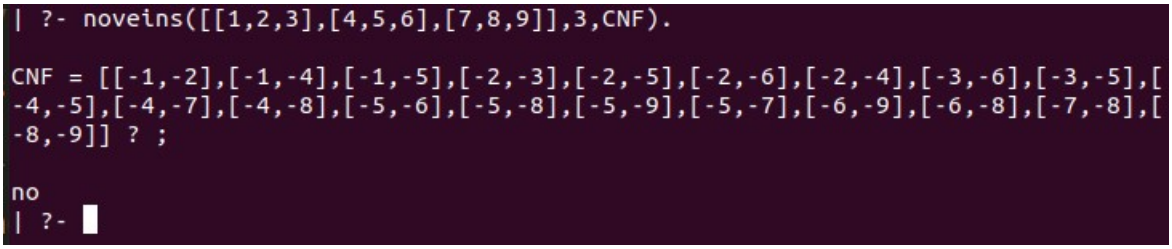

Figura 11: Noveins avaluat per una matriu de mida 3.

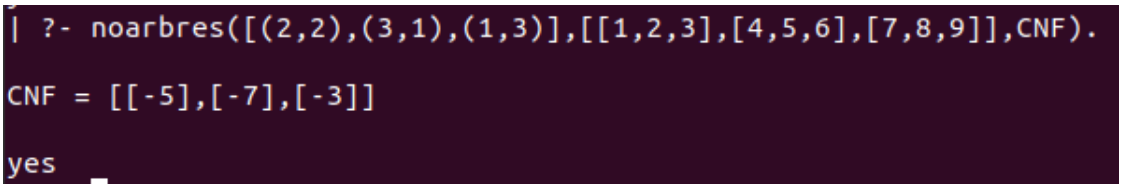

Figura 12: CNF amb arbres a les posicions (2,2),(3,1),(1,3) en una matriu 3x3

3.5 No arbre

```
% noarbres(+LA,M,CNF) => CNF es satisfà quan els literals de les
↪  posicions de LA de la matriu M són falsos
noarbres([],_,[]).
noarbres([(F,C)|L],M,[[X]|CNF]):-elem(M,F,E),nelem(E,C,X),
                                  noarbres(L,M,CNF).
```

Aquest predicat genera la CNF que se satisfà quan no hi ha cap tenda en una casella on hi ha un arbre. Per tant, el que fa és forçar que els literals corresponents a una casella amb arbre siguin falsos o, en altres paraules, que els seus negats siguin certs. Per fer-ho reutilitzem els predicats `elem` i `nelem`. El primer el fem servir per obtenir E, que és la fila de la matriu on hi ha l'arbre. Llavors, amb `nelem` obtenim l'element de la fila que correspon a la columna on hi ha l'arbre, i negat. Així doncs, la CNF està formada per clàusules unitàries de literals negats els quals fan referència a les caselles amb arbre (figura 12).

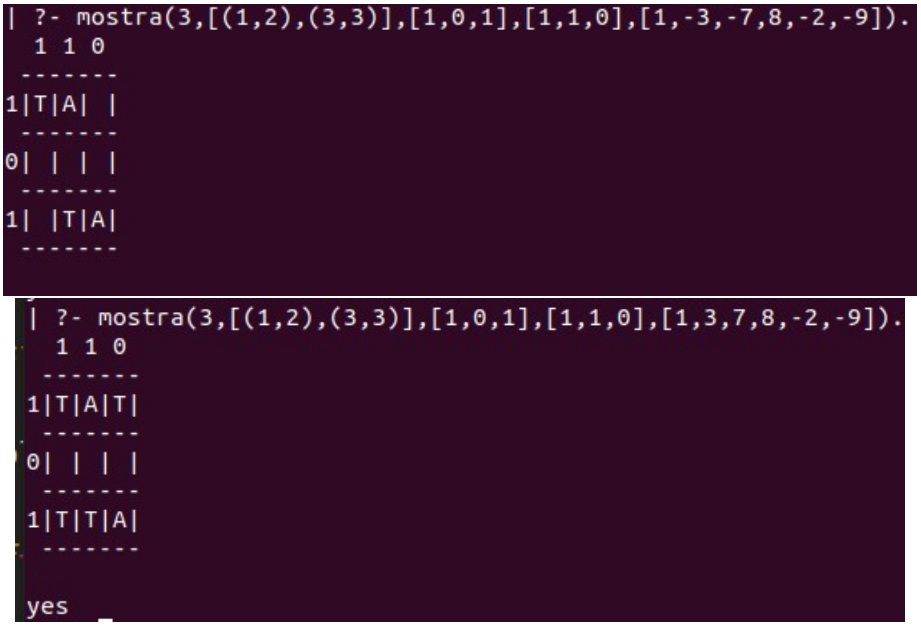

Figura 13: Exemples d'execució del mostra. El primer cas és coherent, però també podem mostrar quadrícules invàlides, tal com es veu en el segon cas, ja que el mostra no fa cap comprovació de res.

4 Mostrar

El predicat de mostrar escriu per pantalla la quadrícula amb la solució. Primer escriu la línia amb les tendes de cada columna, després una línia amb guions i, finalment, les files de la matriu. Se'n pot veure algun exemple a la figura 13.

Veiem ara el predicat de mostrar files:

```
% mostra_files(+F,+N,LA,LF,M) => mostra la fila F i les següent
↪  de la matiu de mida N segons els arbres LA i el model M
mostra_files(F,N,_,_,_):-F>N,!.
mostra_files(Fila,N,LA,[LF|LFS],M):-write(LF), write('|'),
↪  mostra_caselles(Fila,1,N,LA,M), write(' '), G is N*2+1,
↪  mostra_guions(G), FR is Fila+1,mostra_files(FR,N,LA,LFS,M).
```

En primer lloc escriu el nombre de tendes de la fila i després comprova `mostra_caselles`, que escriu les caselles de la fila. Finalment mostra els guions i fa una crida recursiva.

Finalment el mostrar caselles:

```
% mostra_caselles(+F,+C,+N,LA,M) => mostra les caselles de la
↪ fila F a partir de la C en una matriu de mida N segons els
↪ arbres LA i el model M
mostra_caselles(_,C,N,_,_):-C>N,!,nl.
mostra_caselles(F,C,N,LA,M):-append(_,[(F,C)|_],LA),!, write('A'),
↪ write('|'), CR is C+1, mostra_caselles(F,CR,N,LA,M).
mostra_caselles(F,C,N,LA,M):-P is (F-1)*N+C,
↪ append(_,[P|_],M),!,write('T'),write('|'),CR is
↪ C+1,mostra_caselles(F,CR,N,LA,M).
mostra_caselles(F,C,N,LA,M):-write(' |'),CR is
↪ C+1,mostra_caselles(F,CR,N,LA,M).
```

En cada cas mirem si la casella conté alguna cosa (arbres o tendes) i si no és així escrivim un espai en blanc. Finalment fem la crida recursiva per la resta de caselles de la fila.

5 Joc de Proves

Per començar provarem de resoldre el joc de la figura 1. Per aquest tauler només existeix una única solució (figura 14). De fet, aquesta coincideix amb la que hem vist a l'inici de l'informe, per tant no hi ha dubte que compleix totes les restriccions del problema. També hem decidit mostrar el model, en el qual veiem quins literals s'han assignat a cert i quins a fals. En total apareixen els 36 literals i el temps que ha trigat és de 11 ms.

Ara passem a intentar resoldre un tauler on és impossible trobar un model, ja que no es pot satisfer la CNF codificada. Prenent el mateix exemple d'abans, es modifica el valor de l'última columna perquè hi hagi dues tendes i no una. Com que no és possible fer quadrar el nombre de tendes amb les restriccions de files i columnes, no hi ha solució. El temps que ha trigat és de 10 ms (figura 15).

També és interessant veure si troba més d'un model. En aquest cas utilitzarem un tauler de 5x5 una mica més simple perquè únicament hi ha dos arbres. La idea és permetre

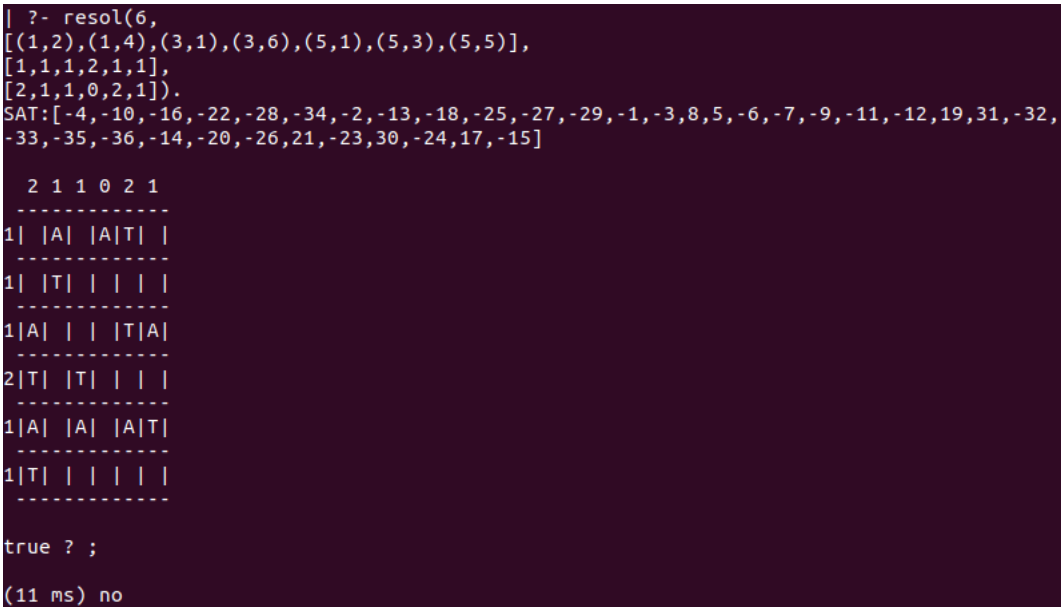

Figura 14: Exemples d'execució amb un tauler 6x6 i una única solució

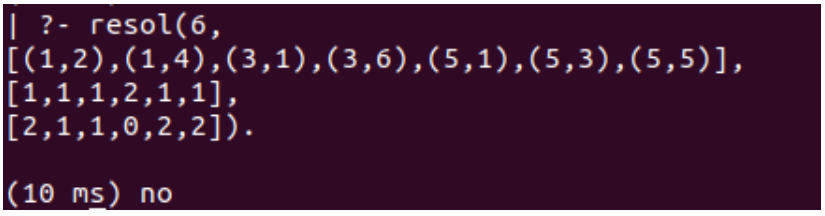

Figura 15: Exemples d'execució amb un tauler 6x6 i sense solució

que dues tendes es puguin intercanviar les columnes entre elles però, òbviament, respectant totes les restriccions. Amb `resol(5,[(4,1),(3,3)],[1,0,1,1,1],[2,0,1,0,1]).` existeixen 3 possibles solucions. Aquestes es poden veure a la figura 16. Totes tres les ha trobat en 8 ms.

Hem vist, doncs, que clarament no hi ha solucions amb tendes posicionades sobre els arbres i també es respecte el nombre de tendes per fila i columna. A la vegada, està prohibit que dues tendes es toquin. Si provem `resol(4,[(2,3),(1,2)],[2,0,1,1],[1,0,2,1]).`, ens diu que no hi ha solució perquè, si existís, llavors hi hauria dues tendes que es tocarien en diagonal, justament com es veu a la figura 17.

També podem demostrar que hi ha solucions amb arbres que tenen de costat més d'una tenda, de fet una solució en què un arbre està envoltat de tendes seria `resol(3,[(2,2)],[1,0,1],[0,2,0]).` (figura 18).

Ara volem posar a prova el temps que triga a trobar solucions amb taulers una mica més grans. Fins ara totes les proves s'havien fet amb taulers de mida 3-6. Passem a provar un amb mida 10x10, el qual sí que té solució. Ha trigat 192 ms (figura 19).

El millor és anar augmentant a poc a poc la mida perquè el temps es dispara. Si ho provem amb mida 12: `resol(12,[(1,6),(1,10),(2,1),(2,6),(2,12),(3,1),(3,4),(3,9),(4,2),(4,12),(5,7),(5,8),(6,1),(6,5),(6,11),(7,6),(7,8),(8,2),(8,4),(8,8),(8,11),(9,4),(9,9),(9,11),(10,3),(10,7),(11,5),(12,6)],[3,2,3,2,3,2,3,3,3,2,2,0],[3,2,2,2,3,2,3,2,2,2,2,3]).`, a més de trobar 4 solucions, ja que en el nostre cas no tenim en compte que cada tenda tingui assignada un arbre, ens triga 4769 ms, és a dir, uns 5 segons (figura 20).

Amb `resol(13,[(1,5),(1,8),(2,1),(2,11),(3,2),(3,5),(3,8),(3,10),(4,1),(4,2),(4,12),(5,3),(5,7),(5,9),(5,13),(6,2),(6,6),(6,11),(7,8),(7,12),(8,1),(8,3),(8,6),(8,11),(10,3),(10,6),(10,10),(10,12),(11,2),(11,9),(11,11),(11,13),(13,3)],[3,2,3,3,3,2,2,4,2,4,2,3,0],[4,2,2,4,0,5,1,2,3,2,3,3,2]).`, troba 6 solucions, una d'elles és exactament la mateixa de la web. També s'ha de dir que triga cosa de 25 segons, encara que també hi ha més models (figura 21).

Tanmateix, en provar-ho amb mida 15, aquest triga al voltant de 7 minuts i mig en trobar la primera solució. Així doncs, a partir de llavors el temps es dispara i resoldre un tauler

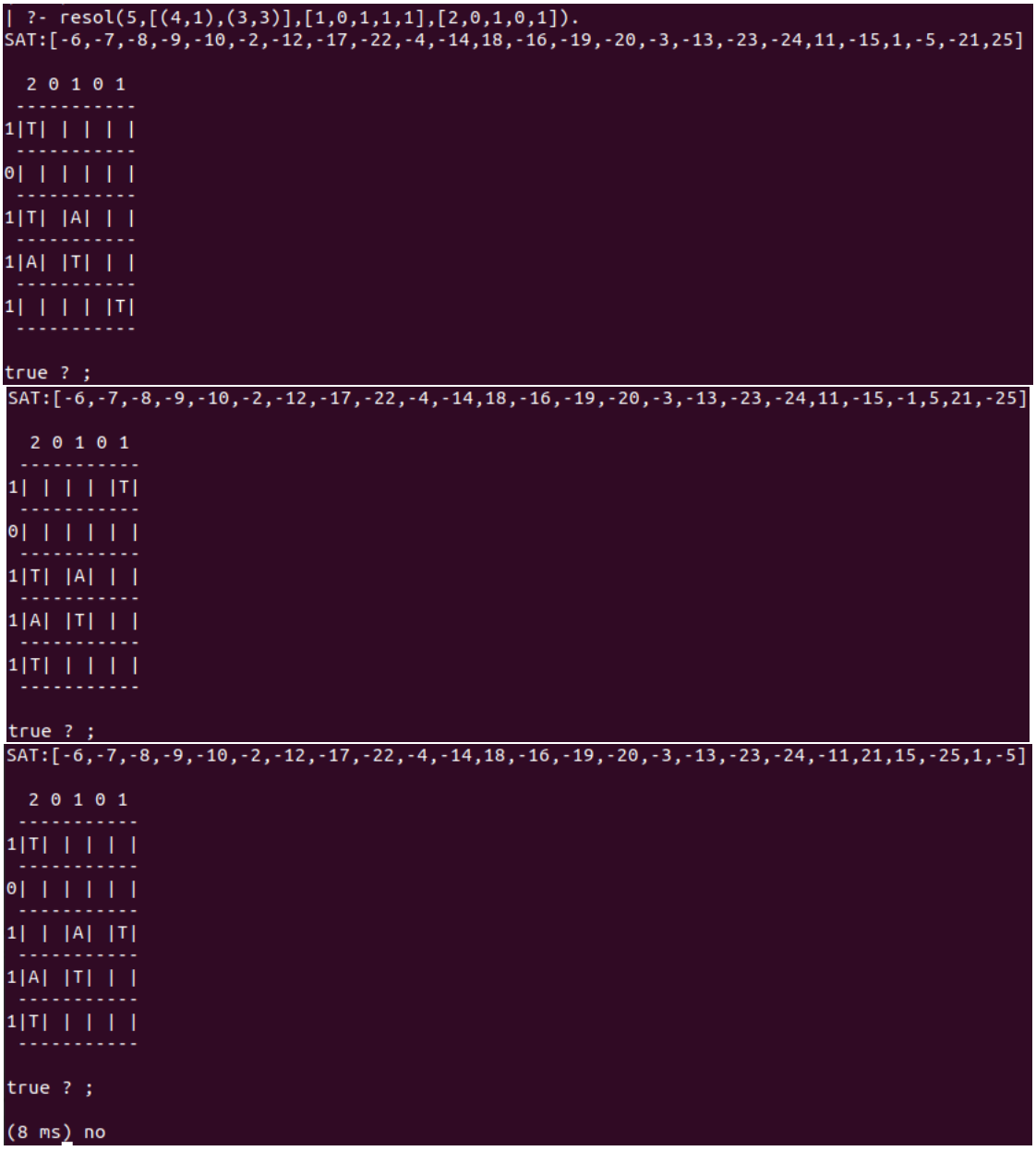

Figura 16: Exemples d'execució amb un tauler 5x5 i tres solucions

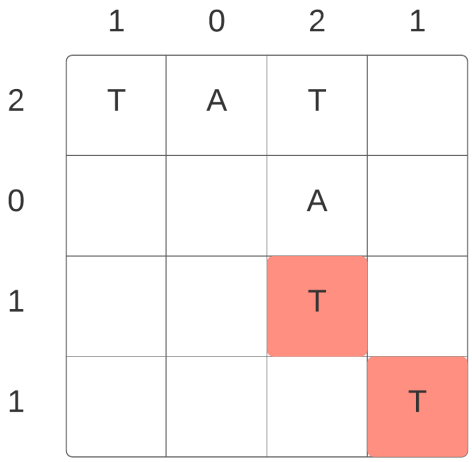

Figura 17: Interpretació incorrecta, no existeix cap model perquè les tendes es toquen.

```
| ?- resol(3,[(2,2)],[1,0,1],[0,2,0]).
SAT:[-4,-5,-6,-1,-7,2,-3,8,-9]

  0 2 0
 -------
1| |T| |
 -------
0| |A| |
 -------
1| |T| |
 -------

true ? ;

(1 ms) no
```

Figura 18: Exemples d'execució amb un tauler 3x3, on un arbre té 2 tendes.

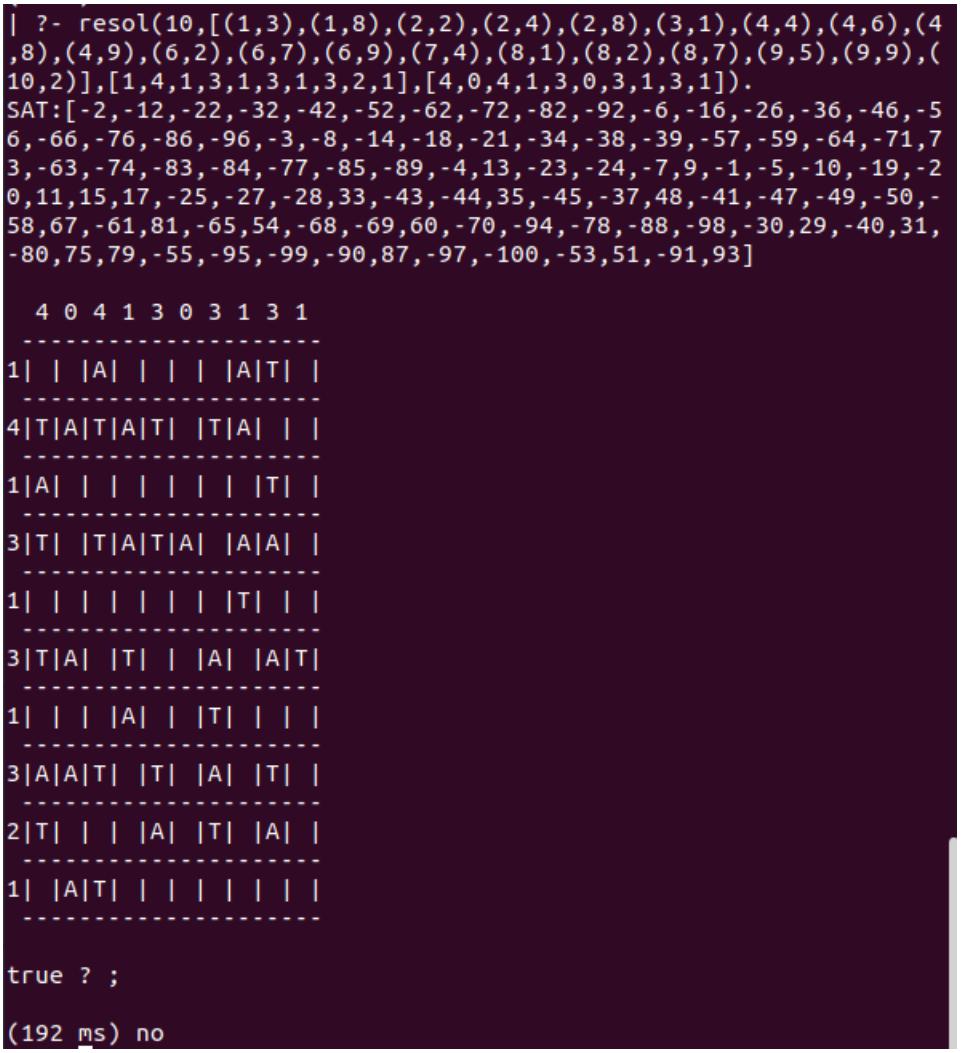

Figura 19: Exemples d'execució amb un tauler 10x10. Una solució.

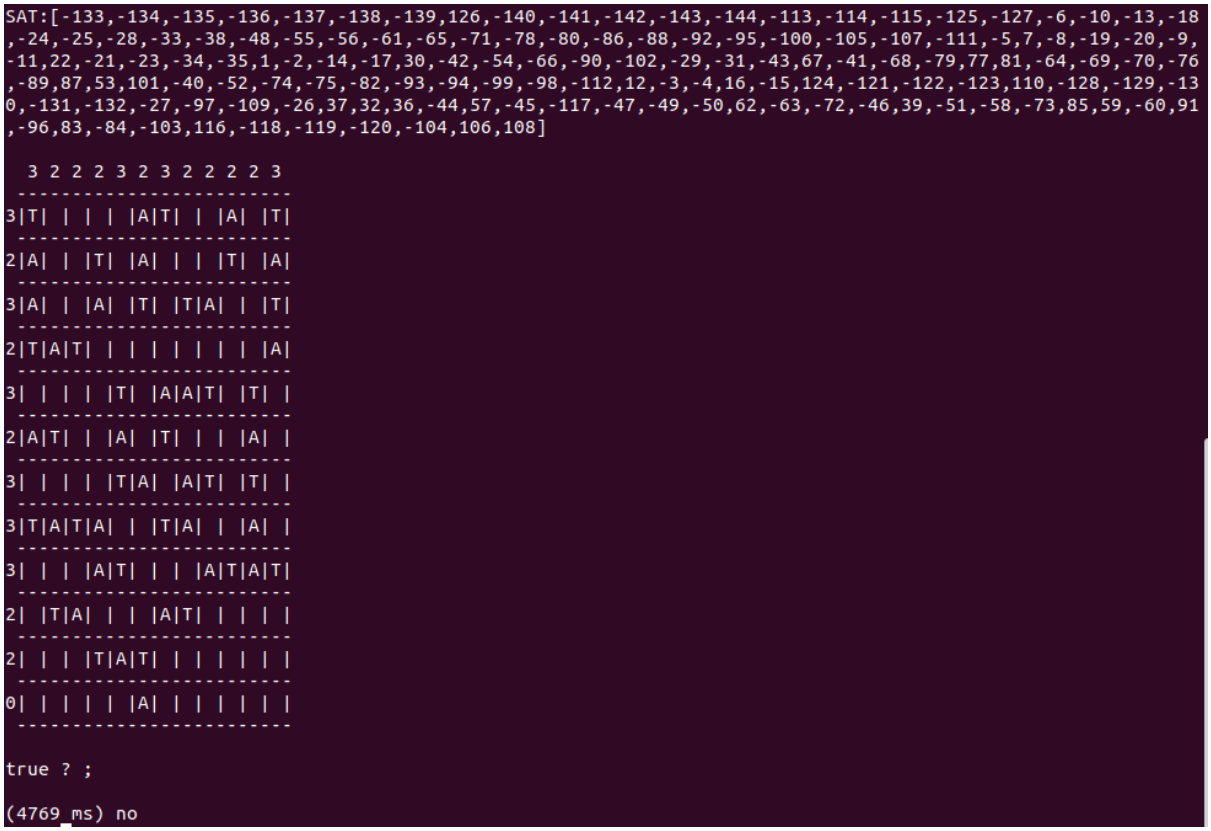

Figura 20: Exemples d'execució amb un tauler 12x12. Quatre solucions, encara que només mostrem una, en la que cada tenda té assignat el seu arbre.

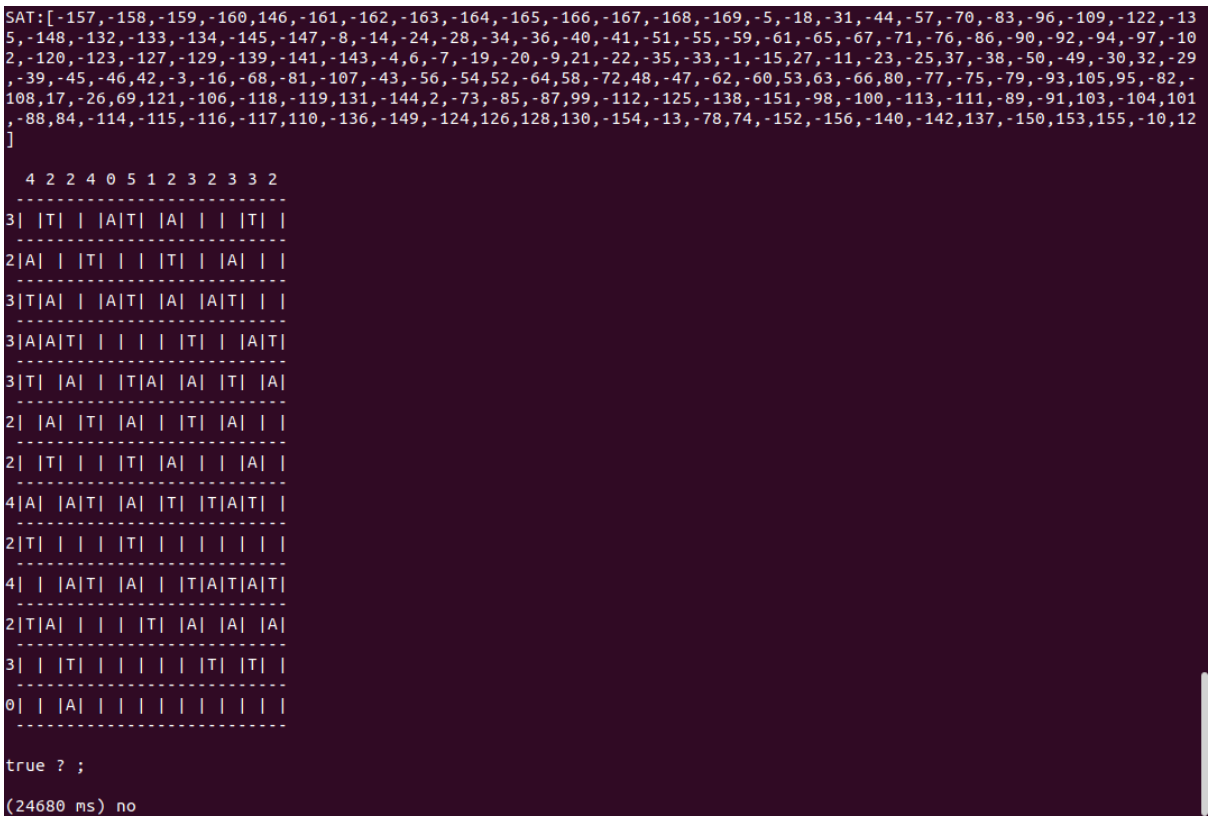

Figura 21: Exemples d'execució amb un tauler 13x13. 6 solucions, encara que només mostrem una.

de mida superior pot trigar molt. Recordem que no coneixem un algoritme de temps polinòmic que decideixi SAT. De fet, SAT és un problema NP-Complet.