# Project Report INF432- Tents and Trees

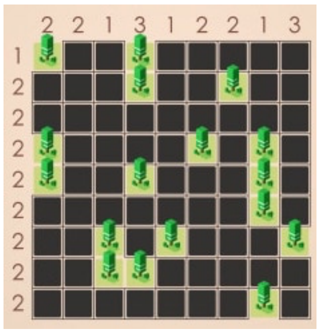
Tanguy Terrien Claire Louboutin Emeric Coudeville and Solene Hubert

2020

# Contents

## I Presentation of the game

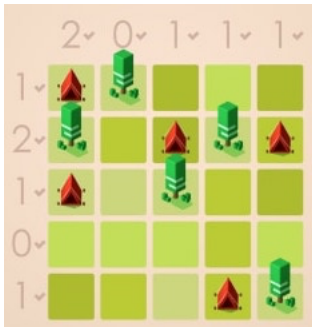
The game Tents and Trees can be presented as follows :
Trees are placed in a grid of size n x n. You have to place one tent next to each tree. The other cells are empty (grass).
The numbers around the grid indicate the number of tents that must be placed in the corresponding row or column :
On the figure, for example, we have a grid of size 9x9. Each dark cell must contain grass or a tent at the end, respecting the numbers on the left side and above.



This game includes 4 fundamental rules :

- A case contains either a tree, a tent or grass. (R1)
- Each tent must be in one of the four adjacent cells of its associated tree (horizontally or vertically but not diagonally). (R2)
- Tents cannot touch each other, not even diagonally. (R3)
- There must be exactly the right number (given) of tents in each row and each column. (R4)

A grid is succeed when each tree has an associated tent, no tent touch each other, and the number of tents in any row or column and the number corresponding to this row or column are the same, as we can see on the example.



# II Propositional logic

We formalize the rules as follows : With i,j belonging to [1,n], we write :
- T(i,j) : "There is a tent on the cell of row i and column j".
- G(i,j) : "there is grass on the cell i,j".
- Tr(i,j) : "there is a tree on the cell i,j."

Then, we can traduce the precedent rules : for all i,j belonging to [1,n] :

## Rule 1 :

$Tr(i,j) + T(i,j) + G(i,j)$.
To simplify, we will right t, tr and g, even though we have to keep in mind it is for a cell i,j.
...= tr + t + g . Also,
$Tr(i,j) => \overline{T(i,j) + G(i,j)}$.
$\equiv \overline{Tr(i,j)} + \overline{T(i,j) + G(i,j)}$
$\equiv \overline{Tr(i,j)} + \overline{T(i,j)}.\overline{G(i,j)}$
$\equiv (\overline{Tr(i,j)} + \overline{T(i,j)}).(\overline{Tr(i,j)} + \overline{G(i,j)}) \equiv (\overline{tr} + \overline{t}).(\overline{tr} + \overline{g})$.
Since the grass is not an important part of the problem (it is just a way to define empty cells), we can only keep $(\overline{tr} + \overline{t})$.

## Rule 2 :

$Tr(i,j) => T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1)$
$\equiv \overline{Tr(i,j)} + T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1)$.

## Rule 3 :

$T(i,j) \Rightarrow \prod \overline{T(x,y)}, x \in [i-1, i+1], y \in [j-1, j+1], (x,y) \neq (i,j)$. This is :
$T(i,j) => \overline{T(i-1,j-1)}.\overline{T(i,j-1)}.\overline{T(i+1,j-1)}.\overline{T(i,j-1)}$
$.\overline{T(i,j+1)}.\overline{T(i+1,j-1)}.\overline{T(i+1,j-1)}.\overline{T(i+1,j+1)}$
$\equiv \overline{T(i,j)} + \overline{T(i-1,j-1)}.\overline{T(i,j-1)}.\overline{T(i+1,j-1)}.\overline{T(i,j-1)}$
$.\overline{T(i,j+1)}.\overline{T(i+1,j-1)}.\overline{T(i+1,j-1)}.\overline{T(i+1,j+1)}$
$\equiv [\overline{T(i,j)} + \overline{T(i-1,j-1)}].[\overline{T(i,j)} + \overline{T(i,j-1)}].[\overline{T(i,j)} + \overline{T(i+1,j-1)}]$
$.[\overline{T(i,j)} + \overline{T(i-1,j)}].[\overline{T(i,j)} + \overline{T(i+1,j)}].[\overline{T(i,j)} + \overline{T(i-1,j+1)}]$
$.[\overline{T(i,j)} + \overline{T(i,j+1)}].[\overline{T(i,j)} + \overline{T(i+1,j+1)}]$

## Rule 4 :

We call Y the size of the grid. For each number X corresponding to the number of tents in the row or column, we have to compute the conjunctive normal form corresponding to "there are exactly X trees in Y cells". It it easy to formalize this into a disjunctive normal form. Let us call 1 the first cell, 2 the second cell, ...., y the y-th cell (of the row or column). For each set of numbers $S \subset \binom{x}{y}$, we call C the conjunction of all the numbers $N \subset S$ as positive variables and all the numbers $N' \subset [1, y] \setminus S$ as negatives variables. The disjunction of all these conjunctions is the final clause in disjunctive normal form.

**Example with $\binom{2}{3}$ :** It corresponds mathematically to : ((1,2) ; (1,3) ; (2,3))
Therefore, it can be transformed into 3 conjunctions : $(1.2.\overline{3}), (1.\overline{2}.3), (\overline{1}.2.3)$
We finally take the disjunction of all the conjunctions : $(1.2.\overline{3}) + (1.\overline{2}.3) + (\overline{1}.2.3)$

We have the clause in disjunctive normal form. The difficult part is to find the equivalent conjunctive normal form. We tried to compute conjunctive normal forms of some random cases to find a recurring pattern. We finally found one : for each set of numbers $S \subset \binom{x+1}{y}$, we call D the disjunction of all the numbers $N \subset S$ as negative variables. For each set of numbers $S' \subset \binom{y-x+1}{y}$, we call D' the disjunction of all the numbers $N' \subset [1, y] \setminus S'$ as positive variables. The conjunction of all these disjunctions is the final clause in conjunctive normal form.

**Example with $\binom{2}{3}$ :** We have x=2 and y=3. Therefore, x+1=3 and y-x+1 = 3-2+1 = 2.
$\binom{x+1}{y}$ corresponds to $\binom{3}{3}$ which is : (1,2,3).
Therefore, it can be transformed into a disjunction : $(\overline{1} + \overline{2} + \overline{3})$
$\binom{y-x+1}{y}$ corresponds to $\binom{2}{3}$ which is : ((1,2) ; (1,3) ; (2,3))
Therefore, it can be transformed into 3 disjunctions: $(1 + 2), (1 + 3), (2 + 3)$
We finally take the conjunction of all the disjunctions : $(\overline{1} + \overline{2} + \overline{3}).(1 + 2).(1 + 3).(2 + 3)$ This is the conjonctive normal form that we want.

## III File format

The format that is accepted by our program to produce clauses is defined as follows :
The file must contain : on a line, the size of the grid, which means the number of columns (and rows). On another line, the number of the total amount of trees in the grid, and then, on the 2 next lines, the list of number of tents by column, and the list of the number of tents by row. Finally we have the grid, with n rows and n columns, with a 1 when there is a tree, and a 0 otherwise.
For instance, let us take a grid of size 5x5 and containing 5 trees (so 5 tents at the end), with the number of tents by columns : 1, 2, 0, 1, 1 ; and the number of tents by row : 1, 0, 2, 0, 2. This grid, expressed in the right format, may give :

```
5
5
1 2 0 1 1
1 0 2 0 2
0 1 0 0 0
0 0 0 0 1
0 0 0 0 0
0 1 0 0 0
1 0 0 0 1
```

## IV Details of the program implementation

The different steps in coding are :

- Coding a program that guides the user to built a grid in our defined format, which is readable by our programs (those which will transform the file containing the grid into a set of clauses). The function EntreeEtVerif(file name) allows the user to create a grid which has a chance to be satisfiable (i.e. the file is correctly written). It returns a file, containing the grid and its information, that we then send to the main function. To create the grid, the user is asked to enter an integer for the size of the grid, the wanted number of trees (with conditions), the number of tents on each columns, the number of tents on each rows, and finally the grid itself, with the right size, puting 1 when there is a tree and 0 otherwise. The grid is displayed on the screen to show to the user the grid he created.

- Coding a function that verifies that the given file is correct, in the right format (isCorrect(file)). Hence, the user can send a file he created himself, or be guided to create a file with the program.

- Coding the part that receives a readable input file, creates a list of clauses (corresponding to the set of clauses) from the 4 rules that are defining the game, and returns a Dimacs file with these clauses, so it can be read by the Sat Solver :
  The main function calls for a function that recuperates the grid and extracts the important information :
  the size of the grid (n x n), the total number of trees in the grid, a list of the values for the columns (liste_abscisse), a list of values for the rows (liste_ordonnee), and a list of the values of the grid.
  Then, we create the clauses with creationClauses, and we get clauses from the four rules we have. This gives a list of clauses, listes_clauses.
  This list is then sent to the function creationDimacs, which takes the size of the grid, the list that represent this grid, our list of clauses, and the name of our file Dimacs. It writes the informations into this file, and finally, the main returns this Dimacs file.

- Coding (this was a facultative part) a Sat Solver, that determines if the list of clauses is satisfiable or not, and returns the solution (if it exists) into a file :
  Our function main takes in argument the name of a Dimacs file and the name of the file that will contain the result of the Sat solver (ResSat). It returns nothing but writes inside the second one the solution of the Sat Solver : that can either be the word "SAT"and a model satisfying the instance, or the word "UNSAT" if the instance is unsatisfiable.
  It gets back the information contained in the Dimacs file, (number of clauses, number of variables, list of clauses), and creates a random assignation. Then, while the assignation is not a model for the list of Clauses we have, we have to change this assignation. We first pick randomly a clause in the set of clauses that is False in the assignation. Then, we have to pick a variable in this clause which we will change in the assignation. This choice of variable depends on the probability and the heuristic we choose : if we take a probability of X in [0,100], we have X% chance for the variable to be chosen randomly, and (100-X)% chance for the variable to be chosen deterministically (see paragraph on Heuristics).
  Then, the value of the variable is switched in the assignation (False becomes True, True

becomes False), which means that the assignation has changed. If a model is found, (before we reach the maximum number of switches, arbitrarily defined as a billion), we write in ResSAt the header meaning that the problem is satisfiable or not. If it is, we write the model.

- Coding a program that transforms a clause into clauses of size 3, so it can be sent to a 3SAT-Solver.
  It reads the clauses contained in a Dimacs file and transforms each one into clauses of size 3, and writes them in a Dimacs3SAT file.

- Coding the program that links everything together. It receives the arguments given by the user when he executes the program.
  It calls for the library Entree.py if necessary, and then for the functions in Clauses.py to obtain the Dimacs file. If the argument <sat-To-3-Sat> is non-empty, the program transforms the dimacs file into a dimacs3Sat one.
  Then, if the user wants to use the SAT-Solver called minisat, the program executes minisat on the Dimacs file (classic or 3SAT). If the user wants to use the SAT-Solver called Walksat, he has to choose between 4 different heuristics the one he wants to use.
  We then use the Walksat program we created with the chosen heuristic. In any case, it returns the file ResSat, that we send to the function afficherResultat in Clauses.py, which writes the model of the instance of the problem found by the SAT-Solver in an understandable way.
  The execution for this program should be :
  ./main.py <problem-file> <final-result-file> <sat-solver> <sat-To-3-Sat> (see README.md)

## V Heuristics

We decided to code different heuristics for the deterministic choice of variables in the walksat.

**1st heuristic :** based on the heuristic MOMS. It first computes the number of occurrences of every literal in small clauses (we decided that a clause is “small” if it is of size 1 or 2). Then, for the given clause, chooses the literal with the most occurences in small clauses. Note that if two literals have the same number of occurrences in small clauses, the smallest value of abs(litteral) is chosen.

**2nd heuristic :** based on the heuristic JW. It first computes the “weight” of each litterals. At the beginning, all the weights of literals are initialised at 0. For each clause, if a literal is in the clause, we adds to his weight the length of the clause. Then, for a given clause, for each litteral in the clause, the program associates to the literal a random number in [0,weight of the litteral]. Finally, it chooses the lowest associated number (therefore, the possibility for a litteral to be selected by the heuristic JW is inversely proportional to the sum of the sizes of the clauses where it occurs.

**3rd heuristic :** based on the value of variables in the assignation. It simply creates a list containing all the literals in the given clause that are False in the assignation, and picks a random literal in this list.

**4th heuristic :** based on the number of switches of variables. At the beginning, all the counters of switches of literals are initialised at 0. For the given clause, it picks the one with the smallest number of switches. Note that if two literals have the same number of switches since the beginning of the execution, the smallest value of abs(litteral) is chosen. Then, the number of switches of the literal is updated (we add one to it).

## Comparison between classic clauses and clauses of size 3 for at same heuristic (see CompHeuristics for more details)

**Clauses and variables :** There is a big difference between clauses and 3-clauses : in 3-clauses, for a small grid (3x3), the number of clauses is multiplied by 2, and the number of variables by approximately 7. For a medium grid (6x6), the number of clauses doubles, and the number of variables is multiplied by 12.For a big grid (10x10), with clauses of size 3, the number of variables is multiplied by 20.

**Heuristic MOMS :** it makes no sense to study it with clauses of size 3, since it involves particularly the clauses of size 1 and 2.

**Heuristic JW :** We can see that there are approximately 6 times more iterations with 3-clauses (when P is high, variables are often randomly picked so the number of iterations only doubles). For a medium grid (6x6), the difference of number of switches made by walksat has exponentially grown : around 650 with classic clauses (whatever is P), and with 3-clauses, we have around 7500 iterations for P=10, and sixfold for P=90. For a big grid (10x10), walksat takes around 9000 iterations to fin a model. The number of switches usually exceeds a billion (for P=10, it once found a model in 802114 iterations, in a huge amount of time).

**Third heuristic :** For a little grid, the number of iterations are more or less the same whenever P=10, 50 or 90. The number of iterations are approximately multiplied by 2 when clauses are transformed into clauses of size 3. For a medium grid, the number of switches is around 700. With clauses of size 3, the result is surprising : between 130 000 and 200 000 iterations for P increasing. Finally, for a grid of size 10x10 and classic clauses, number of iterations are around 9 000 for any value of P, where for 3-clauses, the number of iterations exceeds a billion.

**Fourth heuristic :** For a little grid, the number of iterations are between 10 and 20 for classic clauses, and it increases when P increases. The number of iterations are approximately multiplied by 2 or 3 when clauses are transformed into clauses of size 3. For a medium grid, the number of switches is around 600, but with clauses of size 3, the number of switches sharply increases : more than 80 000 for a high value of P. Finally, for a grid of size 10x10 and classic clauses, the number of iterations is around 3 000. With 3-clauses, the number of iterations exceeds a billion for any value of P.

## Comparison between different heuristics

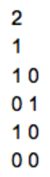
For a small grid, the overall best heuristic is the fourth one, based on the number of switches. The third one is better than the JW one, which is better than the MOMS. Results are the same for 3-clauses. For a medium grid, the best heuristic is again the fourth one, with a mean of around

600 iterations. Then, comes the heuristic based on the JW algorithm with around 650 iterations, and the third heuristic with around 700 iterations. Finally, using the MOMS algorithm, walksat takes approximately 800 iterations.

With clauses of size 3, rankings are significantly different. The best score is done by the JW heuristic 10 as value of P, which takes around 7500 iterations to find a model, against 35 000 for the fourth heuristic and 130 000 for the third heuristic. The higher P, the higher the number of switches (in a more or less quadratic way). For a big grid, the observations are proportionally the same than for a medium grid.

## VI Two examples of execution

Let follow the execution of our program on an example.
We take, first, a very simple grid, to detail the execution.



```
2
1
1 0
0 1
1 0
0 0
```

Figure 1: The first file we send to the program.

(It is then a grid 2x2 with 1 tree, and the tent is on the first column, second row).

Let us follows the creation of clauses for the first rule.
variables values : i = 0, j= 0
condition : grille_arbres[i,j] == '1'
action : dico = {}
dico[0,0] = False
content of liste_clauses : 0,0 : false
for all the other values for i and j, the condition 'grille_arbres[i,j] == '1" is not verified, so the list of clauses stops here. To write it in a clause, we take (with the function creationDimacs) the position numbers of the cell and not its coordinate (which is 0, because it is the first one) and add 1 (so we can make it negative if needed).
Then, as dico[0,0] = False, the 1 is transformed in -1. As it was the only value in the dico, it is the end of the clause.
Hence, the first clause of the Dimacs file should be -1 0.

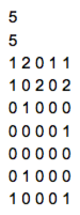
Indeed, the program returns the file :

```
p cnf 4 16
-1 0                   1 2 0
-1 -3 0                -1 -2 0
-2 -3 0                -3 0
-3 -4 0                -4 0
-1 -2 0                -1 0
-2 -3 0                -3 0
-2 -4 0                2 4 0
2 3 0                  -2 -4 0
```

Figure 2: The Dimacs file obtained after the execution of the program that creates clauses from the grid 2x2 previously saw (fig 1).

After sending it to walksat (or minisat, as the user wants), we obtain the following result :
SAT
-1 2 -3 -4 0
meaning the problem is satisfiable, and it gives a model for it.
This is traduced into an understandable solution : negative variables are converted in 0, positive ones in 1. It is given column by column, so the first number is for the cell 0,0, the second for the cell 0,1, ect.
So, we have ResFinal containing :
*beginning of the file*
The solution is :
    0 0
    1 0
Remember:
0: there is no tent
1: there is a tent
*end of the file*

We test it with another example, the same used in the format definition, but we do not detail as much as for the previous one.

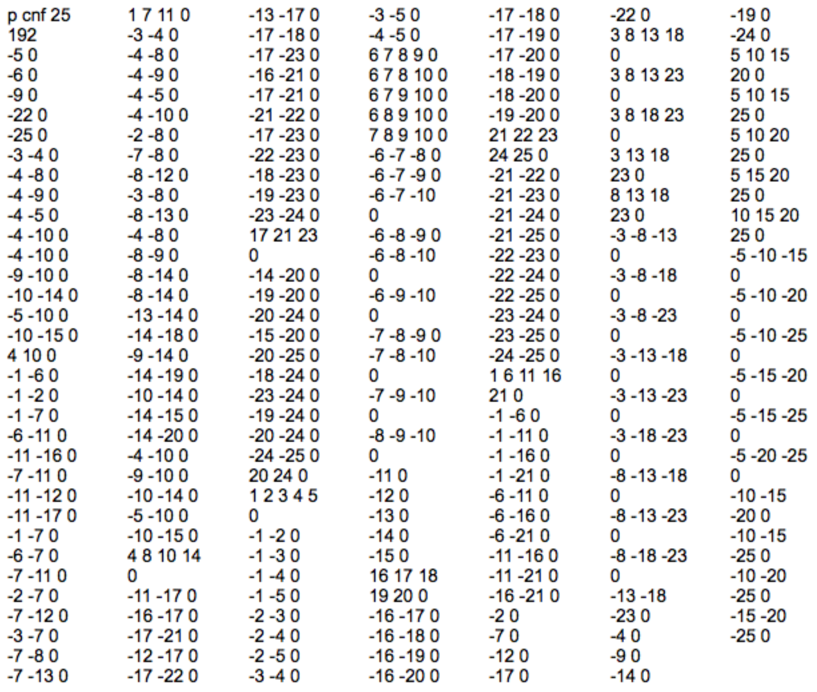


```
5
5
1 2 0 1 1
1 0 2 0 2
0 1 0 0 0
0 0 0 0 1
0 0 0 0 0
0 1 0 0 0
1 0 0 0 1
```

Figure 3: The second file we send to the program.

After sending it is the main program of Clauses.py, it returns the corresponding Dimacs file, which contains 25 variables and 192 clauses.

The zeros indicate the end of the clause:



```
p cnf 25    1 7 11 0    -13 -17 0   -3 -5 0     -17 -18 0   -22 0       -19 0
192         -3 -4 0     -17 -18 0   -4 -5 0     -17 -19 0   3 8 13 18   -24 0
-5 0        -4 -8 0     -17 -23 0   6 7 8 9 0   -17 -20 0   0           5 10 15
-6 0        -4 -9 0     -16 -21 0   6 7 8 10 0  -18 -19 0   3 8 13 23   20 0
-9 0        -4 -5 0     -17 -21 0   6 7 9 10 0  -18 -20 0   0           5 10 15
-22 0       -4 -10 0    -21 -22 0   6 8 9 10 0  -19 -20 0   3 8 18 23   25 0
-25 0       -2 -8 0     -17 -23 0   7 8 9 10 0  21 22 23    0           5 10 20
-3 -4 0     -7 -8 0     -22 -23 0   -6 -7 -8 0  24 25 0     3 13 18     25 0
-4 -8 0     -8 -12 0    -18 -23 0   -6 -7 -9 0  -21 -22 0   23 0        5 15 20
-4 -9 0     -3 -8 0     -19 -23 0   -6 -7 -10   -21 -23 0   8 13 18     25 0
-4 -5 0     -8 -13 0    -23 -24 0   0           -21 -24 0   23 0        10 15 20
-4 -10 0    -4 -8 0     17 21 23    -6 -8 -9 0  -21 -25 0   -3 -8 -13   25 0
-4 -10 0    -8 -9 0     0           -6 -8 -10   -22 -23 0   0           -5 -10 -15
-9 -10 0    -8 -14 0    -14 -20 0   0           -22 -24 0   -3 -8 -18   0
-10 -14 0   -8 -14 0    -19 -20 0   -6 -9 -10   -22 -25 0   0           -5 -10 -20
-5 -10 0    -13 -14 0   -20 -24 0   0           -23 -24 0   -3 -8 -23   0
-10 -15 0   -14 -18 0   -15 -20 0   -7 -8 -9 0  -23 -25 0   0           -5 -10 -25
4 10 0      -9 -14 0    -20 -25 0   -7 -8 -10   -24 -25 0   -3 -13 -18  0
-1 -6 0     -14 -19 0   -18 -24 0   0           1 6 11 16   0           -5 -15 -20
-1 -2 0     -10 -14 0   -23 -24 0   -7 -9 -10   21 0        -3 -13 -23  0
-1 -7 0     -14 -15 0   -19 -24 0   0           -1 -6 0     0           -5 -15 -25
-6 -11 0    -14 -20 0   -20 -24 0   -8 -9 -10   -1 -11 0    -3 -18 -23  0
-11 -16 0   -4 -10 0    -24 -25 0   0           -1 -16 0    0           -5 -20 -25
-7 -11 0    -9 -10 0    20 24 0     -11 0       -1 -21 0    -8 -13 -18  0
-11 -12 0   -10 -14 0   1 2 3 4 5   -12 0       -6 -11 0    0           -10 -15
-11 -17 0   -5 -10 0    0           -13 0       -6 -16 0    -8 -13 -23  -20 0
-1 -7 0     -10 -15 0   -1 -2 0     -14 0       -6 -21 0    0           -10 -15
-6 -7 0     4 8 10 14   -1 -3 0     -15 0       -11 -16 0   -8 -18 -23  -25 0
-7 -11 0    0           -1 -4 0     16 17 18    -11 -21 0   0           -10 -20
-2 -7 0     -11 -17 0   -1 -5 0     19 20 0     -16 -21 0   -13 -18     -25 0
-7 -12 0    -16 -17 0   -2 -3 0     -16 -17 0   -2 0        -23 0       -15 -20
-3 -7 0     -17 -21 0   -2 -4 0     -16 -18 0   -7 0        -4 0        -25 0
-7 -8 0     -12 -17 0   -2 -5 0     -16 -19 0   -12 0       -9 0
-7 -13 0    -17 -22 0   -3 -4 0     -16 -20 0   -17 0       -14 0
```

Figure 4: Dimacs file obtained after the execution of the program that creates clauses from the grid previously saw (fig 3).

Then, we send it to the SAT solver. It return the file "ResSat", containing the satisfiability of the problem and a model if possible :
SAT
1 -2 -3 -4 -5 -6 -7 8 -9 10 -11 -12 -13 -14 -15 -16 -17 -18 -19 20 -21 -22 23 -24 -25 0

Finally, the result is traduced, by the function afficherResultat(file1, file2), in a understandable way, and written in the file ResFinal :
*beginning of the file*
The solution is :

```
    1 0 0 0 0
    0 0 0 0 0
    0 1 0 0 1
    0 0 0 0 0
    0 1 0 1 0
```

Remember:
0: there is no tent
1: there is a tent
*end of the file*

Thus, at the end, the program created the previous grid.

We could also have obtain a Dimacs3SAT, which the first line is : p cnf 209 396. It is a bit long to put in a report.

## VII Test packs and results

| Test type | Caracteristic of the Input file | Output | Dimacs first line |
|---|---|---|---|
| Functionality, test of the limits | normal grid, correct declaration | The solution is :<br>0 0<br>1 0<br>Remember:<br>0: there is no tent<br>1: there is a tent | p cnf 4 16 |
| | number of trees : 0, with size 2 and 6 | The number of trees in the grid must be a strictly positive integer (not 0). | Ø |
| | Correct declaration , unsatisfiable | The grid is unsatisfiable or the maximal number of iteration has been reached. | Ø |
| Performance | size of the grid : 10x10 the grid is satisfiable | A solution is found | p cnf 100 1770 |
| | size of the grid : 9x9 the grid is satisfiable | A solution is found | p cnf 81 1340 |
| | size of the grid : 15x15 | ”the complexity is too high to execute it ”. | Ø |
| Robustness | With a grid of size 1, number of trees : 1 | IndexError: list index out of range | Ø |
| | With a grid of size 1, number of trees : 0 | The number of trees in the grid must be a strictly positive integer (not 0). | Ø |
| | An element is missing in the column values size of the grid : 3 | The line representing the number of trees of each column is incorrect (there must be 3 elements, not 2). | Ø |

| Test type | Caracteristic of the Input file | Output | Dimacs first line |
|---|---|---|---|
| Robustness | An element is missing in the grid values size of the grid : 3 | There is the wrong number of elements in the line 2 of the grid (there must be 3 elements, not 2). | ∅ |
| | An element is missing in the row values size of the grid : 3 | The line representing the number of trees of each row is incorrect (there must be 3 elements, not 2). | ∅ |
| | An empty number of trees | The number of trees in the grid is incorrect (it must be ONE strictly positive integer, not 0). | ∅ |
| | An empty size for the grid | The size of the grid is incorrect (it must be ONE strictly positive integer, not 0). | ∅ |
| | An empty file | The size of the grid is incorrect (it must be ONE strictly positive integer, not 0). | ∅ |
| | A grid line missing | There is the wrong number of elements in the line 3 of the grid (there must be 3 elements, not 0). | ∅ |
| | With many random white spaces | The solution is : 0 0 1 0 0 0 0 0 0 | p cnf 9 36 |
| | Multiple elements for the size | The size of the grid is incorrect (it must be ONE strictly positive integer, not 3). | ∅ |
| | Multiple elements for the number of trees | The size of the grid is incorrect (it must be ONE strictly positive integer, not 3). | ∅ |

| Test type | Caracteristic of the Input file | Output | Dimacs first line |
|---|---|---|---|
| Robustness | Non boolean values in the grid | Error at the line 1 of the grid. Every element of the grid should be a 0 or a 1. | Ø |
| | Real value for the size | The size of the grid is incorrect (it must be a strictly positive integer). | Ø |
| | Real value for the number of trees | The number of trees in the grid is incorrect (it must be a strictly positive integer). | Ø |
| | Real value in the column values | The line representing the number of trees of each column is incorrect (it must be only positive integers). | Ø |
| | Real value in the row values | The line representing the number of trees of each row is incorrect (it must be only positive integers). | Ø |
| | Real value in the grid | Error at the line 1 of the grid. Every element of the grid should be a 0 or a 1. | Ø |
| | Wrong number of trees tents : 1 trees : 2 | The grid is unsatisfiable or the maximal number of iteration has been reached. | |
| | Wrong number of trees tents : 2 trees : 1 | The sum of the number of trees in each column is different than the total number of trees. Therefore, the grid is unsatisfiable. | Ø |

## VIII Conclusions

**Conclusions for the tests :** After testing our programs with the different test packs, we can conclude that it is quite efficient.

For the functionality, the program works and find a model when it is satisfiable. It works on every correct file we send (not too big, see the robustness). We tested more grid than exposed in the previous section, but we do not really need to explain them, as it is the same principle each time and the grids have more or less the same characteristics.

For the performance, it begins to take more time to solve (between 5 and 10 seconds) the problem from a size 9x9 , which represent 81 cells and around 1300 clauses.For a size 10x10 (100 variables), it takes about 1800 clauses. The program is beginning to take quite a long time (about 20 seconds) with this size, and from size 15, the complexity is too high to execute it (even though we worked to improve the complexity). All of this take in account the dimacs file, and not the dimacs3sat, which would give way more variables... The created files with the 'guide program' are mostly unsatisfiable, as there is an extremely huge number of different issues with the same size (even low, 3 for exemple) ; and a low number of satisfiable grid among those (to give a number, we would have less than 1% satisfiable).

For the robustness, it resists to incorrect entries, so a file in the wrong format or with the wrong entries wouldn't be accepted.
In consequence, it verifies if the number of trees, the number of values in the columns and rows and in the grid are correct, and so it returns an error message if the format or the values are incorrect. It verifies the types of the values (integers, booleans, floats) as well.
In addition, the spaces are not taken in account, so a file with too much spaces is read as if there were not here, and so the program can still find a solution.

**Conclusion for the project :** Our programs work and they answer to the problem, which was to find solutions (when there is one) for the game Tents and Trees.

It is efficient (as much as possible) although the complexity is not perfect, and so the program is a little low for grid of size 9 and over. Moreover, it resists to incorrect inputs and explains what is wrong with the input quite precisely. It has been a really interesting project, as much from the coding point of view than from the logic one. It trained us to make relevant and complete test packs, to improve complexity and to translate logic formulas for the program to understand them.

In addition, the facultative parts on coding a sat solver (Walksat) and on the heuristics were very instructive, leading us to really understand the functioning of a Sat solver and the impact of the different methods used. We had even more to compare with the 3-Sat solver.

To conclude, it was a pleasant and enthusiastic project that required a lot of time but was very knowledgeable and constructive.