SAT SOLVER

Constraint Programming and Processing
MAI – UPC

November 2013

José A. Magaña Mesa

DNI:46703957L

Introduction

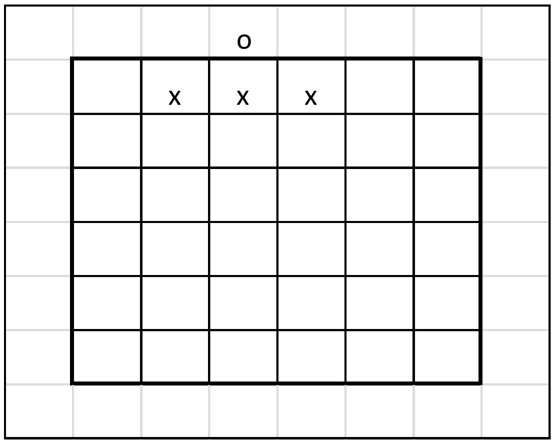
This report corresponds to the description of the solution implemented for the Conway's Game of Life SAT problem.

Conway's Game of Life is usually approached as a dynamic evolutionary system. In the exercise proposed we must find how given a certain system configuration, this configuration must be modified by adding checkers (not removing) in order to have an stable configuration, that is, the configuration does not change at the next iteration. Given that we are approaching the problem as a SAT problem, no distinction is made about the solutions obtained, regarding the number of checkers added or any other criteria that could be considered. Certainly, there is the possibility of introducing MAX-SAT constraints but this is beyond the scope defined for this assignment.

For a cell not to change its state, the conditions are the following:

- If the cell is alive, there must be two or three alive cells. If there are more than 3, the cell dies due to crowdedness. If there are less than 2, the cell dies due to loneliness.
- If the cell is dead, the cell will live in the next cycle if there are exactly 3 neighbors. Hence, a dead cell can have any number of alive cells, except 3.

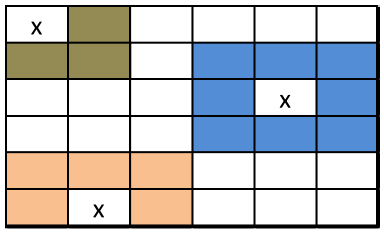
In our case, the board game is not considered cyclic (edges connected on the ends) but instead the board is considered part of a larger board whose stability must also be considered. This will, as the statement of the exercise, clearly indicates, require additional constraints be considered regarding groups of three consecutive cells in the cells in the borders of the board.



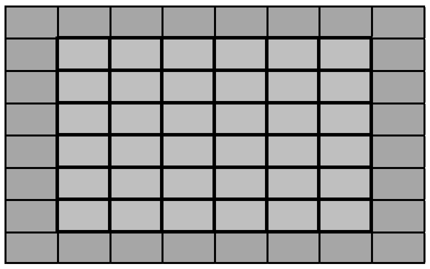
In the figure shown, if the 3 cells marked with an "x" are alive, in the next cycle, the cell marked with an "o" would be alive.

Solution strategy

When considering the stability of a cell, the 8 cells around it must be considered. Considering that our board game is bounded (finite and not cyclic) the neighborhood depends on its position in the board. As can be seen in the next figure, cells in the intern part of the board have 8 neighbors (blue), while cells in the border have 5 (orange) and cells in the corner just 3 (brown).

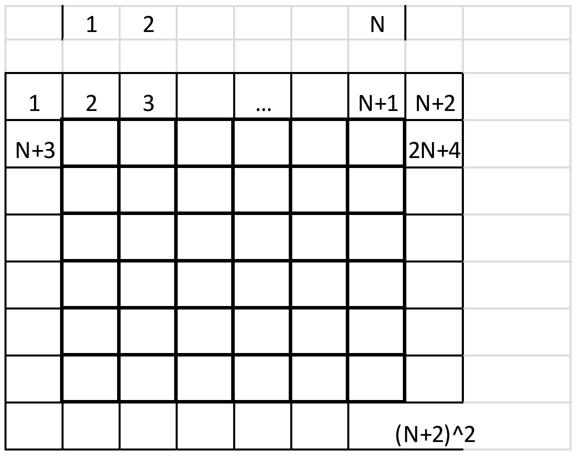


This complicates the definition of our rules so in order to simplify the code the board game is enlarged 1 unit in all dimensions, creating a margin that we will set as a dead region (no checkers) but that will allow us to apply our constraints in a uniform way that will also simplify the counting of auxiliary variables used in the implementation.



For a board with N side cells, we will have then an extended board with N+2 cells.

Considering our extended board, the numbering of the variables required to generate our CNF encoding will be as follows:



After that, any auxiliary variable will be added, starting from (N+2)^2+1.

In our implementation we will consider each cell in the extended board a variable, that will have value True(1) if it is Alive or False(0) if it is Dead.

In the implementation, there will be a (N+2) by (N+2) Boolean matrix and a second matrix containing for each cell its variable index. This is helpful to avoid having to convert from the (row, column) to the linear numbering 9 times (the number of neighbors plus one) as it is the number of times each cell will be involved in calculating constraints for the rest of the cells and its own constraints.

Given the function number of alive neighbors:

$$N(x) = \sum_{k=1}^{8} x_k$$

where x_k corresponds to the k-th neighbor of x, and being the order completely irrelevant, for the cell constraints we have the following conditions.

If the cell is Alive the number of alive neighbors must be 2 or 3. In this case, we are sure that the cell will be alive as the "rules" do not allow removing checkers to reach a stable configuration. For an alive cell, the constraint can be expressed as:

$$2 \leq N(x) \leq 3$$

and we can assert "x is alive" in our constraints.

If cell is dead the number of alive neighbors cannot be 3:

$$N(x) \neq 3$$

If this had to be implemented using cardinality constraints we would translate that as:

$$N(x) \leq 2 \;\; or \;\; 4 \geq N(x)$$

But if the cell becomes a live cell then its number of alive neighbors must be 2 or 3 so we need to consider that the value of the cell for dead cells can change. Then, for dead cells, our constraints are:

$$Alive(x) \rightarrow \; 2 \leq N(x) \; and \; N(x) \leq 3$$

$$Dead(x) \rightarrow \; N(x) \leq 2 \;\; or \;\; 4 \geq N(x)$$

For the implementation of the cardinality constraints several strategies/implementations are possible. In this case a sorting network has been used.

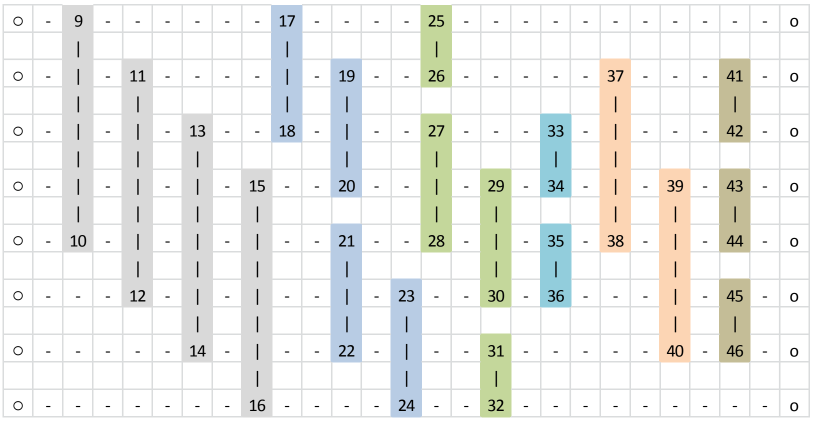
Sorting network implementation

In this case the sorting network implementation used is based on the Batcher’s Merge-Exchange (http://jgamble.ripco.net/cgi-bin/nw.cgi?inputs=8&algorithm=batcher&output=text) whose diagram for N=8 is shown below.

```
o--^--------^-----^-----------------o
   |        |     |
o--|--^-----|--^--v--------^--^-----o
   |  |     |  |           |  |
o--|--|--^--v--|--^-----^--|--v-----o
   |  |  |     |  |     |  |
o--|--|--|--^--v--|--^--v--|--^--^--o
   |  |  |  |     |  |     |  |  |
o--v--|--|--|--^--v--|--^--v--|--v--o
      |  |  |  |     |  |     |
o-----v--|--|--|--^--v--v-----|--^--o
         |  |  |  |           |  |
o--------v--|--v--|--^--------v--v--o
            |     |  |
o-----------v-----v--v--------------o
```

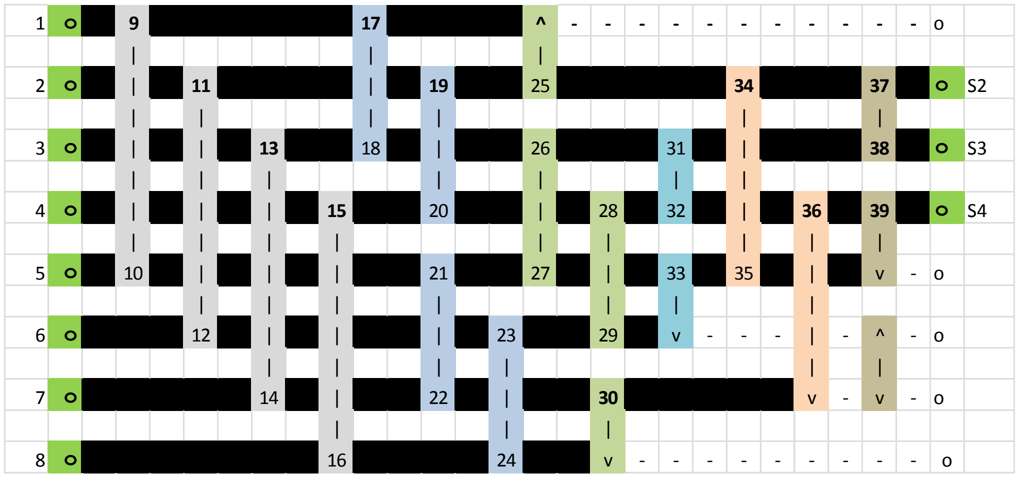
```
[[0,4],[1,5],[2,6],[3,7]]
[[0,2],[1,3],[4,6],[5,7]]
[[2,4],[3,5],[0,1],[6,7]]
[[2,3],[4,5]]
[[1,4],[3,6]]
[[1,2],[3,4],[5,6]]
```

This implementation has 19 MAX/MIN gates, organized in 6 levels. For each output we define a variable that will represent it, obtaining the following equivalent diagram where different colors indicate the gates are in different levels of the cascade operation. The outputs are numbered starting from 9, so that the numbering scheme does not overlap the inputs (numbered from 1 to 8).



For our scenario, we are not interested in the whole network to be sorted; only 3 of the results of the sorting network are of our interest, the 2nd, 3rd and 4th top outputs that correspond respectively to, at least 2, at least 3 and at least 4 alive cells in the input. The inputs represent the neighborhood set for a cell.

Starting from these outputs (2, 3 and 4) and propagating recursively the inputs required to calculate them we obtain a reduced Sorting Network like the one shown in the diagram that uses 7 variables less, a total of 31.



Replacing the variables in the sorting network definition we have this definition, that defines the relevant part of the sorting network and that allows to easily generate the required auxiliary variables only. Variables with value -1 indicate they are not relevant to our implementation:

```
[[[9,10],[11,12],[13,14],[15,16]],      [[0,4],[1,5],[2,6],[3,7]]
 [[17,18],[19,20],[21,22],[23,24]],     [[0,2],[1,3],[4,6],[5,7]]
 [[-1,25],[26,27],[28,29],[30,-1]],     [[2,4],[3,5],[0,1],[6,7]]
 [[31,32],[33,-1]],                     [[2,3],[4,5]]
 [[34,35],[36,-1]],                     [[1,4],[3,6]]
 [[37,38],[39,-1],[-1,-1]]]             [[1,2],[3,4],[5,6]]
```

Having side by side the original definition and our new numbered definition we can easily define how to generate the auxiliary variables in an automatic way.

For a pair of auxiliary variables [x, x+1] or [x,-1] we find the values in the same position in the original scheme. i.e. for [26,27] that is [3,5].

Then we find the original values in the previous level. i.e. for [3,5] the values are 20 and 23, 20 because is the one that matches 3 and 23 because is the one that matches 5, in the previous level in both cases.

We have that [26, 27] is generated from 20 and 23. The first variable corresponds to the OR (MAX) and the second to the AND (MIN). With that, applying the Tseitin method we can obtain easily the definition for each auxiliary variable:

$$26 \leftrightarrow 20 \ or \ 23$$

$$27 \leftrightarrow 20 \ and \ 23$$

For the auxiliary variables in the first level of our diagram, the input variables are directly the values on the same position. i.e. For [11,12] is [1,5].

If a certain value is not on a level, we keep looking for it in the previous. By construction, we know that we will find it. i.e. [34, 35] corresponds to [1, 4], 1 that does not appear in the prior level, but two levels down it does, corresponding to variable 29. On the other side, 4 correspond to 33, on the previous level.

Expanding the auxiliary variable obtained for our first example for the first(MAX/OR) clause, to obtain CNF clauses:

Not 26 or 20 or 23

Not 20 or 26

Not 23 or 26

And for the second(MIN/AND) clause:

Not 27 or 20

Not 27 or 23

Not 20 or not 23 or 27

Following this technique we can generate the whole set of clauses and replicate (applying the proper index offset for each of the cells) the whole CNF definition for the cardinality constraints.

For a cell in the position i, j (both indexes 1-based) and knowing that we have 31 auxiliary variables for each cell (in the board, not the extended board), the offset for the first auxiliary variable is calculated like:

Offset (i, j) = ((j - 1) * N + (i - 1)) * 31 + (N+2)^2 – 8

Where:

-8: is due to the fact that our first auxiliary variable is numbered 9 to avoid the conflict with the inputs.

(N+2)^2: is the number of cells in the extended board, the size of the board, and the index of the last (non-auxiliary) variable.

((j – 1) * N + (i – 1)) * 31: is the number of auxiliary variables used by the previous cells.

Note that only for the cells in the original board (not the extended board with the margin) the cardinality constraints are added.

If we generate a truth table to translate our original constraints to the outputs of the sorting network we obtain the following results:

	Neighbors				Functions	
	1	2	3	4	2 or 3 Neighbors	Not 3 Neighbors
S2	0	1	1	1	S2 AND NOT S4	NOT S3 OR S4
S3	0	0	1	1		
S4	0	0	0	1		

Also, translating the outputs to the auxiliary variables we obtain the following equivalences:

S2 ≡ 37

S3 ≡ 38

S4 ≡ 39

Wrapping up, we obtain the following expressions for our constraints.

For alive cells, x, two clauses are generated (numbers in brackets):

(1: the cell is alive) x

(2: there are at least two and not four or more) 37 and not 39

For dead cells, x, three clauses are generated. If the cell stays dead:

not x → (not 38 or 39)

(1: the cell is alive or not 2 neighbors or at least four) x or not 38 or 39

If the cell gets alive:

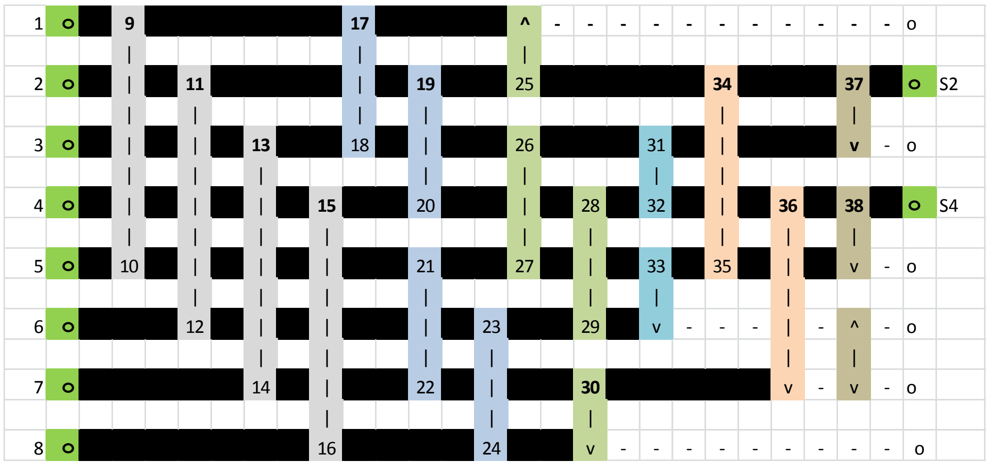
x → (37 and not 39)

Not x or (37 and not 39)

(2: the cell is dead or there are 2 neighbors) not x or 37

(3: the cell is dead or there are not 4 neighbors) not x or not 39

In the case of alive cells, we could further optimize the network by not using the output 3 that does not appear on the translated version. This way we would save one auxiliary variable but would complicate (slightly) keeping track of the auxiliary variables corresponding to each cell. So, this optimization has not been used in the implementation, since it doesn't provide any benefit. The simplified figure is shown in the next diagram:



In fact, these clauses referring to S3 for Alive Cells will be detected by the SAT Solver as Pure Literal Rules and won't be considered for resolution. The reason is that the variable will appear only once not being possible then to apply resolution to them.

Number of variables and clauses

For a configuration with NxN cells, the total number of variables will be:

N Vars = (N+2)^2 + 31 * N^2

N Clauses = 31 * N ^ 2 * 3 + → Each auxiliary variable requires 3 clauses

+ 4 * N + 4 + → The board margin is set to Dead (0) and there are 4N+4 cells on it

+ 4 * (N-2) + → The sides of the board cannot have 3 consecutive alive cells, corners excluded

+ N^2 * 3 → Each cell adds 3 clauses to express its constraints either alive or dead

The longest clause has size 3 and there are as many unary clauses as alive cells in the configuration.

Results

To show that the system is indeed working some captures of the results for some input configurations is shown. The first row corresponds to the input and the second to the output. The first pattern corresponds to a stable pattern; see that the input is identical to the output.

For the two modified configurations in the first row, the checkers removed to make the configuration unstable are highlighted. On the second row, '1' have been replaced by '*' when the checker was not present in the input. Differences from the original configuration have been highlighted to ease verification of the solutions (that are correct).

SL_15(ORIGINAL)	SL15_1	SL15_2
------------------------------	------------------------------	------------------------------
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 0 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	1 1 1 1 1 1 0 1 1 1 1 1 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 1 0 0 1 0 0 1 0 1	1 0 0 1 0 0 1 0 0 1 0 0 1 0 1	1 0 0 1 0 0 1 0 0 1 0 0 1 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 0 1 0 0 1 0 0 1 0 0 0 1 0	1 1 0 0 0 0 1 0 0 1 0 0 0 1 0	1 1 0 1 0 0 1 0 0 1 0 0 0 1 0
1 0 0 1 1 1 1 1 1 1 0 0 1 0 0	1 0 0 1 1 1 1 1 1 1 0 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 1 1 1 0 1	0 1 1 0 0 0 0 0 0 0 1 1 1 0 1	0 1 1 0 0 0 0 0 0 0 1 1 1 0 1
0 0 1 0 1 1 1 1 1 1 0 0 0 1 1	0 0 1 0 1 1 1 1 1 1 0 0 0 1 1	0 0 1 0 1 1 1 1 1 1 0 0 0 1 1
1 0 1 0 1 0 0 0 0 0 0 1 0 0 0	1 0 1 0 1 0 0 0 0 0 0 1 0 0 0	1 0 1 0 1 0 0 0 0 0 0 1 0 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1	1 1 0 1 0 0 1 0 1 1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1 0 0 0 0 0 1	0 0 0 1 0 1 1 0 1 0 0 0 0 0 1	0 0 0 1 0 1 1 0 1 0 0 0 0 0 1
1 1 0 1 0 1 0 0 1 0 1 1 0 1 0	1 1 0 1 0 1 0 0 1 0 1 1 0 1 0	1 1 0 1 0 1 0 0 1 0 1 1 0 1 0
1 1 0 1 1 0 0 1 1 0 1 1 0 1 1	1 1 0 1 0 0 0 1 1 0 1 1 0 1 1	1 1 0 1 1 0 0 1 1 0 1 1 0 1 1
------------------------------	------------------------------	------------------------------
------------------------------	------------------------------	------------------------------
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 * 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1 0 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	1 1 1 1 1 1 * 1 1 1 1 1 1 1 0	* * * * * * * 0 * * * 0 * * 0
1 0 0 1 0 0 1 0 0 1 0 0 1 0 1	1 0 0 1 0 0 1 0 0 1 0 0 1 0 1	1 0 0 1 0 0 1 * 0 1 0 * 1 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 0 1 0 0 1 0 0 1 0 0 0 1 0	1 1 0 * 0 0 1 0 0 1 0 0 0 1 0	1 1 0 1 0 0 1 * * 1 * * * 1 0
1 0 0 1 1 1 1 1 1 1 0 0 1 0 0	1 0 0 1 1 1 1 1 1 1 0 0 1 0 0	* 0 0 * * * * 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 1 1 1 0 1	0 1 1 0 0 0 0 0 0 0 1 1 1 0 1	0 1 1 0 0 0 0 0 * 0 1 1 1 0 1
0 0 1 0 1 1 1 1 1 1 0 0 0 1 1	0 0 1 0 1 1 1 1 1 1 0 0 0 1 1	0 0 1 0 1 1 1 1 1 1 0 0 0 1 1
1 0 1 0 1 0 0 0 0 0 0 1 0 0 0	1 0 1 0 1 0 0 0 0 0 0 1 0 0 0	1 0 1 0 1 0 0 0 0 0 0 1 0 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1	1 1 0 1 0 0 1 0 1 1 1 1 1 * 1	* * 0 * 0 0 * 0 * * * * * * *
0 0 0 1 0 1 1 0 1 0 0 0 0 0 1	0 0 0 1 0 1 1 0 1 0 0 0 0 0 1	0 0 0 1 0 1 1 0 1 0 0 0 0 0 1
1 1 0 1 0 1 0 0 1 0 1 1 0 1 0	1 1 0 1 0 1 0 0 1 0 1 1 0 1 0	1 1 0 1 0 1 0 0 1 0 1 1 0 1 0
1 1 0 1 1 0 0 1 1 0 1 1 0 1 1	1 1 0 1 * 0 0 1 1 0 1 1 0 1 1	1 1 0 1 1 0 0 1 1 0 1 1 0 1 1
------------------------------	------------------------------	------------------------------

For each of the previous configurations the following outputs are obtained:

1. SL15: since the configuration is already stable, there are no degrees of freedom and the given variables and clauses when propagated get solved by themselves. A so called "original" configuration with 0 elements (variables, clauses, literals) is obtained. Only propagation is applied.

```
This is MiniSat 2.0 beta
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  7264                                                 |
|  Number of clauses:    21716                                                |
|  Parsing time:         0.00         s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|         0 |       0        0        0 |        0        0    nan |  0.000 % |
===============================================================================
Verified 0 original clauses.
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 1              (0.00 % random) (67 /sec)
propagations          : 7264           (484267 /sec)
conflict literals     : 0              ( nan % deleted)
CPU time              : 0.015 s
```

2. SL15_1: Despite having a few checkers removed and the configuration not being stable, only by simple propagation the problem can be solved. As in the previous configuration the "original" problem has 0 clauses.

```
$ bash sat.sh sl15_1
This is MiniSat 2.0 beta
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  7264                                                 |
|  Number of clauses:    21716                                                |
|  Parsing time:         0.00         s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|         0 |       0        0        0 |        0        0    nan |  0.000 % |
===============================================================================
Verified 0 original clauses.
restarts              : 1
conflicts             : 0              (nan /sec)
decisions             : 1              (0.00 % random) (inf /sec)
propagations          : 7264           (inf /sec)
conflict literals     : 0              ( nan % deleted)
CPU time              : 0 s
```

3. SL15_2: A greater number of checkers are removed to create the input, in many cases in blocks. In this case, after propagating the evidences, the original problem still has a significant number of clauses where a real search must be made (decisions: 49, conflicts: 28)

```
$ bash sat.sh sl15_2
This is MiniSat 2.0 beta
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  7264                                                 |
|  Number of clauses:    21716                                                |
|  Parsing time:         0.00         s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|         0 |    2518     6474    15070 |     2158        0    nan |  0.000 % |
===============================================================================
Verified 6474 original clauses.
restarts              : 1
conflicts             : 28             (1867 /sec)
decisions             : 49             (0.00 % random) (3267 /sec)
propagations          : 14418          (961200 /sec)
conflict literals     : 106            (44.79 % deleted)
CPU time              : 0.015 s
```

	Minisat	Minisat2
sl15 (stable)	94.44Mb 0.031 sec 1 decision 7264 propagations	0.015 sec 1 decision 7264 propagations
Sl15_1	94.44Mb 0.031 sec 1 decision	0 sec 1 decision 7264 propagations
Sl15_2	18 conflicts 42 decisions 13540 propagations 38 conflict literals 95.44Mb 0.015 sec	28 conflicts 49 decisions 14418 propagations 106 conflict literals 0.015 sec

Other solvers

Lingeling (http://fmv.jku.at/lingeling/)

It´s a SAT solver created by researchers at the Johannes Kepler University in Linz, Austria, the same group that has also implemented PICOSat. PICOSat has not been included in the evaluation because it has not been possible to compile it.

The version used is AQW that won 7 medals in the 2013 SAT competition.

This solver is highly parametric but the documentation is not very clear and the parameters look very specific to advanced techniques implemented in the engine whose impact on the problem we are evaluating I have not been able to observe.

The results have been compared with the ones obtained for minisat.

One clear advantage of Lingelint is that it uses much less memory (1.3Mb vs 94.44Mb) what is an strong point if it is to be used in environments with very limited resources or where huge problems are to be processed.

A new parameter, agility, is given as part of the report that related to the restart ratios (according to the documentation)

	Minisat	Lingelint
sl15 (stable)	94.44Mb 0.031 sec 1 decision 7264 propagations	15914 irredundant clauses 7 agility 0 decisions 7264 propagations 1.3 Mb 0.0 sec
Sl15_1	94.44Mb 0.031 sec 1 decision	16195 irredundant clauses 7 agility 1.3 Mb
Sl15_2	18 conflicts 42 decisions 13540 propagations 38 conflict literals 95.44Mb 0.015 sec	17983 irredundant clauses 5 agility 28 conflicts 84 decisions 6452 propagations 1.3 Mb

UBCSAT

It’s a SAT Solver from the University of British Columbia. It uses methods based on local search methods. It provides the possibility of measuring the quality of the solution.

In our case, it considers the best solution the one with more alive cells (true variables).

For our complex test case (SL15_2), we obtain executing with 10 repetitions and for different algorithms, the following results:

WALKSAT-TABU

```
#        F  Best          Step       Total
#    Run N Sol'n            of      Search
#    No. D Found          Best       Steps
#
       1 0    58         84389      100000
       2 0    49         71568      100000
       3 0    61         55880      100000
       4 0    55         63900      100000
       5 0    69         77067      100000
       6 0    65         83051      100000
       7 0    44         92544      100000
       8 0    65         88370      100000
       9 0    68         95058      100000
      10 0    53         96450      100000


Variables = 7264
Clauses = 21716
TotalLiterals = 50316
TotalCPUTimeElapsed = 0.187
FlipsPerSecond = 5347593
RunsExecuted = 10
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUTime_Mean = 0.018700003624
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.018700003624
```

GSAT

```
#        F  Best          Step       Total
#    Run N Sol'n            of      Search
#    No. D Found          Best       Steps
#
       1 0   107         92898      100000
       2 0   121         40002      100000
       3 0   115         36709      100000
       4 0   100         79496      100000
       5 0   127         82606      100000
       6 0   105         97988      100000
       7 0   131         80266      100000
       8 0   126         69341      100000
       9 0   127         66185      100000
      10 0   104         70182      100000


Variables = 7264
Clauses = 21716
TotalLiterals = 50316
TotalCPUTimeElapsed = 0.349
FlipsPerSecond = 2865330
RunsExecuted = 10
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUTime_Mean = 0.0348999977112
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.0348999977112
```

GWSAT

```
#        F  Best          Step       Total
#    Run N Sol'n            of      Search
#    No. D Found          Best       Steps
#
       1 0   109         86980      100000
       2 0   115         91989      100000
       3 0   124         74271      100000
       4 0   113         64659      100000
       5 0   118         74468      100000
       6 0   116         99217      100000
       7 0   104         88081      100000
       8 0   118         96266      100000
       9 0    92         78273      100000
      10 0   130         95775      100000

Variables = 7264
Clauses = 21716
TotalLiterals = 50316
TotalCPUTimeElapsed = 4.879
FlipsPerSecond = 204960
RunsExecuted = 10
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUTime_Mean = 0.48789999485
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.48789999485
```

WalkSAT-TABU is the faster solution but also the one that gets to the worsts solutions, although in our case it is not relevant.

RSAT

RSAT is a SAT solver implemented at the UCLA university (http://reasoning.cs.ucla.edu/rsat/). Its algorithm promises the use of a phase selection heuristic oriented toward reducing work repetition and a frequent restart policy.

None of the parameters available allow modifying the behavior of the engine:

```
Usage: rsat <cnf-file-name> [options]
Solve the SAT problem specified in <cnf-file-name>.
Example: rsat sat-problem.cnf
RSat 2.01 options:

 -q             quiet. Do not print out the answer line. Suppress -s.
 -s             solution. Print out solution if one is found.
 -t <timeout>   time-out. Stop and return UNKNOWN after <timeout> seconds.
 -v             verbose. Print out useful information during execution.

Example:
        ./rsat problem.cnf -s -t 100 -v
Report bugs to <rsat@cs.ucla.edu>.
```

The output gives information about number of decisions, conflicts and learning steps. The CNF stats clauses do not refer to the total number of clauses.

```
$ ./rsat_2.01_win.exe sl15_2cnf.txt -v
c Rsat version 2.01
c +----+-----------------+-----------------+---------------------------------+---------------------------+---------+-----------
c | Re | Conflicts       | Original        | Learned                         | Decisions                 | Time    | KB
c | st |      Max  Actual | Clauses Literals |      Max Clauses Literals     LPC |   Total     Per Sec    C/D |         | Red. Sim.
c +----+-----------------+-----------------+---------------------------------+---------------------------+---------+-----------
c |  0 |      512      34 |    6474    15311 |    2158      21       109     5.2 |      68   22666.67  0.500 |   0.003 |    0    4
c +----+-----------------+-----------------+---------------------------------+---------------------------+---------+-----------
+
c
c
s SATISFIABLE
c CNF stats: (7264 vars, 17983 clauses)
c Decisions: 68
c Conflicts: 34
c Running time: 0.02500 seconds
```

The solution generated as output (not shown here) does not return the variables in order, probably the order depends on the order the variables have been assigned a value.

MARCH_RW

This SAT solver is provided by the Delft University of Technology (http://www.st.ewi.tudelft.nl/sat/index.php).

It belongs to a family of SAT solvers also highly awarded in the SAT competition. The advantage of this SAT solver seems to be the use of a Look Ahead method based on measuring the difference between two formulas F and F' in terms of size and equivalence. This look ahead operation is performed in a part of the free variables and only applied if a second look ahead does not indicate it will cause a conflict.

Algorithm 1 PartialLookahead( )

```
for each variable $x_i$ in $\mathcal{P}$ do
    $\mathcal{F}' := \text{IterativeUnitPropagation}(\mathcal{F} \cup \{x_i\})$
    $\mathcal{F}'' := \text{IterativeUnitPropagation}(\mathcal{F} \cup \{\neg x_i\})$
    if $\mathcal{F}' \ll \mathcal{F}$ and $\emptyset \notin \mathcal{F}'$ then
        $\mathcal{F}' := \text{DoubleLookahead}(\mathcal{F}')$
    else if $\mathcal{F}'' \ll \mathcal{F}$ and $\emptyset \notin \mathcal{F}''$ then
        $\mathcal{F}'' := \text{DoubleLookahead}(\mathcal{F}'')$
    end if
    if $\emptyset \in \mathcal{F}'$ and $\emptyset \in \mathcal{F}''$ then
        return "unsatisfiable"
    else if $\emptyset \in \mathcal{F}'$ then
        $\mathcal{F} := \mathcal{F}''$
    else if $\emptyset \in \mathcal{F}''$ then
        $\mathcal{F} := \mathcal{F}'$
    else
        $\text{H}(x_i) := 1024 \times \text{Diff}(\mathcal{F}, \mathcal{F}') \times \text{Diff}(\mathcal{F}, \mathcal{F}'')$
                       $+ \text{Diff}(\mathcal{F}, \mathcal{F}') + \text{Diff}(\mathcal{F}, \mathcal{F}'')$
    end if
end for
return $x_i$ with highest $\text{H}(x_i)$ to branch on
```

Algorithm 2 DoubleLookahead($\mathcal{F}$)

```
for each literal $l_i$ in $\mathcal{D}$ do
    $\mathcal{F}' := \text{IterativeUnitPropagation}(\mathcal{F} \cup \{l_i\})$
    if $\emptyset \in \mathcal{F}'$ then
        $\mathcal{F} := \text{IterativeUnitPropagation}(\mathcal{F} \cup \{\neg l_i\})$
        if $\emptyset \in \mathcal{F}$ then
            break
        end if
    end if
end for
return $\mathcal{F}$
```

Algorithm 3 IterativeUnitPropagation($\mathcal{F}$)

```
while unit clause $y \in \mathcal{F}$ and $\emptyset \notin \mathcal{F}$ do
    satisfy $y$ and simplify $\mathcal{F}$
end while
return $\mathcal{F}$
```

Unfortunately, the algorithm does not allow any parameters nor the traces provide much information other than the number of successful look-aheads and double look aheads.

It provides information about the complexity of the problem since from the original set of clauses and variables extracts the number of free variables eliminating constraints that are satisfied initially.

For the configuration sl15_2 we obtain:

```
$ ./march_rw.exe sl15_2cnf.txt
c main()::***              [ march satisfiability solver ]                   ***
c main()::**Copyright (C)2001-2009 M.J.H.Heule, J.E. van Zwieten, and M.Dufour **
c main()::   *  This program may be redistributed and/or modified under the terms of
the GNU Gereral Public License  *
c main()::
c initFormula():: searching for DIMACS p-line....
c initFormula():: the DIMACS p-line indicates a CNF of 7264 variables and 21716
clauses.
c parseCNF():: parsing....
c parseCNF():: the CNF contains 292 unary clauses.
c runParser():: parsing was successful, warming up engines...
c find_and_remove_tautogolies():: found and removed 437 tautologies
c find_and_remove_tautogolies():: found and removed 454 tautologies
c find_and_remove_tautogolies():: found and removed 1 tautologies
c simplify_formula():: removed 6 tautological, 17574 satisfied and 3 duplicate clauses
c preprocessing fase I completed:: there are now 1352 free variables and 4133 clauses.
c using 3-SAT heuristics (occurence based diff)
```

```
c simplify_formula():: removed 0 tautological, 0 satisfied and 0 duplicate clauses
c simplify_formula():: removed 0 tautological, 0 satisfied and 356 duplicate clauses
c main():: clause / variable ratio: ( 4133 / 7264 ) = 0.57
c longest clause has size 3
c simplify_formula():: removed 0 tautological, 0 satisfied and 0 duplicate clauses
c number of free variables = 1352
c dynamic_preselect_setsize :: off
c main():: all systems go!
c |---------------------------------------------------------------|
c |.000000$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$|
c |
c main():: nodeCount: 7
c main():: dead ends in main: 0
c main():: lookAheadCount: 967
c main():: unitResolveCount: 1352
c main():: time=0.234000
c main():: necessary_assignments: 77
c main():: bin_sat: 0, bin_unsat 0
c main():: doublelook: #: 12, succes #: 2
c main():: doublelook: overall 1.311 of all possible doublelooks executed
c main():: doublelook: succesrate: 16.667, average DL_trigger: 45.179
c main():: SOLUTION VERIFIED :-)
s SATISFIABLE
```

while if the input is an already stable configuration(sl15), we obtain that all clauses are initially satisfied and hence there is no work to do by the solver:

```
$ ./march_rw.exe sl15cnf.txt
c main()::***              [ march satisfiability solver ]                    ***
c main()::**Copyright (C)2001-2009 M.J.H.Heule, J.E. van Zwieten, and M.Dufour **
c main()::   *  This program may be redistributed and/or modified under the terms of
the GNU Gereral Public License  *
c main()::
c initFormula():: searching for DIMACS p-line....
c initFormula():: the DIMACS p-line indicates a CNF of 7264 variables and 21716
clauses.
c parseCNF():: parsing....
c parseCNF():: the CNF contains 394 unary clauses.
c runParser():: parsing was successful, warming up engines...
c simplify_formula():: removed 0 tautological, 21716 satisfied and 0 duplicate clauses
c preprocessing fase I completed:: there are now 0 free variables and 0 clauses.
c using 3-SAT heuristics (occurence based diff)
c simplify_formula():: removed 0 tautological, 0 satisfied and 0 duplicate clauses
c simplify_formula():: removed 0 tautological, 0 satisfied and 0 duplicate clauses
c main():: clause / variable ratio: ( 0 / 7264 ) = 0.00
c longest clause has size 0
c simplify_formula():: removed 0 tautological, 0 satisfied and 0 duplicate clauses
c number of free variables = 0
c dynamic_preselect_setsize :: off
c main():: all systems go!
c |---------------------------------------------------------------|
c |.0$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$|
c |
c
c main():: nodeCount: 1
c main():: dead ends in main: 0
c main():: lookAheadCount: 0
c main():: unitResolveCount: 0
c main():: time=0.140000
c main():: necessary_assignments: 0
c main():: bin_sat: 0, bin_unsat 0
c main():: doublelook: #: 0, succes #: 0
c main():: doublelook: overall nan of all possible doublelooks executed
c main():: doublelook: succesrate: nan, average DL_trigger: nan
c main():: SOLUTION VERIFIED :-)
s SATISFIABLE
```