CSP Project for Game of Life

CPP- MAI – UPC

José A. Magaña Mesa

January 2014

# Introduction

This report corresponds to the implementation of the Conway’s Game of Life using Constraint Satisfaction Programming techniques. The objective is to find stable configurations starting from a given setup and that the stable configuration maximizes the number of Alive Cells in the board.

The Gecode libraries have been used for this using IntVars for the model defined. As an extension, some of the different options in the library for Variable and Value selection have been evaluated to see the difference in performance. Also, a set of redundant constraints have been defined that are included in the model based on command line settings.

# Implementation

## Constraints

For a direct implementation of the Conway’s Game of Life rules, the following constraints must be added. On them, the matrix minimodel has been used so that manipulation of cells becomes straightforward.

As it was the case for the SAT problem, the board has been widened in order to avoid having to care about the position of the cell when looking for the neighbor cells.

For each Alive Cell in the original configuration, we assign the value:

rel(\*this, A(x,y), IRT\_EQ, 1); // the cell is alive

For all the cells (i,j) in the original board (not extended), the neighborhood is calculated in the variable s, that adds the number of Alive Cells. The conditions for Alive Cells and Dead Cells are added. Operator >> implements the conditional in logic (🡪).

s=A(i-1,j-1)+A(i-1,j)+A(i-1,j+1)+A(i,j-1)+A(i,j+1)+A(i+1,j-1)+A(i+1,j)+A(i+1,j+1);

rel(\*this, (A(i,j)==0) >> (s!=3));

rel(\*this, (A(i,j)==1) >> (s==2) || (s==3));

Additional conditions are added on the side cells, to prevent propagation on a potentially infinite board:

// horizontal

rel(\*this, A(1,i)+A(1,i+1)+A(1,i+2)<3);

rel(\*this, A(n,i)+A(n,i+1)+A(n,i+2)<3);

// vertical

rel(\*this, A(i,1)+A(i+1,1)+A(i+2,1)<3);

rel(\*this, A(i,n)+A(i+1,n)+A(i+2,n)<3);

The margin of the extended board is set to zero (horizontally and vertically):

rel(\*this, A(i,0), IRT\_EQ, 0);

rel(\*this, A(i,n+1), IRT\_EQ, 0);

rel(\*this, A(0,i), IRT\_EQ, 0);

rel(\*this, A(n+1,i), IRT\_EQ, 0);

## Cost function

As the problem has as objective maximizing the number of Alive Cells, the cost function is directly the number of Alive Cells, that is, the sum of the values of the board. The minimum of the function is the number of Alive Cells in the initial configuration.

upperBound=n\*n;

Cost=IntVar(\*this,nAlive,upperBound); // cost: minimum is the number of alive cells // maximum is the number of cells

rel(\*this,Cost==sum(X));

The maximum can also be bounded based on the results of [1] and set to half the size of the board plus a delta. Nevertheless, this optimization has not been incorporated to the solution and the upper bound has been set to the number of cells in the board.

Using this optimization would highly speed up the search process as it would prune plenty of options that are not feasible solutions but that the algorithm must discard itself.

# Evaluation

In order to efficiently calculate all the mentioned cases, intensive use of the parameterization of the library has been used.

solutions(0) 🡪 all the solutions must be explored

Dynamically, the Variable and Value selection are set:

setVar(----)

setVal(----)

In addition, an extra parameter has been set to decide if the redundant constraints must be incorporated to the model:

opt.model(0); // by default: no redundant

opt.model(1,"redundant");

opt.model(0,"noredundant");

The following command line parameters have been used:

-print-last 1 🡪 only the last (best) solution is printed

-file-sol solsl15\_2.txt 🡪 the solution is printed to a defined file

-mode solution 🡪required to obtain the solution on the file

-model redundant /noredundant 🡪the model may or not incorporate redundant constraints

sl15\_2.txt 🡪 the input file

Some incompatibilities between the settings have limited the use of certain functionalities. It is not possible for example to export the solutions for each setting set to the same file obtaining a list. Each execution deletes the existing content on the file. The same happens if we try to obtain the statistics.

If the experiment wants to be repeated for a number of iterations so that we can extract averages to compensate for any stochasticity in the algorithm, the setting –mode time must be used but this does not produce the solution.

## How to execute

To ease validation of the model, a .bat file (GoL.bat) has been provided that takes as parameters the input file: <input.txt>

The output will have the solution in a file named: sol<input.txt>

The solution includes in the last line the value of the cost function for the configuration.

The batch file can be easily converted to a script in any Linux environment where it needs to be executed.

## Variable and Value Selection impact

In order to evaluate how variable and value selection influence the algorithm the following predefined methods have been selected. Some of them may not make sense for cases, as this one, where the domain is {0,1}. In both cases, the test has been limited to options that do not require non-trivial parameters. In the case of the Random selection, it has been initialized with a 0 seed for repeatability.

The execution times expressed in milliseconds for the sl15\_2.txt example are shown in the table below. The results correspond to a single execution what is not a limitations as the methods are most of them, and except for ties, deterministic.

The results are colored with red for the higher (worst) execution times and green for the lower execution times.



From the table we can observe that one of the axes dominates the other in the sense that the values follow the same order if we fix one of the values of the non-dominant axis. The dominant axis is the Variable selection. Once we fix that, the Value selection has little influence in the result (except for the Random case).



A basic analysis of the results brings interesting conclusions. The most favorable settings are those that use the Size of the Domain of the Variable (INT\_VAR\_SIZE\_MIN and INT\_VAR\_SIZE\_MAX). The best execution time is for INT\_VAR\_SIZE\_MIN what in this case where the domain has cardinality 2, indicates that first all the cells whose value is defined either in the original configuration or by the application of the constraints are fixed and only after this the rest of cells are calculated.

The fact that the second best option is INT\_VAR\_SIZE\_MAX can be interpreted as that the other criteria are not relevant and following any strategy based on domain size is a winning strategy, especially considering that in this case, the domain size is 2.

Good results, in some case close to the best obtained, are achieved for INT\_VAR\_ACTIVITY\_SIZE\_MIN and INT\_VAR\_ACTIVITY\_MIN variables. No detail is provided on Gecode Documentation about how Activity is measured so it is not possible to analyze further for this case.

Regarding the Value Selection Variable, the table shows that for better performance they must be correlated with the Variable Selection. Hence, if the Variable Selection is a ???\_MAX the Value Selection must also be a ???\_MAX. This is especially evident in the INT\_VAR\_ACTIVITY\_???\_MAX cases where the execution time is triplicated.

The fact that the worst values are in the cases where random criteria are used is a clear indicator (despite the low statistical significance of the experiment as it has only been repeated once) that Value and Variable selection are key to achieve good performance. If the results for random where closer to the best it would have been absolutely necessary to repeat several times the results.

Another interesting result is that in most cases the number of solutions found to get to the optimal is 2.

Only in a few cases, the first solution is the optimal, curiously in all cases there is a Random element, and curiously also there is a MIN and a MAX component.

INT\_VAR\_ACTIVITY\_MIN() - INT\_VAL\_RND(0), time=11 msec

INT\_VAR\_SIZE\_MAX() - INT\_VAL\_RND(0), time=11 msec

INT\_VAR\_RND(0) - INT\_VALUES\_MAX(), time= 244 msec

INT\_VAR\_RND(0) - INT\_VALUES\_MIN(), time=192 msec

In other few cases, 3 solutions are required to find an optimal one. Again a Random factor appears:

INT\_VAR\_RND(0) - INT\_VAL\_RANGE\_MAX(), time=135 msec

INT\_VAR\_RND(0) - INT\_VAL\_SPLIT\_MAX(),time=151 msec

INT\_VAR\_RND(0) - INT\_VAL\_SPLIT\_MIN(), time=193 msec

# Redundant constraints

In order to try to make the search quicker and in general more efficient, some constraints have been added to the problem.

First, for those cells that are Alive in the initial configuration we can include:

r=expr(\*this, A(x-1,y-1)+A(x-1,y)+A(x-1,y+1)+

A(x,y-1)+A(x,y+1)+

A(x+1,y-1)+A(x+1,y)+A(x+1,y+1));

rel(\*this, (r==2) || (r==3) );

This is the same constraint that we added for all the cells in the board but knowing that the cell is Alive.

A second set of redundant constraints explodes the reverse conditional constraints:

rel(\*this, (s>3) >> (A(i,j)==0));

rel(\*this, (s==1) >> (A(i,j)==0));

rel(\*this, (s==0) >> (A(i,j)==0));

rel(\*this, (s==3) >> (A(i,j)==1));

For those values of the number of alive cells that only permit one value for the cell the condition is added as constraint.

# Results with Redundant Constraints

Adding the redundant constraints and executing for all the configurations, the results in the following table are obtained. The first observation is that the execution is in all cases worse than without the redundant constraints. The best times are obtained for the same parameters.



By adding the redundant constraints, the number of propagators in the model is increased from 1575(or 1574) to 3900(or 3899) depending on the settings. Surprisingly (for novice users of Gecode) the number of propagators does not depend only on the constraints added.

An analysis of additional performance values is required. For some of the Variable Selection policies (INT\_VAR\_SIZE\_MIN, INT\_VAR\_ACTIVITY\_SIZE\_MIN , INT\_VAR\_ACTIVITY\_SIZE\_MAX and INT\_VAR\_RND) an extended analysis is performed.

The first observation is that execution time is proportional to Propagations and Nodes. The table for Propagations is shown:



The pattern is the same than for the model without redundant clauses, also the number of solutions is 1 for the same cases.

# Conclusions

Gecode provides is very comprehensive library for modelling CSP problems. In this case, we have used Integer Variables and arithmetic and logical constraints to model the problem.

The model has been evaluated successfully using the same test set that was used for the SAT implementation of the same problem.

Intuitively, it would have been expected, being a Max Optimization problem, that selecting the Maximum value(1) had been a better strategy and had served for both obtaining faster execution time and a smaller exploration of the search tree as the cost function for a solution is used as a criteria for pruning solutions. But the results show that INT\_VAL\_MAX/INT\_VALUES\_MAX does not provide better results. Variable selection is a more discriminant factor to improve performance.

When the objective is optimization of a given function and it is required to explore the whole search tree (applying pruning when possible), adding redundant constraints may not originate performance improvements and as it has been the case can even increase execution time significantly.

# References

[1] Chu, Geoffrey, Peter J. Stuckey, and Maria Garcia De La Banda. "Using relaxations in maximum density still life." *Principles and Practice of Constraint Programming-CP 2009*. Springer Berlin Heidelberg, 2009. 258-273.