# Bellman: a program for finding catalysts in cellular automata

*Mike Playle, Cambridge, UK*

*Draft B – 2 August 2014*

---

## 1 Introduction

Bellman, named after the character in Lewis Carroll's *The Hunting of the Snark*, is a program for searching for catalytic interactions in Conway's Game of Life and potentially other similar cellular automata. Results are derived directly from the automaton's evolution rule, not generated from a list of candidate catalysts as with previous searchers. The algorithm uses a “divide and conquer” approach to systematically examine large volumes of search space by assigning values to previously unknown cells and examining the consequences. Aggressive pruning of the search tree is used to render large problems tractable.

In its present condition Bellman is not suitable for use by non-technical users. This document assumes basic familiarity with standard terminology in Conway's Life and with compiling and running programs from the Unix command line. Sections 4 and 5 are more technical and also assume some familiarity with mathematical notation and with software implementation and optimisation techniques. The reader who just wishes to run searches without understanding the details should feel free to stop at the end of section 3.

## 2 Getting Started

Bellman was developed on a 64 bit Linux machine. To compile it, follow the conventional process of typing "make" and then fixing any errors. It requires the "gd" library for writing GIF files (on Ubuntu this is provided by the "libgd2-xpm-dev" package). Further guidance on fixing compile errors is outside the scope of this document.

Once built, this search takes a minute or two and produces 47 results:

```
$ time ./bellman inputs/test.in
```

The results produced by this stage are partial - they only contain active catalytic regions, not entire still lives. The program `mkstill` assigns values to the remaining cells to produce a finished pattern that can be viewed in Golly or other Life program:

```
$ ./mkstill result000001-4.out
Best answer: 7
Best answer: 5
Best answer: 4
Best answer: 2
#P 5 2
.*......
..*.....
***.....
........
....**..
....*.*.
......*.
......**
```

As a more interesting example I've also included the input file which found the snark

reflector, as "inputs/snark.in".

# 3 Reference Manual

The process of searching for catalysts consists of:

- Constructing an input file describing the space to be searched
- Running the searcher to generate candidate catalysts
- Classifying the partial results to eliminate trivial duplicates
- Filtering the partial results to identify potentially 'interesting' catalysts
- Completing the most interesting partial results to produce finished Life patterns

## *3.1 Constructing an input file*

Bellman reads an input file containing:

- a Life pattern whose evolution is to be modified by the catalyst
- some unknown cells, to which Bellman will attempt to assign values
- optionally, some cells which are already part of the catalyst
- filter patterns which must appear at particular times and places in the resulting catalysed reaction
- parameters used to prune the search space

The format has some similarities with the popular “Life 1.05” format, and can be prepared in a standard text editor. Lines beginning with a **#** character set parameters and introduce patterns; other lines represent the states of cells in regions of the universe.

### *3.1.1 Pattern definition*

This example shows a search for small patterns which react with an incoming glider.

```
#P 0 0
..............
..............
......@.......
.......@......
.....@@@......
.........???????
.........???????
.........???????
.........???????
.........???????
.........???????
.........???????
.........???????
```

The `#P` at the start of the pattern section sets its origin coordinates; this is followed by a series of lines which “draw” the pattern using the following ASCII characters:

| | |
|---|---|
| `.` | Dead cells. Whitespace can be used to represent dead cells too. |
| `*` | Cells which are live in the catalyst. |
| `@` | Cells which are live in the evolving pattern which is being perturbed. |

| | |
|---|---|
| ? | Cells which may or may not be live in the catalyst. Bellman will output patterns with values assigned to some or all of these cells. |

### 3.1.2 Filter definition

This example filter tells Bellman to exclude patterns that don't contain a glider travelling northeast at position (20, 11) in generation 46 of the search result:

```
#F 46 19 10
.....
.***.
...*.
..*..
.....
```

These sections are similar to pattern definitions except that they are introduced with a `#F` line which provides a generation number in addition to an (x, y) position. Patterns will be excluded from the search results if their evolving state doesn't match the '`*`' and '`.`' cells in the filter pattern. Filters don't have to be rectangular; use '`?`' in a filter pattern to indicate a "don't care" cell which will match any evolving pattern.

### 3.1.3 Search tree pruning parameters

Various numerical parameters can be used to restrict the amount of search space examined, thus speeding up the search at the cost of excluding many potential solutions from the output:

```
#S first-encounter 24
#S last-encounter 65
#S repair-interval 50
#S stable-interval 8
#S max-live 500
#S max-active 8
```

The meanings of the various parameters in these sections are:

| | |
|---|---|
| `first-encounter` | The earliest generation at which the catalyst is permitted to interact with the evolving pattern. Any unknown cells which interact with the evolving pattern prior to this generation will be cleared in the output. |
| `last-encounter` | The latest generation at which interaction can begin. Catalysts which don't interact with the pattern before this time will be excluded from the output. |
| `repair-interval` | The maximum time that the interaction between the pattern and the catalyst is permitted to last. The catalyst must return to its pre-interaction state within this many generations of the interaction beginning, or it will be excluded from the output. |
| `stable-interval` | Number of generations for which the catalyst must remain untouched by the evolving pattern after the interaction ends. |
| `max-live` | The number of unknown cells which Bellman is permitted to turn on in its output. Note that this doesn't count cells which were already on in the input file (those marked as '*' in the pattern section). |
| `max-active` | The number of cells in the catalyst which are permitted to |

| | |
|---|---|
| | differ from their stable state during the interaction. If more cells than this change from their stable state then the catalyst is assumed to be 'exploding' and is excluded from the search. |

### 3.1.4 Symmetry parameters

It is possible to restrict the search to patterns with a horizontal or vertical axis of symmetry:

```
#S symmetry 6:horiz-odd
```

Valid symmetry types are `horiz-odd`, `horiz-even`, `vert-odd` or `vert-even`.

The number is the x-coordinate or y-coordinate of the axis; for `horiz-odd` and `vert-odd` symmetries the axis passes through the centre of the selected row or column of cells, while for horiz-even and vert-even the axis is between the selected row and the next.

The example inputs under `test-cases/` might make this more obvious.

## 3.2 Running the searcher

The `bellman` command-line program reads the input file produced previously and performs most of the work of the search. Any patterns it finds get written into the directory you run it from, so you may want to change into a new directory before starting it.

For each result it finds, three output files are created:

| | |
|---|---|
| `result-NNNNNN-T.out` | The discovered pattern, in the same format as the input file (with some of the '?' cells replaced with '.' or '*' as appropriate). |
| `result-NNNNNN-T-partial.gif` | GIF image file showing the generated catalyst in its stable, non-interacting state. |
| `result-NNNNNN-T-full.gif` | Animated GIF image file showing the evolution of the perturbed pattern. |

Every 10 seconds while running, the program will print a summary of the decisions it has made, which is useful for monitoring its progress and adjusting the input parameters.

## 3.3 Classifying partial results

The raw output from `bellman` is not very readable and often contains many near-duplicates. To make it easier to monitor the progress of the search, there is a script which produces a summary of the results:

```
$ python mkhtml.py <path to directory with results>
```

This will generate an `index.html` file in the same directory which can be viewed in a browser.

The script can be run before bellman has finished, and then re-run when more patterns are found.

## 3.4 Filtering partial results

This stage of the search selects interesting partial results using a classifier based on a neural network of approximately $10^{11}$ nodes which was generated using a genetic algorithm over a timespan of several hundred million years. Unfortunately I am unable to reveal

further details of the operation of this algorithm.

## 3.5 Completing partial results

Bellman only chooses values for cells which actually affect the pattern's evolution. Unneeded cells remain unknown in the program's output. As a result, most output patterns are incomplete, containing only the active region of the catalyst; more live cells must be added to stabilise the edges.

The `mkstill` command-line program will read `bellman`'s `.out` files and search for such a stabilisation. If any exist, it outputs one with minimal population, otherwise it prints "No solution." It can be quite slow – it is often much faster to inspect the `.out` file first and complete any fishhooks, eater2s etc manually. Also it is sometimes necessary to manually add additional '?' cells outside the original unknown region, if an active pattern was found too close to the edge.

Redirecting its output produces a file which can be read by Golly:

```
$ ./mkstill result000001-4.out > eater.l
Best answer: 7
Best answer: 5
Best answer: 4
Best answer: 2
$ cat eater.l
#P 5 2
.*......
..*.....
***.....
........
....**..
....*.*.
......*.
......**
```

It is not uncommon for `bellman` to produce false positives: candidate catalysts for which no stabilisation exists.

## 3.6 Tips and tricks

### 3.6.1 Iterative deepening

The search space grows rapidly as the `repair-interval`, `max-live` and `max-active` parameters are increased. Small changes can make the difference between a search which runs for minutes and one which runs for months. Experimenting with different values for these parameters is highly advisable in practice. One useful approach is to start with a parameter set to 1 and run a series of searches, increasing it by 1 each time and monitoring the run time of the search and the total number of tree prunes performed.

### 3.6.2 Dealing with the eater2

This common family of catalysts consists of a block that reacts with the evolving pattern surrounded by a framework of live cells which causes the dying block to reconstruct itself.

When `bellman` finds an eater2 it is not unusual for it to subsequently find several dozen more with slightly different supporting frameworks. This causes it to repeatedly re-examine large volumes of search space that are encountered after the reaction with the eater2.

Interesting results can often be found more quickly by inserting single OFF cells into the unknown region. Setting one cell of an eater2's block to OFF is enough to exclude it and all its many variants from the search.

# 4 Technical Description

## *4.1 Partial-knowledge automaton*

For any cellular automaton A with N states, there is a corresponding automaton PK(A) with N+1 states, which is derived by adjoining an additional state to represent cells whose value in A is unknown. States of PK(A) thus represent sets of possible states of A, and the evolution of a state in PK(A) can be used to draw conclusions about the evolution of all the corresponding states of A.

PK(A)'s state transition table contains A's transition table as a subset, representing the evolution of a cell whose neighbours all have known values. In the case where one or more of a cell's neighbours has unknown value, all possible states of the corresponding neighbourhood in A must be considered. If they would all evolve to the same state in A, then they evolve to the corresponding state in PK(A) too, however if there is any ambiguity then the neighbourhood in PK(A) evolves to the unknown state. Note that if A is outer-totalistic then so is PK(A).

Bellman operates primarily by evolving patterns in PK(Conway's Life). However in this automaton, regions of unknown cells tend to grow at the speed of light, so a fourth “unknown-stable” state is used to represent cells whose value is unknown but is assumed to be part of a stable catalyst. This is implemented in an ad-hoc manner by `bellman_evolve()` at the same time as it checks for filter mismatches etc.

## *4.2 Bitslice representation*

Modern CPUs generally have 32 or 64 bit wide registers. The CA representation used by Bellman attempts to make efficient use of the whole width of the register to perform calculations on 32 or 64 cells of the universe simultaneously. Dividing the universe into register-width “tiles” results in code which tends to compile into straight-line blocks of logical and arithmetic operations and conditional moves, with few data-dependent branches; the theory is that this should result in efficient execution on modern PC CPUs, with few cache misses or branch mispredictions, though this has not been rigorously adhered to or benchmarked in detail.

## *4.3 Backtracking search mechanism*

## *4.4 Changing the evolution rule*

The PK(Life) evolution rule and an auxiliary 'stabilisation' rule are implemented as bit-parallel operations in `evolve_bitwise.c`. These blocks of code were produced with the included `mkrule.py` script, which derives both sets of logical operations from the behaviour of the `life_rule()` function, and relies on the `espresso` Boolean logic minimiser.

To re-run this step, at a minimum you will have to edit `life_rule()`, re-run both the `make_3state_rule()` and `make_stabilise_rule()` paths, and cut-and-paste the results into `evolve_bitwise.c`.

Supporting rules with B0 or B1 will be more complex. B0 implies that a completely dead

tile will change state on the next generation, an assumption that violates many assumptions made in the design of the tile-based data structure. Rules with B1 will be easier to support, but will require the tile structure to be able to expand diagonally, not just orthogonally.

# 5 Further Work

## *5.1 Distributed / parallel searching*

This approach should be well suited to massively parallel or distributed systems. A quick initial round of searching can be used to split the problem space into as many subproblems as desired, which can be farmed out to individual compute nodes for the bulk of the work.

## *5.2 FPGA acceleration*

## *5.3 Investigate time complexity of the search*