Searching for patterns in Conway's Game of Life

Johan Bontes

Minor dissertation presented in partial fulfilment of the requirements for the degree of

Master of Science

in the department of Computer Science

University of Cape Town

Supervisor: Prof. James Gain

18 September 2019

Abstract

Conway's Game of Life (Life) is a simple cellular automaton, discovered by John Conway in 1970, that exhibits complex emergent behavior.

Life-enthusiasts have been looking for building blocks with specific properties (patterns) to answer unsolved problems in Life for the past five decades.

Finding patterns in Life is difficult due to the large search space. Current search algorithms use an explorative approach based on the rules of the game, but this can only sample a small fraction of the search space.

More recently, people have used SAT solvers to search for patterns. These solvers are not specifically tuned to this problem and thus waste a lot of time processing Life's rules in an engine that does not understand them.

We propose a novel SAT-based approach that replaces the binary tree used by traditional SAT solvers with a grid-based approach, complemented by an injection of Game of Life specific knowledge.

This leads to a significant speedup in searching. As a fortunate side effect, our solver can be generalized to solve general SAT problems. Because it is grid-based, all manipulations are embarrassingly parallel, allowing implementation on massively parallel hardware.

This work is dedicated to the memory of Dietrich 'Dieter' Leithner.

Contents

List of figures

List of tables

Listings

Plagiarism declaration

I know the meaning of plagiarism and declare that all of the work in this dissertation, save for that which is properly acknowledged, is my own.

Johan Bontes

Chapter 1: Introduction

In 1970 Martin Garner published an article on John Conway's Game of Life (Life) in Scientific American [23], introducing the concept of cellular automata (CA) to the general public for the first time. A CA is a simple mathematical model with properties of space and time; a machine consisting of an arrangement of cells in a lattice which evolves over time. Although his was not the first cellular automaton, the publication of its hypnotic order and chaos made Conway a celebrity overnight, one of only a handful of mathematicians to reach such status[1].

The rules of Life cause cells to interact in complex ways, but if a pattern designer desires some specific behavior, most of these interactions are useless. Patterns that interact in specific ways are rare and hard to find computationally. Conway's Life is too simple to allow for fine-tuning using complex rules or many states and thus functionality can only arise naturally, as a consequence of its evolution.

A number of people [3, 19, 63, 64, 69] have devised search algorithms to find Life patterns that exhibit specific properties. We intend to improve on the current state-of-the-art by feeding knowledge specific to Life into a novel solver for the boolean satisfiability problem. This is one of the fundamental problems in computer science, which aims to find a satisfying assignment to a given expression in boolean logic.

1.1 Problem description

Traditional search algorithms Many search algorithms for Life use a trail-and-error method to search a binary tree containing all possible patterns that fit the search description, either using depth-first, or breadth-first traversal, or a combination of the two. Whilst a number of interesting patterns have been found using this approach, it does not scale beyond low hanging fruit. This fruit consists of either patterns that are small, making it easy to search a sizable fraction of the search space, or patterns that are common, meaning that there are many trees in the orchard.

Because the search space expands exponentially with the number of cells under investigation, throwing more computing power at the search does not help much either. A new approach is needed.

Sat solvers Since the year 2000, solvers for the boolean satisfiability problem (Sat)—employed to solve combinatorial problems—have shown spectacular increases in performance. Modern solvers are able to resolve problems with millions of variables, seemingly defying the NP-complete nature of this problem. Sat solvers have successfully been used to find Life patterns that have defied traditional search for decades [30].

Impressive as this may seem, there is a problem: up to now, general-purpose Sat solvers have been used. This means that the rules of Life have to be 'taught' to the solver. This expands the size of the problem about 100-fold. Also, the solver has to compute the evolution of a pattern using its inherent logic. In effect, the search is shoehorned into an alien representation dictated by the general-purpose solver, sacrificing several orders of magnitude of performance.

Another issue is that current Sat solvers are stuck with sequential implementations. In the 2016 Sat solver competition, some parallel solvers performed slower on 48 cores than on a single core [1]. Successful implementations that run on a GPU have not yet been found. This leaves the 100 to 1 000× speedup that such an implementation might provide on the table.

[1] The fact that John Conway is a notorious jester and eccentric did little to diminish this status.

We aim to create an SAT solver which is specifically tuned to the Game of Life and give this solver access to an algorithm which can calculate possible past and future patterns for any given start position. This will eliminate the need to have the solver infer the rules of the game using logic and allow us to perform searches on parallel hardware.

1.2 The GridWalker algorithm

To solve our problem, we create an algorithm called GridWalker. This replaces the binary tree search, used by traditional SAT solving algorithms, with a grid-based approach. This method is better suited to GPUs. It is also a good fit for the 2D-lattice that Life employs. Hence, neatly solving the mapping of the spatial dimensions of the problem.

In the experiments chapter, we show that starting from first principles—a lookup table with 3 entries—the GridWalker algorithm can enumerate all ancestors of a given starting configuration with any combination of `on`, `off`, and `unknown` cells.

Our grid-based solver does not waste time trying to reason about the past or future evolution of a pattern, but simply looks up all possible outcomes. Because we do not 'teach' the rules of the game to the solver, there might be a number of patterns it cannot solve. To plug this gap, we re-use our Life iteration algorithm to validate any candidate solutions GridWalker serves up. These remaining candidates are a tiny fraction of the total search space. This is the approach traditionally used in search programs for Life. The difference in our solution is that we only have to apply this step to a small subset of candidates, rather than to every entry in the search space.

1.3 Research question

Can we improve the execution speed of searches for patterns in Conway's Life—compared to the currently best performing search algorithms—by creating a SAT solver based algorithm that uses domain specific knowledge?

1.4 Contributions

1. An implementation of a Life iteration algorithm on a GPU that outperforms the current fastest [25] by 1 000 000× (see section 3.2).

2. A grid-based SAT solver specifically tuned to finding patterns in the Game of Life (see chapter 5). Extending this approach to general-purpose SAT solving is an avenue for future research.

1.5 Overview of the thesis

We explore the Game of Life and a number of current search algorithms in chapter 2. Chapter 3 details our Life iteration algorithm and its implementation on a GPU. We take a short detour in chapter 4 to explain the workings of the current crop of award-winning SAT solvers. Next, chapter 5 returns to the core issue and discusses the GridWalker lattice-based search algorithm. A number of relevant experiments are described in chapter 6.

Finally, the conclusion summarizes the results obtained and talks about future avenues to extend this research.

In appendix A some of Life's colorful lingo is explained; appendix B shows where the software developed to conduct the experiments can be obtained.

Chapter 2: What is Life?

In the 1940's Ulam and Von Neumann set out to prove that artificial life is possible by constructing a machine that can create a copy of itself: a self-replicating machine. They invented a cellular automaton with cells that interact on a rectangular lattice where each cell can be in one of 29 states. A self-replicating machine was built using a universal constructor—a pattern that can construct any pattern given a blueprint—the details of which were posthumously published in 1966 by Burks [72].

A cellular automata (CA) is a simple mathematical model with properties of space and time; A CA is a machine consisting of an arrangement of cells in a lattice, which has one or more spatial and a single time dimension.

Expanding on their work, John Conway invented a simpler version of the complex machine. His predecessors had a toolkit of states, each of which performed a specific function. Conway abhorred this handcrafting and wanted any complex behavior to arise naturally from the simplest possible beginnings. After struggling for 18 months to get a 2-state game to work, he tried a 3-state system with two genders and a mating rule requiring three parents, mischievously called 'Actresses and Bishops' [62]. The rules worked, but were deemed unnecessarily complex; he dropped the gender requirement and settled on a simplified rule with only two states.

The simplified rule, known as 'B3/S23', is one of a few rules in its rule-space that displays interesting behavior without requiring fine-tuned starting patterns [60].

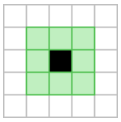


Figure 2.1: Eight neighbors surround a single cell

Little could have prepared John Conway for the pandemonium that ensued after Martin Garner's column on Conway's game of Life [23] in Scientific American. It instantly made him one of the world's most famous mathematicians[1].

2.1 Rules and patterns

The game uses the following simple rules:

environment The game is played on an infinite 2-dimensional rectangular lattice, the *universe*, where every cell has 8 neighbors, corresponding to the eight squares a king in chess can visit in a single move: the Moore neighborhood [56]. Cells have 2 states: alive or dead.

birth A new cell is born at $t+1$, when a dead cell at t is surrounded by 3 live neighbor cells.

death A live cell dies at $t+1$, when at t it has less than 2 (underpopulation) or more than 3 live neighbors (overcrowding), otherwise it remains alive.

evolution The rules are applied to all cells simultaneously in discrete time intervals called *generations*.

epoch The game is seeded with an initial population from which it then develops deterministically.

[1] Conway resents Life overshadowing his other achievements [62], famously declaring to 'hate the Game of Life'.

2.1.1 Cell interactions

The interactions between neighboring cells, based on the rules of Life, can be traced by simply counting alive neighbors for every cell in a given generation.

gen. n					gen. $n+1$				
0	0	0	0	0	0	1	1	1	0
1	2	3	2	1	0	2	1	2	0
1	1	2	1	1	0	3	2	3	0
1	2	3	2	1	0	2	1	2	0
0	0	0	0	0	0	1	1	1	0

Figure 2.2: The evolution of a blinker

Let us use the blinker shown in figure 2.2 as an example. The center pixel has two live neighbors in either configuration and thus stays the same. The dead pixels orthogonal to the center have three live neighbors and are thus born, whilst the live pixels next to the center only see a single live cell each and thus die out in the next generation. It is helpful to label each cell with its count of live neighbors: in the next generation all cells with 3 live neighbors are born, all cells with 2 neighbors stay the same and all other cells will die.

2.1.2 Patterns in Life

Cells interact with their neighbors to form *patterns*: constellations of live and dead cells that evolve over time. The simplest behavior is a stable pattern: a *still life*.

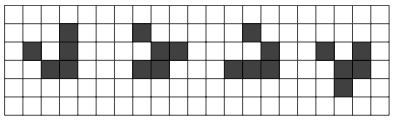
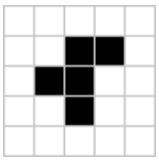


Figure 2.3: The four phases of a glider

Many patterns settle in a repeating cycle: an *oscillator*. An oscillator that moves across the lattice is called a *spaceship*. The smallest known spaceship is the diagonally traveling *glider*, shown in figure 2.3. Some patterns die out after a number of generations. In other cases a small starting pattern explodes before settling in a stable configuration of still-lifes and oscillators, a *chaotic* pattern. A well known example is the r-pentomino, shown in figure 2.4, consisting of 5 cells, which takes 1 103 generations to stabilize in a 116 cell configuration of still lifes, oscillators and gliders (see figure 2.4). The fewer cells a pattern (or one of its predecessors) has, the more likely it is to turn up in the evolution of a given starting configuration.



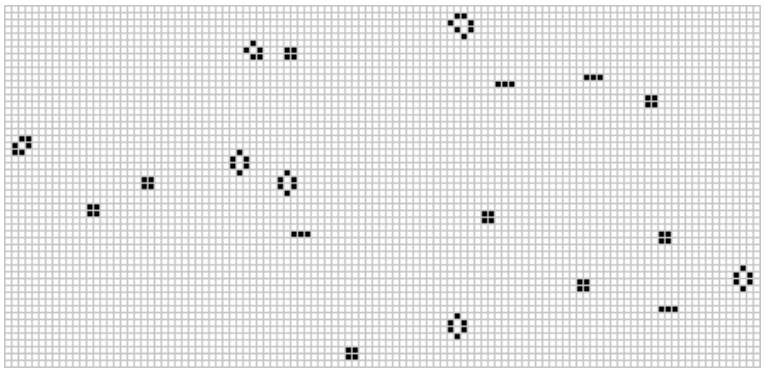


Figure 2.4: The r-pentomino and its final 116 cell evolution after 1 103 generations (excluding 6 escaping gliders)

Conway set out to prove that a simple cellular automaton could exhibit a number of interesting behaviors, for example: is infinite growth possible, does a self-replicating pattern exist, and can a Turing machine be constructed?

A (semi) oscillating pattern that produces gliders would demonstrate infinite growth. Bill Gosper built the first such pattern, consisting of two interacting queen bee shuttles, that produces a glider every 30 generations: a (period) p30 glider gun known as the Gosper gun. Gosper constructed the pattern by hand without the benefit of a computer program to validate its correctness. It constructs an endless stream of gliders moving away from the gun, demonstrating infinite growth [24].

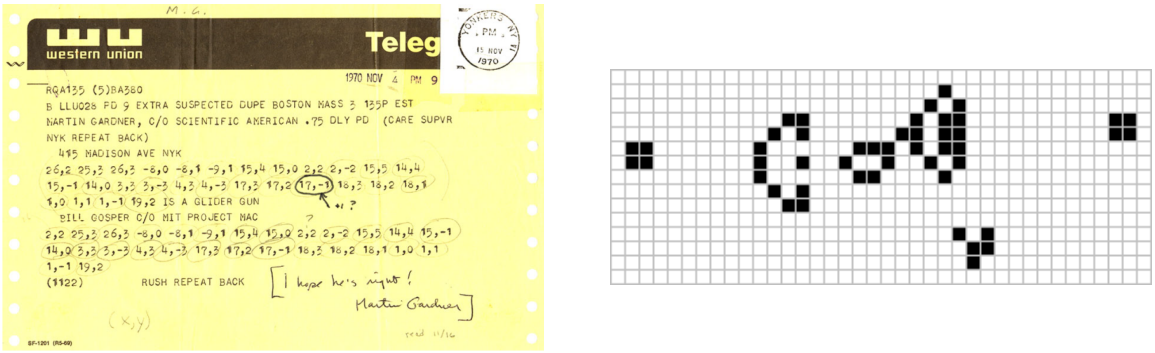


Figure 2.5: Gosper's seminal telegram to Garner containing the recipe for the Gosper gun.

Computer programs to animate Life patterns were soon developed by several authors independently, and this allowed for easier verification and inspection of pattern evolution. With the help of such a program [36] Gosper's Life team at MIT developed a flotilla of ships that lays down a track of glider guns, thus demonstrating quadratic growth, the fastest growth rate possible on Life's 2D lattice (see figure 2.6).

Existential questions Conway et al. published the outline of a proof for his holy grail, a self-replicating pattern, in 1982, but stopped short of construction [4]. A working example was finally built by Dave Green in 2013 [33].

Currently a number of outstanding questions remain, examples of which are:

- Is Life omni-periodic? Oscillators for all periods, except 19, 23, 38, and 41, have been found or have been proven to exist [8].
- What is the smallest pattern for which no ancestor exists (Garden of Eden)? All patterns fitting inside a 6×6 bounding box are known to have ancestors. The smallest known Garden of Eden fits inside a 12×8 bounding box [17].
- The fastest speed at which signals can be transmitted is one cell per generation, known as the 'speed of light'. Is there a pattern that can transmit signals at light speed between arbitrary points in space? Currently, only light speed wires that transmit signals in a straight line are known. A light speed wire that can turn a corner might prove omni-periodicity in Life.

Figure 2.6: The Breeder consists of a flotilla of spaceships, leaving behind streams of gliders, which construct a row of Gosper guns, thus demonstrating quadratic growth.

Initial discoveries came from evolving small starting patterns by hand. With the advent of computer programs to evolve patterns a few months later, random starting configurations were fed into a program and evolved for a number of generations. In order to determine existential questions, such as: *"is infinite growth possible?"*, a number of patterns were discovered using a combination of automated search and manual construction. Nearly 50 years later this is still the prevailing method of discovery.

Most progress on these questions after the initial years has been made using search programs. Since the number of candidate patterns increase as 2^n with the number of cells under investigation, this quickly renders brute-force infeasible. Smarter approaches exist, but their runtime degrades as the number of generations under investigation increases. Section 2.3 describes search approaches in more detail.

In order to answer outstanding questions, new building blocks are needed. Because the search space is huge, these building blocks can only be found using search programs.

2.2 Evolution of patterns

Because of Life's simple rules, programming a simulator for the game is straightforward. The infinite lattice is the only requirement that presents a problem. This is dealt with by either having a hard boundary at the edges or by running the simulation on a toroidal grid (so that moving patterns leaving the grid on one side enter the grid on the opposite side).

Two basic algorithms for pattern evolution exist: counting of neighbor cells, which has a running time of $\Theta(n)$ (given n cells) and Bill Gosper's 1984 HashLife [26], a memoization algorithm that caches previously encountered patterns, which runs in $\Omega(log(n))$ time in the best case. HashLife breaks down for patterns with an irregular evolution, because previously memoized patterns do not recur, resulting in $O(2^n)$ run time.

StreamLife [28] is a modification of HashLife written by Adam Goucher in 2018, designed to fix a deficiency in HashLife when working with self-replicating machines. HashLife does not recognize that returning streams of gliders do not affect the predictability of a pattern. This forces it to do redundant memoization. StreamLife splits subpatterns into demonstrably independent parts, which can be cached separately. It is inspired by Miéville's sci-fi novel, *'The City & The City'* [55].

Writing a truly efficient $\Theta(n)$ evolution algorithm for Life is complex, and as far as we know, there are no publications on this subject. In our research we use a novel $O(n)$ evolution algorithm,

LifeCube, suggested by Michael Simkin [65], which outperforms the next best algorithm—Tom Rokicki's QLife [70]—by a hundred-fold on a CPU. (see table 3.1).

Gibson et al. [25], and many others, have implemented a Life evolution algorithm that runs on a GPU. Unfortunately, they take as the starting point a naïve neighbor counting method and parallelize this, resulting in a 1 000× speedup. This looks impressive, until one realizes that Tom's QLife is 10 000× faster and LifeCube 1 000 000× faster than naïve counting on a CPU. On the GPU, LifeCube runs a 1 000× faster still (see section 3.2). This is possible because evolving Life patterns is an embarrassingly parallel problem, but unless the parallel implementation is derived from an optimized starting point, many orders of magnitude speedup are wastefully discarded.

2.3 Searching for patterns

Conway's team started 'playing' the game by hand using cheap flat black and white stones on a Go board. But things quickly got out of hand when Martin Garner published his first Life column in 1970, offering a $50 reward for the discovery of the first life pattern to exhibit infinite growth [23]. Guy and Bourne wrote the first program to animate Life patterns in Algol in 1970[2].

David Eppstein maintains a list of search algorithms [19] that are known to have found novel Life patterns. Current search methods can be divided into the following categories:

- Brute-force;
- Binary tree based;
- Random sampling, including genetic algorithms;
- Manual or machine-aided construction of complex patterns using known sub-patterns as building blocks;
- Searching using Sat solvers.

Many search programs for Life can be found on the internet, although few papers have been published on search in Life. This is unfortunate, because the broader search community might well benefit from the insights incorporated in these programs.

2.3.1 Brute-force

The easiest method to program is a brute-force search of either a bounded region of space, patterns with a limited number of live cells, or patterns made up of a number of smaller building blocks. The search could be exhaustive, or sample a random subspace. Most of the earliest discoveries for patterns that do not naturally occur have been made using this method.

An example of combining known objects is Paul Chapman's Glue 2 project [9] which takes a stationary object and then bombards this with a *slow salvo* of two gliders (i.e., glider #2 arrives after the activity from the first collision has settled down). Slow salvo's are important because they are a key ingredient in universal constructors which are used in self-replicating patterns. The output is added to a database that uses the initial positions of the starting patterns as its key.

Searching by brute-force quickly breaks down because the search takes $O(2^{xyp})$ time for a pattern with period p and a bounding box of x by y cells.

[2] A copy of the original program can be found at `https://rosettacode.org/wiki/Conway%27s_Game_of_Life/ALGOL_68`

2.3.2 Binary tree based search

Dean Hickerson was the first to break new ground and move away from brute force with his LS (Life search) program in 1989. Written in 6502 assembler, it was later ported to C by David Bell who renamed it LifeSrc [3]. Many (now) famous oscillators and non-naturally occurring spaceships where found by either LS or LifeSrc. LS assigns a fixed rectangle to each generation of a pattern. For each cell therein it stores the states `alive`, `dead`, or `unknown`. It then tries to fix the state of unknown cells by examining neighboring cells in the previous and next generations. If it cannot determine the state, it tries a depth-first search in which it tries both dead and alive states for that cell. LifeSrc works nearly the same, but David added a fourth `don't care` state to prevent the program from trying to fix unimportant border cells. To facilitate moving patterns, an offset is added between two generations. David Bell wrote a 6-part article on the spaceships he found using LifeSrc on the Usenet newsgroup `comp.theory.cell-automata` during the period August-October 1992, supplemented with a 1996 addendum [2].

In article #2 Bell explains the program's workings, stating: *"During a search, the program recursively attempts to set cells ON or OFF, and for each cell that is set uses transition and implication rules in order to detect contradictions in the current state of cells. This allows the program to quickly stop and backtrack over impossible situations, and thus reduce enormously the size of the search"*. A more detailed description of its inner workings can be found in the README file accompanying the source code for LifeSrc [2].

Dean's LS program is especially notable, because it was the first program to find new spaceships after a 20 year wait, highlighting the dire need to move on from brute-force.

David Eppstein developed an algorithm, called gfind, that searches for oscillating patterns [18] using De Bruijn graphs [49] to eliminate sub-patterns that cannot have repeating cycles in the given period.

De Bruijn graphs A De Bruijn[3] graph is a directed graph that maps overlaps between sequences of symbols. An n-dimensional graph for m symbols has $\leq m^n$ vertices. The Hamiltonian cycles of a De Bruijn graph are called De Bruijn sequences. These sequences can be defined as: given an alphabet A with k distinct characters, find a string S over A such that every possible substring thereof with fixed length n appears exactly once in S. Such a string S is called a De Bruijn sequence. The De Bruijn graph $Bg(A, k, n)$ is a tool for finding all possible De Bruijn sequences $Bs(A, k, n)$.

De Bruijn diagrams are extensively used in genetics, where DNA analysis finds many fragments from which the whole genome must be reconstructed. The De Bruijn sequences that come out of the graph are used to piece together the full genome.

gfind Instead of using LifeSrc's cell-by-cell approach, Eppstein's gfind groups rows of cells from all generations of the pattern in a single sequence. Because the program looks for oscillators, it tries to make the sequence self-repeating. The program compensates for moving patterns by offsetting the grid by $\{k/p \mid \gcd(k, p) = 1\}$ (k = shift in cells, p = number of generations). Because of the offsetting, a row i for each generation is located at a distinct y coordinate. The program takes three rows at $r[i-p+k], r[i-p], r[i]$ which combine to a three pixel high strip inside the pattern and uses the rules of the game to calculate the middle strip in the next generation. This middle strip is completely defined by its three ancestor strips, assuming a little padding with empty cells on the sides. It uses a transposition table to detect equivalent states (two states are deemed equivalent if their last $2p$ rows are the same). A De Bruijn graph is then created by forming a vertex for every equivalent class of states in the search space and an edge when a state in one vertex can be extended to form a state in the other vertex. The

[3]Many people spell 'de Bruijn'; this is incorrect. The infix of a Dutch surname only loses its capital if it is, in fact, in the middle ('Nicolaas Govert de Bruijn'). When used as a last name only, the infix becomes a prefix and the first prefix is always capitalized e.g.: 'I am positively (ec)static about my Van de Graaff generator'.

size of the De Bruijn graph depends on the width of the pattern and its period. The number of vertices is $2^{2^{pw}}$. This need not be its size in memory however, any vertices that cannot be reached from the starting position are never constructed.

It performs a breadth-first search (BFS) on a fixed-sized tree (2^{22} nodes). When this tree fills up, it switches to a depth-first 'deepening' to prune the search tree and free up space for more wide searching. The deepening is limited to a fixed depth δ, defined as ap, a small multiple (a) of the period (p) of the target pattern. When the maximum level is reached then the deepening stops and moves on to a new BFS node, if it finds a dead end instead then the BFS node is removed. The search terminates when a target is found. The deepening always makes some progress by removing nodes from the BFS tree. This method resembles Korf's iterative deepening (ID) [48], however, unlike ID, early termination of deepening at level δ skips large parts of the search space.

In addition to a deepening search, GFIND uses lookahead. When selecting rows $r[i]$ it also selects its neighbor $r[i+p-k]$ and checks that neither contradicts the other using a simple lookup table detailing the evolution of all 2×2 blocks of cells. Some blocks in this table do not evolve into others and this is used to flag contradictions. It is this lookup table that forms the inspiration for GRIDWALKER's 3×3 slices (see section 5.1).

2.3.3 Random sampling

Random sampling, also known as 'soup search', means running random starting configurations (soups) and tabulating the resulting patterns. It is the approach Conway used when first exploring Life on the PDP-7 machine at Cambridge.

Adam Goucher uses a distributed computing approach to generate random 16×16 grids (2^{256} possible patterns) and track their final evolution in his AGPSEARCH program. He maintains a growing online census containing over 200 trillion objects to date [27].

It is preceded by Nathaniel Johnston's Online Life-Like CA Soup Search (TOLLCASS) [44] which ran 6 412 048 029 soups of 20×20 grids between 2009 and 2011. Unfortunately, the data was lost in a server crash and only a cached capture of the results website at the Internet Archive remains[4].

2.3.4 Genetic algorithms

Little work has been done on using evolutionary algorithms for searching in Life-like CA's. Examples are: Karafyllidis [45] who tweaks a genetic algorithm tuned to modeling the spread of forest fires to find patterns in a rule closely related to Life. Gibson et al. [63] use a genetic algorithm to search for *guns* in a cellular automaton similar to Life—which happens to have multiple small guns easily found using a sampling search.

To our knowledge no genetic algorithm has successfully been used on CA's as hostile to search as Conway's Life. The reason for this is unknown. We suspect they degrade into a subset of random sampling because their fitness functions cannot discriminate between good and bad potential solutions, meaning they can only tell an exact match from no match.

However, we do believe that adding clever heuristics inspired by genetic algorithms to existing search methods is an avenue worth exploring.

[4] https://web.archive.org/web/20110510032152/http://www.conwaylife.com/soup/census.asp?rule=B3/S23&sl=1&os=1&ss=1

2.3.5 Machine-aided search using known sub-patterns

Hersrc Karel Šuhajda's HERSRC [69] finds *Hershel conduits* (see appendix A). These are used as part of a toolbox for constructing many ingenious patterns, and are key to the near proof for omni-periodicity of Life. It uses a databases of known Hershel track sub-patterns to guide a backtracking search which prunes the search tree when the pattern grows too large or when sub-patterns overlap. The latter would cause—possibly destructive—interference. The search space is reduced by an Aho-Corasick string matching automaton on a predefined list of impossible sequences.

SLMAKE Adam Goucher's SLMAKE [29] (a backtracking compiler for glider constructible Life patterns) uses a mix between a recursive tree search and big data. It takes an 800 megabyte database of slow saldo reactions (gliders crashing into stuff) from GLUE [9] successor HONEYSEARCH [29] and then assembles the parts to form recipes for large glider-constructible objects. This program was used to aid construction of the *Orthogonoid*, a self-constructing spaceship (with a bounding box of $868\,856 \times 707$), build by Dave Greene on 29 June 2017; as well as many universal constructor based spaceships like it. SLMAKE uses STREAMLIFE (a HASHLIFE variant, discussed in section 2.2) to check if a glider stream will work correctly. It basically automates the search for an input recipe of a universal constructor, creating every known glider constructible object from just two components: a simple block and one or more gliders—although the recipe can be simplified if other seeds than the block are allowed.

Searching using SAT solvers Knuth devotes a section on searching for Game of Life patterns in his 300 page section 7.2.2.2 on 'Satisfiability' [47]. In March 2018, Adam Goucher and Nick Gotts announced the discovery of the first *elementary* (not made up of smaller parts) *knight ship* (a spaceship traveling at an oblique angle) after a month of automated searching using a parallel SAT solver [30, 31].

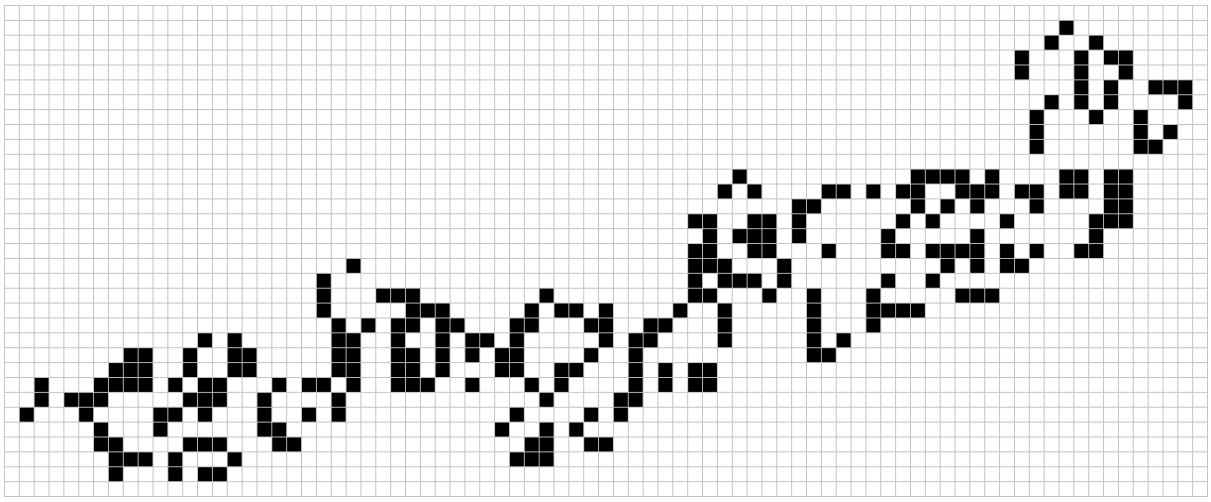


Figure 2.7: Sir Robin (*The knight who bravely ran away*), shown here traveling northeast, an elementary knightship found using a parallel SAT solver search.

In order to explain how these searches work, we will have to skip ahead a bit and delve into the inner workings of SAT solvers. Readers unfamiliar with the subject are advised to read chapter 4 on SAT solvers first and then return here. In order to encode Life into a SAT solver, two things are needed. First the target patterns needs to be encoded. Because Life is two-dimensional, we encode the variables as v_{xy}, or v_{xyt} if we wish to include targets in different generations t. Next, we need to encode the transition rule $T(v_t, v_{t+1})$. If we do not want to add additional variables, 190 new, and rather long, clauses per cell are needed—bad! It is better to exploit the fact that we can use `and`, `or`, and `not` operations to construct a

counting circuit. These three operations are inherent to the data representations of a SAT-solver, so this is *relatively* cheap. Depending on the design of the counting circuit, more or fewer additional variables and clauses are needed. Knuth's design adds 14 variables in 63 short—good!—clauses per cell to implement a binary tree; In their knightship search, Goucher and Rokicki use 9 or 8 variables[5] with the same number of clauses by using a split-radix binary-ternary tree. If needed, the pattern can be constrained to any boundary by specifying that, at the border, no 3 alive cells are allowed next to each other; this forces the outer region to be filled with dead cells.

In Knuth's (toy) example, a 7×15 frame with a fully specified final pattern takes 34 000 clauses with 9 000 variables. Drawing conclusions from this, such as, *"what is the smallest ancestor for this pattern?"*, takes a between 10 to 20 gigamems[6].

Goucher and Rokicki used Heule's CUBE AND CONQUER [42] algorithm to parallelize their search. CUBE AND CONQUER uses an—expensive—lookahead solver [41] (MARCH-CC[7]) to divide the work, the 'Cube' phase, which passes the work on to 'Conquer' using a traditional SAT solver (IGLUCOSE[8]). Using this complicated setup allows the search to be split into a number of independent units that can each run on a different thread in a multi-core CPU. The method borrows heavily from GFIND (see section 2.3.2). Goucher writes: *"Unlike in GFIND, the 'rows' are exactly perpendicular to the direction of travel, even for knightships. Specifically, we map cell (x, y, t) to lattice coordinates $(2x - y, 3y + t)$. The 'state' of the search is a 6-tuple of consecutive rows of length W"*. In order to increase the potential for parallelization, separate searches are performed for the head and the tail of the knightship. If their centers align, a match is found.

[5] Their write-up is a bit vague on the exact number of variables used, merely stating 40% less than Knuth's 14 variables.

[6] 'gigamems' is Knuth's clever, machine independent, metric for performance. A 'mem' counts as a single read or write access to memory, the most expensive common operation on a modern computer.

[7] `https://github.com/marijnheule/march-SAT-solver`

[8] `https://github.com/marijnheule/CnC`

Chapter 3: Enumerating life patterns

In this section we outline an algorithm to evolve Life patterns as fast as possible. The algorithm combines elements from a number of existing approaches in order to minimize memory accesses. This results in a 100-fold speedup over the current state of the art on a CPU. A GPU version of this code improves this another 1 000-fold. Fast enumeration of Life patterns is used in our search algorithm to validate (fragments of) search results.

3.1 Enumeration algorithm

In order to enumerate patterns we use a fast algorithm (LifeCube) based on an implementation of Knuth's exercise 7.1.3-167b [46] by Michael Simkin [65]. The algorithm works as follows:

Cells are stored as 1-bit pixels in 16×16 bitmaps called 'blocks'. Pixels at the edge of a block are compared with their counterparts in a neighboring block. The bitmap's state in generation t is derived from generation $t - 1$. All even bitmaps are stored in `P` and all odd bitmaps in `Q`; thus any `P` is derived from the previous `Q` and vice versa.

In order to reduce memory accesses, we borrow an idea from Alan Hensel's JavaLife [38]. We stagger the bitmaps of even and odd generations (figures 3.1 and 3.2); specifically, we offset the coordinates of bitmaps by 1 pixel to the northwest in odd or to the southeast in even generations. Any block is thus completely contained by 4 blocks of the previous generation; instead of having to access 9 neighboring blocks we only have to take 4 neighbors into account, reducing expensive memory accesses and optimizing the CPU operations needed for neighbor blocks.

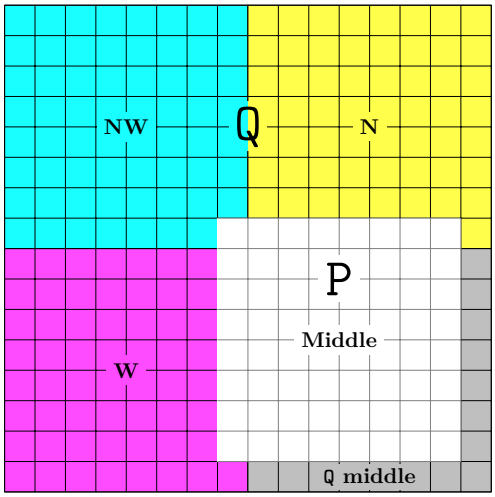


Figure 3.1: Using stagger stepping, the block is fully contained by 4 neighboring ancestor blocks.

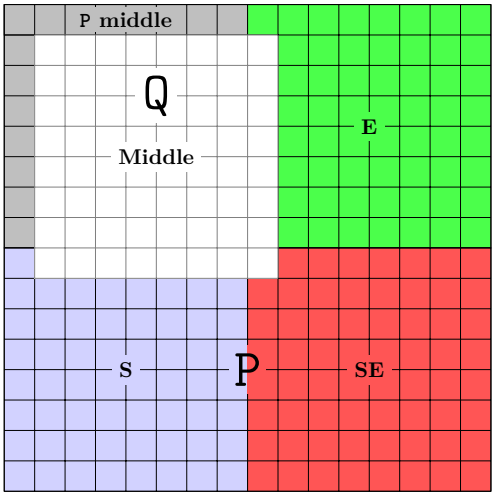


Figure 3.2: The x/y offset is reversed in odd generations.

When calculating `P` from `Q` the algorithm creates 8 neighbor bitmaps out of `P_S`, `P_E`, `P_SE`, and `P_Middle` by offsetting the region marked by `Q_Middle` (see figure 3.1) by 1 pixel in each of the directions [N, NE, E, SE, S, SW, W, NW], borrowing bits from neighboring blocks as needed. This is done using the instructions `shift` (to cut) and `or` (to paste). Note that the white box labeled 'P middle' in figure 3.1 corresponds to the gray area marked 'P middle' in figure 3.2 and vice versa for 'Q middle'.

It then uses the boolean operators `and`, `or`, and `xor` to simulate a saturated half-adder circuit. This uses the well known formula for radix-2 addition: $x + y = (x \oplus y) + ((x \wedge y) \ll 1)$. Thus adding neighborhood counts of the 8 planes into 3 destination planes *(plane 1, plane 2, and plane 4)* with 3 bits of neighborhood counts. Because a plane has only one bit per pixel, the left shift ($\ll 1$) is replaced with promotion to the next plane. Plane 1 holds the least significant bit, plane 4 holds the most significant bit, and plane 2 holds the middle bit (see figure 3.3). This allows us to count 0 to 7 neighborhood bits. Both 4 and 8 neighbors result in death under Conway's rules; we abuse this fact to speed up the calculation by merging the count for 4 and 8 neighbors in plane 4; this allows us to use three planes instead of four for the neighborhood counts at a slight loss of generality. Next, we combine the starting bitmap 'middle' with the count planes, according to Conway's rules, to get the next generation. Listing 3.1 shows pseudo code for the algorithm.

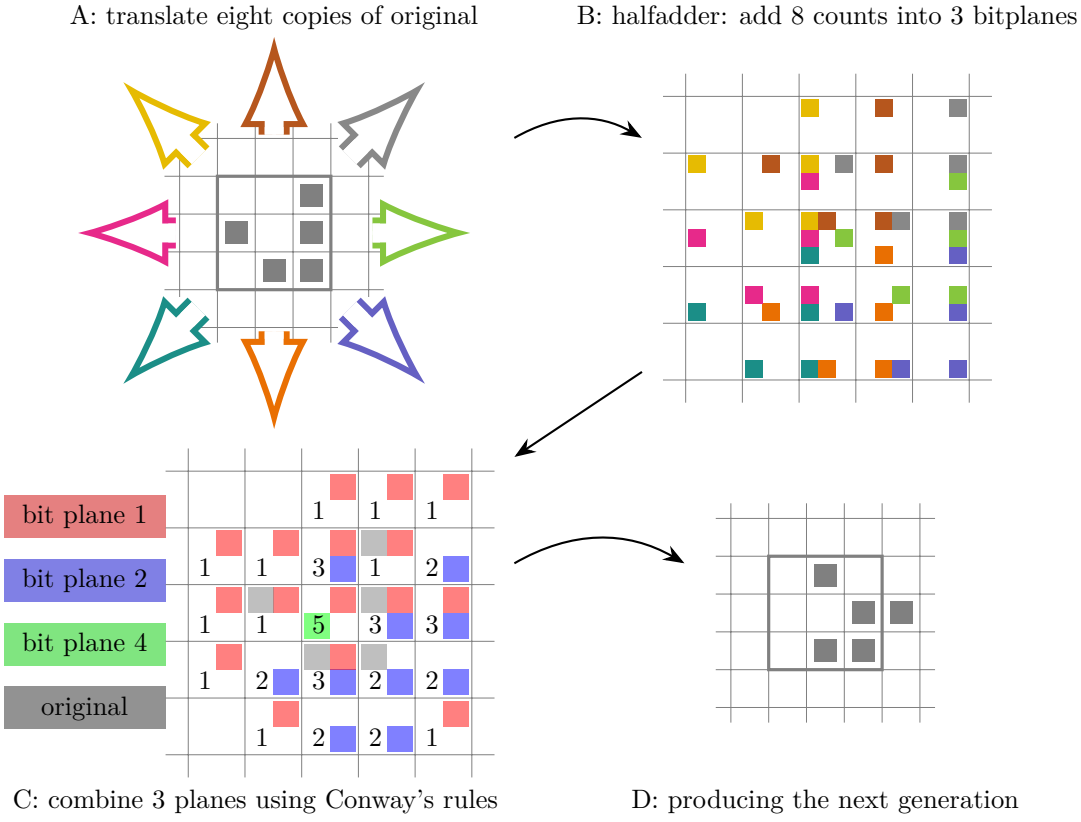


Figure 3.3: LifeCube combines eight neighborhood planes into 3 bitplanes containing the neighborhood counts.

The LifeCube algorithm is fast (20 clock cycles to process 256 neighbor pixels), because it does not need to access a lookup table, it calculates many bits in parallel, and all manipulations take place inside CPU registers. It is 100× faster than the fastest known algorithm using a lookup table (see table 3.1). An additional benefit of storing the data in bitmaps is that it is cheap to compare differences between generations; a simple `difference = (pattern_before xor pattern_after)` suffices.

Listing 3.1: The LifeCube algorithm

```
original = bitmap_of_pattern
for i in (S, SE, E, NE, N, NW, W, SW) do //shift in 8 directions
    neighbors[i] = shift(original, direction[i])
//use a saturated half-adder
plane1 = xor(neighbors)
intermediate_bitplanes2 = and(pairs of neighbors)
plane2 = xor(and(pairs of neighbors))
plane3 = plane1 and plane2
plane4 = and(intermediate_bitplanes2)
//the neighborhood count of every cell is divided across 3 bitplanes.
//bitplane_1 contains the LSB, bitplane_4 contains the MSB.
newbitmap = (original and plane2) or (plane3) and not (plane4) //rules of Life
difference = original xor newbitmap
```

We use intrinsic operations built into the processor like: `popcnt` *(population count)* and `tzcnt/lzcnt` *(trailing/leading zero count)* to further speed up processing of the classification steps. Because the data is already in the CPU registers this is quite efficient.

Loop optimization Blocks are stored in a 64-ary tree. Every node in the tree holds an 8×8 array. In order to keep track of active nodes/blocks a 1-bit-per-block `set` 'ActiveList' is used. The program moves through each node in this tree using optimized code.

Listing 3.2: Keeping track of blocks to process

```
i = ActiveList.First
while i < MaxBlocks do
    BorderStatus = GeneratePtoQ(p[i], p[i].North, p[i].West, p[i].NorthWest)
    Additions.Add(BorderStatus)
    Subtractions.Add(BorderStatus)
    i = ActiveList.NextActive(i)
ActiveList.AddSubtract(Additions, Subtractions)
```

The code inside the while loop in listing 3.2 is implemented in assembly as a flat sequence of instructions with no (conditional) jumps. For example the `ActiveList.NextActive` call is implemented as shown in listing 3.3.

The code in listing 3.3 has a reciprocal throughput[1] of 3 CPU cycles per iteration [21]. More importantly, it needs to run only once per iteration of the `while` loop because of the intrinsic `tzcnt` instruction (which would otherwise have to be implemented using a loop). The other lines in listing 3.2 are similarly stripped of jumps using conditional move (`cmov`) instructions and other tricks which allow us to intersperse the body of the `while` loop inside the generation code from listing 3.1.

The generation code uses AVX instructions which are executed using dedicated circuitry [20] of the CPU; the while loop runs normal integer code which is executed in the—otherwise idle—integer pipeline. By intermixing these two execution streams we completely eliminate the loop overhead. This cuts the execution time in half.

[1] The latency does not matter because the dependency chain is broken up by other instructions.

Listing 3.3: Find the next block to process

```
1  function ActiveList.Next(previous_block_index: int): int;
2      //rcx = self = pointer to bitset
3      //edx = previous_block_index
4      //eax = result
5      mov rax,[rcx]            //bitset := self
6      lea ecx,[edx+1]          //start at previous_block_index + 1
7      shr rax,cl               //bitset := bitset >> bits_to_skip
8      tzcnt rax,rax            //count empty positions to skip
9      add eax,ecx              //return ((prev + 1) + empty_positions_found)
```

Memory management of an unbounded universe Life patterns tend to cluster in geographically dispersed blobs. This blobbiness is mirrored in the way memory is organized. It makes no sense to allocate memory for parts of the universe that do not contain live cells. In order to keep memory management simple, active regions are allocated in *super blocks*: a square array containing $(8 \times 8) = 64$ blocks. Taking into account that blocks store 16×16 pixels, the total surface area of a super block is thus $(8^2 \times 16^2) = 16\,384$ pixels; a pixel uses one bit, therefore its memory footprint is 2KiB.

Super blocks are also stored in a 64-ary tree. This tree partitions the 2D lattice of the universe in a recursive 8 by 8 grid, like a traditional quadtree overlays a 2 by 2 grid on a 2D space. As the live region of the universe outgrows the tree, additional levels are added. The leafs of the 64-ary tree contain an 8×8 array with pointers to super blocks, every node (leaf as well as non-leaf) of the tree contains an `ActiveList`, an `AliveList`, and an `ActiveBorderList`, detailing the status of the nodes or blocks underneath. The ActiveList is a bitset containing 64 bits, every `on` bit denotes an active (recently changed) super block. The AliveList is the same structure, but every `on` bit denotes a super block with at least one live cell. The ActiveBorderList uses a single bit to track activity at the each of the borders of $2 \times (8 + 8 + 1)$ blocks: the north/south 8, the east/west 8, and redundantly, a single pixel in the northwest/southeast corner. Because of the stagger stepping used, it is not necessary to track the northeast and southwest borders as these borders fall wholly inside the block's own `P`/`Q` block.

If a super block contains no live cells and the borders of its neighbors contain no active cells, its pointer in the tree is replaced with nil and the record is put in a cold storage array: the *morgue*. Before any new super block is allocated the morgue is inspected. If not empty then the block is revived from there instead of being newly allocated. Because all super blocks in the morgue are dead, they do not need to be initialized, saving time. Super blocks are only deallocated if the animation is paused or if the system exceeds its maximum memory allocation. Note that in order to be declared dead, both the `P` and `Q` generation need to be zero. In order to be tagged active either `P` or `Q` needs to be different from its previous `P` or `Q` state. The algorithm thus considers a period 2 pattern to be *inactive*.

bounded universes If the pattern is limited to a tiny bounded universe, a simple 2D array of blocks is used instead of the tree. Many universes can be run in parallel in this mode, all of which use this trivial configuration.

3.2 LifeCube on a GPU

To further speed up the calculation of Life patterns, the LifeCube algorithm has been ported to a graphical processor (GPU) using the NVidia CUDA framework. A NVidia GTX 1080 was used, but care was taken to only use instructions that can run on compute capability 3.0 so that the code can also run on older hardware.

Register allocation In order to keep many concurrent threads in flight, a few changes needed to be made to the CPU version of the algorithm. The GPU does not have dedicated registers in the CPU sense, instead there is a register file of limited size (64K integer registers per streaming multiprocessor (4×32 compute cores)[2]) that is shared between all running threads. If the space needed for local variables exceeds the space in the register file, the GPU scheduler will reduce the number of threads running concurrently to compensate; this results in less than 100% utilization, wasting resources. The LifeCube algorithm uses sixteen 32-byte registers (512 bytes in total, needing 128 integer registers). To prevent overloading the register file, the size of a block is reduced from $16 \times 16 = 256$ bits to $8 \times 8 = 64$ bits (8 bytes). The optimizing compiler further reduces the local storage needed using its register allocation optimization (from 128 bytes per thread to 68 bytes per thread). This results in full utilization, compared to less than 25% without space savings.

The GTX 1080 GPU can run a maximum of 57 344 threads concurrently—there are 3 584 compute cores, every core runs 16 threads in lockstep in the pipeline[3]—compared to a maximum of 8 concurrent threads for an Intel i7 CPU. It can thus run the core calculation much faster, even if it only does a quarter of the work per thread and runs at a lower frequency.

The problem now shifts to the allocation of work units to the GPU threads. Threads need to know which block to process. For maximum performance the GPU should only work on 'dirty' blocks (i.e. blocks that have been altered in the previous generation or whose neighboring blocks are active).

Work list In the CPU code an 'ActiveList' (see listing 3.2) is used, in the GPU variant a parallel version thereof is needed. It no longer suffices to hand out work units one-at-a-time, they have to be calculated concurrently. Luckily, there is a $O(N)$ method to calculate the ActiveList. Intel CPUs have the intrinsic PDEP (parallel bit deposit) function which can be ported to the GPU in a few lines of code (see listing 3.4). This code takes a 64-bit set (work list) where every *set* bit denotes an item to work on and every *unset* bit an item to skip. After processing, a list of 65 items is generated. The first entry is the number of items to work on; the remaining items are the indexes of the work items in order. Items that do not need to be processed are denoted by -1 (meaning: 'do not process') entries at the back of the list. This takes $O(N)$ time.

[2] a compute core can execute 16 threads in lockstep in the warp pipeline, hence $(4 \times 16 \times 32)$ 2 048 threads running in a warp share a single register file.

[3] technically the maximum is 2 048 threads $\times$ 28 streaming multiprocessors which works out to the same number.

If we know the GPU has an intrinsic population count function we can use the following simple code:

Listing 3.4: Get the index of a work item using built-in popcount

```
1  void find_nth_set_bit (uint64_t bitset, int n) {
2      if (( bitset & (1 << n) ) != 0) {
3          local_work_list[populationCount(bitset & (-1 >> (64-n) )] = n;
4      }
5  }
```

Clean up A clean up sweep then processes all the lists and puts all items in one long work list to be worked through by the pattern calculation routine. The array of local work lists is sorted using radix sort; because there are only 64 different keys we can treat the key length w as a small constant, hence the $O(Nw)$ running time of radix-sort reduces to $O(N)$. The work lists are then handed to different CUDA warps according to the number of work items in a list. Warps that process work lists with many work items need to work harder and are thus assigned fewer lists, warps that process near empty work lists get assigned more lists per warp. In this way all warps are kept fully engaged. The individual work lists are concatenated into one long list. Because the number of items per work list and per warp are known beforehand, all items can be added to the long list concurrently, without the need for synchronization. Every item in the long list contains: the address of a work item, the addresses of their N, W, and NW neighbors for `P`→`Q` generation (or S, E, and SE neighbors for `Q`→`P`), plus the addresses of four local active bit sets to update, one set for the new pattern block and three sets for its neighbors as per figures 3.1 and 3.2. This means the cleanup takes $O(N)$ time.

Calculation Finally, the calculation routine calculates the next generation of a pattern and updates the active bit sets of any block and its neighbors as needed. Racing conditions do not occur in the pattern generation: reading the current pattern is a read-only operation; and when writing the next generation, every thread only writes to the block it has been allocated. Updates to bit sets are performed using atomic boolean operations so that racing conditions do not occur there either. Thus every step takes $O(N)$ time.

To recap, the GPU version of LifeCube runs in triple jump fashion:

skip Convert the bit set of active items into a list of local work items.

step Collate all local work lists into one global work list that details the addresses of work items and their neighbors.

jump Work through the global work list, process `P`→`Q`, and update the given active bit sets.

3.2.1 Limitations of the GPU algorithm

In order to work efficiently, many thousands of threads in GPU need to be kept occupied. However, the LifeCube algorithm can only work on one generation of a given pattern at a time. Unless the number of active cells in a pattern is huge, not all available concurrent threads can be kept occupied and efficiency will suffer. For this reason the GPU code will not perform faster than the CPU code when processing small universes.

If we want to process small patterns, we can run many of them in parallel. If each pattern is run in a tiny bounded universe, the time-consuming bookkeeping steps (skip and step) can be dispensed with

since—nearly—all slices will be active and bookkeeping is only needed to prevent inactive slices from being processed.

If a bounded pattern smaller than 8×8 pixels is calculated then processing can be further sped up by not taking neighbor blocks into account.

No temporal concurrency The GPU version is much faster than its CPU-based cousin. However, the algorithm used can only parallelize spatial processing. Temporal dimensions can not be computed concurrently. Unlike Gosper's HashLife, every generation must be calculated in sequence. This limits the usefulness of the GPU implementation for most practical patterns; no matter how fast the GPU can run the pattern, HashLife will always outperform it, because the first is an $O(N)$ algorithm and HashLife takes $\Omega(log(N))$. The latter skips increasing powers of 2 for all parts of the pattern whose evolution it has fully memoized, whereas LifeCube is forced to calculate every single generation.

GPU LifeCube shines in computing many bounded universes in parallel, an application ill suited to HashLife's obsessive compulsive need for 'predictability' of a pattern.

Table 3.1: Comparison of optimized $O(N)$ Life enumeration algorithms

Algorithm	**method**	**cells/gen/sec**	**speedup**[4]
Naïve counting	count neighbor cells for every pixel	10^4	
JavaLife [38]	lookup table + linked list	$3 \cdot 10^7$	3 000 ×
QLife [70]	lookup table + quad tree	10^8	3 ×
LifeCube [65]	halfadder logic	$2 \cdot 10^9$	20 ×
Optimized LifeCube	low level assembly + stagger step + 64-ary tree	10^{10}	5 ×
GPU LifeCube	CUDA code running on a GPU	10^{13}	1 000 ×

[4]Speed up relative to the previous line item

Chapter 4: SAT-solvers

In order to have some background into solving the satisfiability problem (SAT), a short diversion into the inner workings of modern solvers for this problem is needed. Our search engine for Life patterns, as described in chapter 5, uses a unique grid-based adaptation of the DPLL algorithm for SAT solving. We use several SAT solvers coupled with the LOGIC LIFE SEARCH [11] front-end—a search engine for Life patterns—to benchmark our algorithm and validate its results. This chapter first describes the greedy conflict-driven clause learning algorithm used by MINISAT [68], on which many solvers are based. It then discusses thrifty LOOKAHEAD solvers, which are used in the CUBE AND CONQUER algorithm [42] to parallelize large SAT problems. Then, we highlight same of the problems encountered in parallelization of SAT, closing the chapter with a short summary.

4.1 Introduction

Satisfiability is one of the fundamental problems in computer science. It has been successively classified as interesting, irrelevant, trivial, and intractable before earning its place as one of the central building blocks of AI.

SAT is a member of the NP-complete class of problems. As such, it currently exhibits exponential asymptotic running time. Solvers for SAT are used to find solutions in logics, automated formal reasoning [7], program analysis [34], and many other combinatorial problems. To the chagrin of mathematicians, it has also been used to construct proofs for a number of previously intractable conjectures, most famously the 200 terabyte proof for the Boolean Pythagorean triples problem by Heule et al. [40].

The boolean formula (also: instance or problem) to be satisfied is typically encoded in conjunctive normal form (CNF): a conjunction $\wedge$ (`and` statement) containing one or more disjunctions $\vee$ (clauses), which are `or` statements of literals (a boolean variable a or its negation $\neg a$). Being boolean, variables can either hold the value true (1) or false (0).

SAT tries to find an assignment for all variables in a given formula such that it evaluates to true. If so, the problem is `SAT`, if none can be found, it is unsatisfiable (`UNSAT`). The benefit of CNF encoding is that the outer `and` statement can only return `SAT` if every clause therein evaluates to true. Thus, in order to disprove `SAT` it suffices to have a single clause return false; this representation also facilitates a number of interesting deductions.

Every statement in propositional logic can be transformed into CNF in $O(n^2)$ time, albeit with a possible linear increase in size.

Although a SAT problem with n variables has a worst case running time of $O(2^n)$, solvers are often able to produce answers much faster due to two factors. First, negations can cause contradictions: clauses that always evaluate to false, (e.g., $x = a \wedge \neg a$). Any subsets of the problem that depend on such a contradiction can be excluded from the search for a satisfying assignment. Secondly, many real world problems exhibit an underlying structure. This is another way of saying that the encoding of the problem into variables and clauses is redundant to some degree. Working with a structured (redundant) problem N with n variables only has a worst case running time $O(2^r)$ related to its non-redundant rendition R with r variables.

4.2 History

The earliest known writings on propositional logic, which underlies SAT, come from the ancient Greek philosopher Socrates. Its origins are possibly much older.

Up to the middle of the 20th century, logicians referred to satisfiability as a toy problem known as 'Tautology'; believing it to be trivial because it can always be solved in finite time by looking at its truth table, conveniently disregarding that this truth table grows exponentially with the number of variables.

4.2.1 SAT algorithms

In the late 19th and early 20th century, many people derived interesting approaches to solving SAT, many of which were forgotten, only to be rediscovered years later. A breakthrough came in 1962 when Davis et al. redefined SAT solving as probing a binary tree [14] by speculative assignment. They combined this with elimination of clauses to prune the search tree, resulting in the famous DPLL algorithm [13].

In 1971, Stephan Cook proved that any problem in NP-complete can be restated as 3SAT (a rendition of SAT where every clause has at most 3 literals) [10]. For a fleeting year, computer scientists rejoiced with dreams of P = NP, because 3SAT seemed trivially easy to solve by generating a truth table for every possible assignment; until they realized that this table needs 2^n storage for n variables.

The next major breakthrough came in 1995 when João Marques-Silva added non-chronological backtracking and conflict analysis to the DPLL algorithm [53], leading to conflict-driven clause learning (CDCL). In 2001 Moskewicz et al. building on an idea by Chu Min Li [50], added a number of innovations: firstly, a new heuristic called 'variable state independent decreasing sum' (VSIDS) which attaches probabilities to variable assignments; next, a restriction to only watch two literals per clause, and lastly, restarts, which throw away most progress in a search, except for learned clauses, then restart the search for a solution from a new beginning. Restarts allow the search to free itself from a local optimum [58]. Their CHAFF solver operates 1 to 2 orders of magnitude faster than previous ones.

These developments caused a revitalization of international SAT contests, which have been held every year since 2002. Niklas Sörensson and Niklas Eén won the 2005 International SAT competition with a 600 line C program called MINISAT [68]. Despite having gained some weight in the interim, simple and efficient MINISAT is widely used in research.

In 2005 Heule et al. introduced CUBE AND CONQUER [42], a parallel paradigm based on Gustafson-Barsis's law (*any large enough problem can be sped up by parallelization*) [35]. It splits the search in a delegation step, run by a LOOKAHEAD solver and a work step, run by many greedy CDCL-solvers in parallel.

4.3 The workings of a SAT solver

MINISAT and its ilk use a greedy strategy which can solve industrial problems with millions of clauses, apparently defying the NP-complete nature of SAT. The extent to which a given problem is easily solvable relates to the amount of underlying order in the problem, a concept closely related to a metric called 'community structure' [59], itself a proxy for the N-R ratio mentioned in the last paragraph of section 4.1. Unfortunately, measuring community structure for a given SAT instance is itself an NP-complete task.

If Life exhibits community structure to a significant degree, this can be exploited to solve outstanding Life questions using SAT solver derived techniques.

4.3.1 CNF encoding

Before we can determine the satisfiability of a problem, it needs to be encoded into conjunctive normal form (CNF). A CNF formula takes the form:

$$\bigwedge_{j=1}^{m} \left(\bigvee_{i=1}^{n} l_{i,j} \right) \tag{4.1}$$

Where $l_{i,j}$ is a *literal* (either $v_{i,j}$ or its negation $\overline{v_{i,j}}$) and $v_{i,j}$ is a *variable* which can be either true or false. $\bigvee_{i=1}^{n}$ is a *clause* containing n literals, and $\bigwedge_{j=1}^{m}$ is a SAT *instance* containing m clauses. Because an instance can have many variables, v_i is usually denoted simply by its number i.
An example is the following formula; the clauses are reused in figures 4.2 and 4.3 and labeled A–D.

$$f = \bigwedge \begin{cases} v1 \vee \overline{v2} \\ v2 \vee v3 \\ \overline{v1} \vee \overline{v3} \\ \overline{v1} \vee \overline{v2} \vee v3 \end{cases} \quad \text{or more succinctly} \quad \begin{matrix} 1, \bar{2} & \text{A} \\ 2, 3 & \text{B} \\ \bar{1}, \bar{3} & \text{C} \\ \bar{1}, \bar{2}, 3 & \text{D} \end{matrix} \tag{4.2}$$

Whilst solving, clauses fall into three classes: *unresolved* (no literals are true), *satisfied* (one or more literals are true), and *unsatisfied* (all literals are false). In the latter case the entire instance is unsatisfiable. A literal in a clause that has not been assigned a value is called a *free literal*.

Transforming first order logic into CNF In order to coax a formula in first order logic into its CNF equivalent we must first transform it into negation normal form (NNF). This means we transform all existential ($\exists x$), for all ($\forall y$), and implication ($x \implies y$) relations to the simpler boolean operations ($\wedge, \vee, \neg$). After transforming $\forall$ relations to $\exists$ using the equivalence $\forall x f(x) \equiv \neg\exists\neg x f(x)$ we can use Skolemization [67] to transform the latter to boolean formulas. The resulting NNF formula will not be equivalent to the original, but equisatisfiable—*it is satisfiable if and only if the original formula is satisfiable*—which is sufficient for the purposes of SAT.

The next step is a transformation from NNF into CNF. A naïve translation using De Morgan's laws and distribution would result in an exponential increase in size. This can be prevented using Tseytin transformation [71], which only yields a linear increase in size and takes $O(n^2)$ time.

4.3.2 Boolean Constraint Propagation

Now that we have translated our problem into a format that a solver can work with, we are ready to start looking for a solution. The first step in this process is boolean constraint propagation (BCP), also known as unit propagation (UP)[1]. When all the literals, except one, in a clause are set to false, the remaining literal must be true if the instance is, in fact, satisfiable. That this is so, is obvious from the fact that an `OR` statement must contain at least one true literal to return true. This means that clause: $\{1, 2, 3 \mid 1 = 2 = \text{F}\}$ is akin to $\{3\}$, both are called unit clauses.

The remaining unassigned literal in a unit clause must be true and we can use this knowledge to propagate the value of its underlying variable throughout the instance. By doing so, we hope to reduce other clauses to 'unit' and thus be able to infer the remaining unassigned literal. This is comparable to a Sudoku square with one unassigned number as shown in figure 4.1.

3	2	6
5	9	
7	4	1

Figure 4.1: A unit clause in Soduko

When we have fixed the value of one literal, we can broadcast the value of the underlying variable to all clauses containing it. Most time in solvers is spent doing this.

[1] 'BCP' is the common abbreviation, and 'unit propagation' the common full form.

4.3.3 The DPLL algorithm

The Davis-Putnam-Logemann-Loveland algorithm expands on unit propagation by representing a SAT instance as a binary search tree where every variable is a node. The algorithm then walks the tree, depth-first, by randomly assigning unassigned variables and examining the results. If there is a conflict, it backtracks and tries an alternative path. This continues until a satisfying assignment is found or until all branches have been tried, in which case the instance is unsatisfiable. Crucially, DPLL is a complete SAT algorith: given enough time it will *always* find either a satisfying assignment, or proof that the instance is `UNSAT`. Any solver based on this algorithm inherits this useful property.

This algorithm turns satisfiability into a depth-first search of a binary tree. We re-use the clauses in equation (4.2) above to illustrate the concept in figure 4.2 below.

We build a binary search tree containing all possible assignments for the variables v_1, v_2, v_3. Because all clauses in a satisfiable instance must evaluate to `true`, all nodes that force a clause to `false` are considered invalid. In figure 4.2 the violating nodes are labeled with the color and the letter of the clause they invalidate.

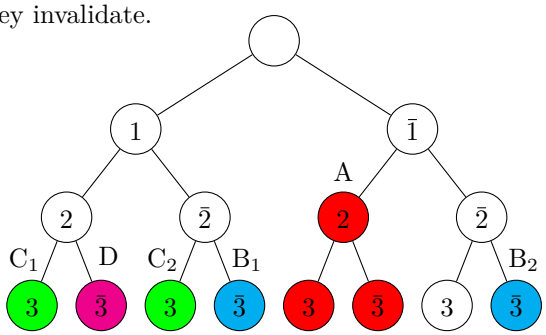


Figure 4.2: DPLL performs a depth first search of a binary tree; backtracking when an invalid assignment is found. The colored and labeled nodes conflict with clause A, B, C, or D in equation (4.2).

DPLL performs a pre-order traversal. Assuming the tree is traversed from left to right, it will first visit the left subtree $\bar{1}$. Here, it will find a conflict in every leaf and eventually backtrack to the root, before visiting the right subtree $\bar{1}$. Now something interesting happens, it will reach inner node 2 which conflicts with clause A. There is no need to go any further and thus this subtree is skipped and we backtrack to $\bar{1}$ instead. From there it visits subnodes $\bar{2}$ and 3, reaching a leaf node that provides a satisfying assignment, allowing it to return `SAT`. The remainder of the tree—the rightmost leaf node $\bar{3}$—is skipped.

The advantage of DPLL is that it skips any subtrees under conflict nodes, which means that (potentially huge) parts of the search space can be disregarded.

VSIDS There is a lot of flexibility in how the tree can be organized and how it is traversed, as can be seen in figure 4.2. If there is a single solution, which happens to be to the extreme right of the tree and one walks from left to right, it can take a long time to arrive at the solution.

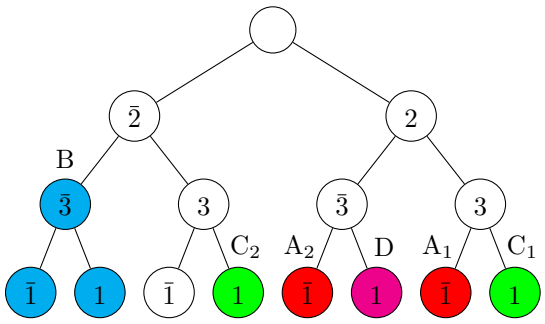


Figure 4.3: SAT solving affords a lot of flexibility in how the search is organized. This tree is equivalent to figure 4.2.

VSIDS (Variable state independent decaying sum) attempts to first resolve those variables that occur most often. It does this by updating a tally per variable every time it is encountered in a clause [53]. Tallies are periodically divided by a constant so that newly encountered parts of the tree weigh more heavily in the ranking. At every junction, VSIDS selects the branch that contains the variable with the highest score. This means that VSIDS tries to solve high scoring variables before lower scoring variables. Because high scoring variables affect more clauses in the currently active region of the tree than low ranking ones, this leads to faster clause resolution.

Two watched literals In order to keep track of unresolved clauses, some bookkeeping needs to be done. For every non-unit clause the literals that are—as yet—unresolved need to be tracked. This means keeping a large database for every unresolved literal in every clause. 'Two watched literals' [53] reduces the size of this database by only ever tracking two unresolved literals per clause. The idea behind this is that it really does not matter if a literal resolves until that assignment flips a clause to *unit*. As long as a clause has at least two unassigned literals, it has not attained that coveted status, therefore tracking three or more unassigned literals serves no purpose. Whenever one of the two watched literals becomes assigned, another unassigned literal in the same clause takes its place; if none can be found, then the clause becomes 'unit' and can be taken off the watch list.

Tracking only two unassigned literals per clause reduces the size of the database without sacrificing any accuracy.

4.3.4 Conflict-driven clause learning

In the DPLL algorithm, conflicts allow for skipping (sub-)branches of the search tree that need not to be investigated. However, one can do better by spending some time investigating the causes of the conflict and using that information to help guide the search. In 1996 Marques-Silva and Sakallah introduced conflict-driven clause learning (CDCL) in their GRASP solver [53].

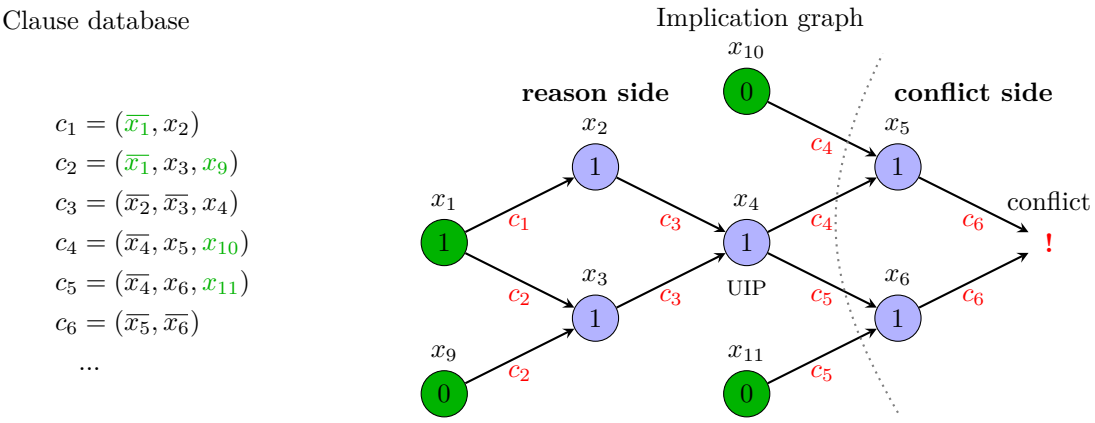


Figure 4.4: Variables (green nodes) are assigned a value speculatively. Through unit propagation, this forces decision variables (blue) to be flipped. Vertices are labeled with the unit clause (red) that caused the flip. After assigning `true` to x_1 a cascade of consequences follow. First, the unit clauses c_1 and c_2 force x_2 and x_3, which forces x_4 in c_3, leading to $x_5 = 1$ and $x_6 = 1$. These last two flips force c_6 to `false`, a conflict. When a conflict is reached, we perform a breadth-first search for the first unique implication point (UIP). All decision variables up to that point are part of the conflict side, the rest is the reason side. The nodes directly leading into the conflict side are taken as the 'cause' of the conflict. To ensure this conflict does not occur again, its negation is added as a 'derived clause': $\overline{(\overline{x_{10}} \wedge x_4 \wedge \overline{x_{11}})} \Rightarrow (x_{10} \vee \overline{x_4} \vee x_{11})$. *Adapted from lecture notes by Vijay Ganesh at the University of Waterloo* [22].

Every time a conflict is encountered, a CDCL-solver builds an implication graph (see figure 4.4). A cut is then made in the implication graph, separating it into a *conflict* and a *reason* side. The conflict side includes the conflict itself and any decision variables up to (but not including) the first unique implication point (1UIP). The reason side contains everything else. A UIP is a node in the graph that all paths in the outer parts of the implication graph must go through to reach the conflict. The first unique implication point 1UIP is the UIP closest to the conflict. There are alternative strategies, but 1UIP is generally deemed to perform best [15]. It minimizes the number of literals in newly learned clauses, aiding unit propagation and reducing the size of the clause database.

The conjunction of literals that lead to the conflict is called the 'cause' of the conflict. Because this cause is incompatible with a solution, its opposite **must** be true. Therefore, we first take the negation of

the conjunction of cause literals, then use De Morgan's rule $\overline{(x \wedge y \wedge z)} \Rightarrow (\bar{x} \vee \bar{y} \vee \bar{z})$ to transform this into a disjunction and finally add it as a newly learned clause to the SAT instance.

These new clauses are called *derived clauses* and their inclusion may lead to an exponential increase in clauses. We thus need some way to contain the growth of the instance, this is done by forgetting.

Forget Learning new clauses is useful because it prevents the search running in circles. However, if the clause database grows too large it can slow down the search. Moreover, if learned clauses do not contain any variables encountered in the current region being traversed then these clauses do not help the search. By keeping a recent activity counter for all learned clauses—akin to VSIDS—and discarding those that have not recently been used, we can remove stale data from the clause database. All derived clauses are logically implied in the original instance, so forgetting does not affect soundness.

Non-chonological backtracking The basic DPLL algorithm eliminates certain parts of the tree, but it does not try to prevent duplication of work. Referring back to figure 4.2, when a conflict is found at C_1 we can already see that the same conflict will occur at C_2 because neither of these clauses depend on the value of v_2. To prevent this, we need to build an implication graph and do a bit of analysis. We already built this graph for the clause learning of CDCL, however, here the partition into a conflict and a reason side uses a different heuristic (see figure 4.5).

Based on data gathered from the implication graph, we can backtrack non-chronologically (backjump) instead of having to retrace our steps as in normal backtracking. This redirects the search to a dissimilar part of the tree, hopefully preventing it from finding the same conflicts over and over.

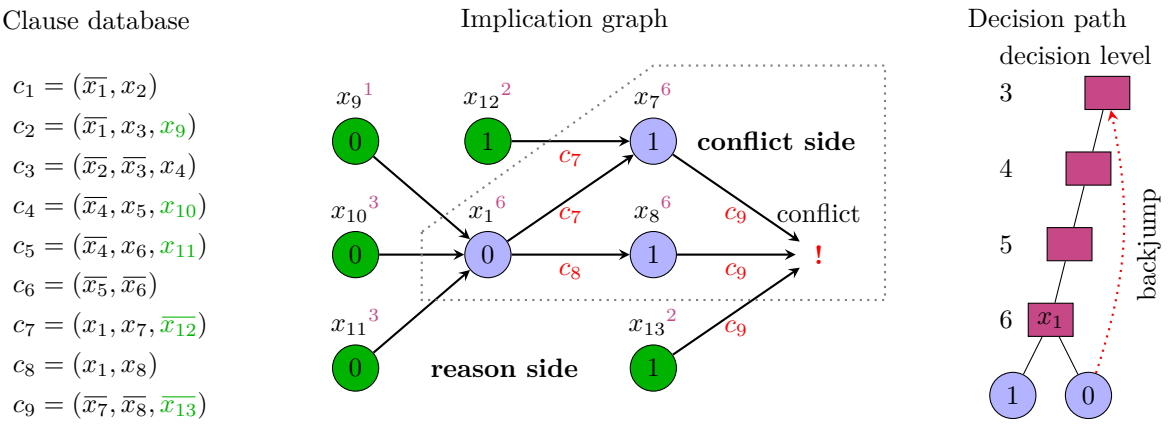


Figure 4.5: After a conflict, the solver corrals all decision variables (blue) into the conflict side. It then searches the directly adjacent assigned variables (green) and selects the newest one (e.g., x_{10} at decision level 3). Having been assigned most recently, it has the deepest decision level (purple superscripts), this prevents jumping back too far. Assignments in the current level are undone and the solver jumps to the selected level. *Adapted from lecture notes by Vijay Ganesh at the University of Waterloo* [22].

Restarts As illustrated in figures 4.2 and 4.3, there is a lot of flexibility in how the search tree is organized. In order to prevent becoming bogged down in an intractable part of the tree, it is often advantageous to clear the search and start afresh. After such a restart, the tree is rooted at a different node, leading the search in a different direction. Because we keep all derived clauses learned thus far, a large part of the knowledge gained from the previous run is retained.

Listing 4.1: Conflict-driven clause learning algorithm

```
1  input := SAT instance in CNF form
2  repeat
3      repeat
4          propagate_BCP
5          if conflict then  //analyse_conflict performs clause learning
6              if analyse_conflict = top_level_conflict then return UNSAT
7              else backjump
8          else if all_vars_assigned then return SAT
9          else decide       //VSIDS
10     until timeout_has_passed
11     forget                //but keep 'active' derived clauses
12     restart_search
13 until forever
```

At the heart of any solver, unit propagation (BCP) forces conclusions in clauses through unit propagation (line 4); the VSIDS heuristic (line 9) decides the most promising variables to investigate next.

Next, conflict-driven clause learning (CDCL), a extension of DPLL, adds new knowledge to the clause database by deriving clauses from every conflict encountered (line 6). Learning clauses prevents the same conflict from being flagged multiple times and also aids propagation. CDCL builds an implication graph, which is reused to determine the decision level for backjumping (line 7).

If bigger jumps are needed to get out of a local quagmire, we first forget learned clauses (line 11) in order to prevent clause database overflow, only keeping clauses that have seen (a lot of) recent activity. Then, the search is restarted (line 12), taking care to reroot the search at a different point.

4.4 LOOKAHEAD solvers

In this section we briefly consider LOOKAHEAD solvers. Our approach is too unlike lookahead to draw many analogues, but it is key to CUBE AND CONQUER, currently the most successful method to parallelize SAT, so it is worth explaining how this approach to SAT helps to parallelize problems.

Suppose you are an intrepid French revolutionary in an alternate timeline. You are ready to storm the Bastille, but Louis XVI has wised up and blocked the access roads. Undeterred, you decide to use the dark web of tunnels underneath Paris. Navigation in the complex labyrinth is aided by countless inscriptions left by brigands on their way to banks and jewelry shops. A simple random walk should suffice to gather enough clues to get you to your destination. Alas, the paranoid monarch has foreseen this avenue to his downfall and has had the scribblings removed. You will have to carefully examine every juncture if you are to reach your objective.

Both conflict-driven and lookahead solvers are based on DPLL, making them complete solvers that translate SAT into a search problem. The similarity ends there. CDCL-solvers follow a cheap random walk strategy, whereas LOOKAHEAD carefully examine every juncture. This is the reason the latter perform well on hard SAT instances, which lack redundant information, and CDCL-solvers excel at industrial problems with lots of internal correlations (community structure).

The core of a lookahead solver is formed by the LOOKAHEAD procedure which selects the branch variable and—as a side effect—may produce *failed literals*: if a lookahead on $x = 1$ returns a conflict, then obviously x must be false. Failed literals shorten any clause that contain them, reducing the DPLL-tree.

4.4.1 Heuristics

LOOKAHEAD solvers rely heavily on heuristics as an alternative to costly—potentially NP-complete—operations on the data.

Pre-selection To reduce its workload, PRESELECT can confine LOOKAHEAD to a subset of all available variables. If a bad subset is selected, performance will suffer because fewer failed literals are detected and a good decision variable might be missed. Two heuristics in common use are: Li's $prop_z$ [51] and *clause reduction approximation* introduced in the MARCH solver [39].

Direction heuristic Do we choose the $x_d = 1$ or $x_d = 0$ branch of decision variable x_d? If we could always choose the optimal branch then backtracking would not be needed for satisfiable instances. Because such omniscience requires an NP-complete procedure, two heuristic strategies are used instead.

A: Estimate which subformula is most likely to be satisfiable. If the instance is satisfiable and the less likely branch is unsatisfiable then it will be a waste of time to investigate it. KCNFS, MARCH, and OKSOLVER use this scheme.

B: Estimate the subformula requiring the least effort. The POSIT solver assumes it is too hard to work out the odds that a subformula can be satisfied. This makes the cheapest branch its best starting point.

Interestingly, these two strategies are counterparts, the cheapest subformula is likely to have more constraints and is thus less probable to be satisfiable.

Difference heuristic In order to examine the impact of choosing a direction, a DIFF heuristic measures the difference between the instance before and after a lookahead. Metrics used are: the reduction in free variables, difference in clause count, decrease in clause size, or a combination thereof.

4.4.2 The LOOKAHEAD procedure

The LOOKAHEAD procedure outlined in listing 4.2 examines all free variables in ($\mathcal{P}$) (line 4) selected by PRESELECT (line 1) and returns the best decision variable x_i according to the decision heuristic DIFF (line 9), assuming there is a decision to be made (lines 7 and 8). In addition, it returns a simplified instance $\mathcal{F}$. In two cases it cannot return x_i: if the formula is `UNSAT` due to failed literals, or if all preselected variables are already assigned (line 6).

Listing 4.2: The LOOKAHEAD procedure, *Adapted from algorithm 5.3 in the Handbook of Satisfiability [41]*

```
1  P:= Preselect(F)
2  repeat
3    for x in P do
4      F:= LookAheadReasoning(F,x)
5      if (empty_clause in F|x = 0|) and (empty_clause in F|x = 1|) then
6        return(F|x = 0|, nil)
7      else if empty_clause in F|x = 0| then F:= F|x = 1|
8      else if empty_clause in F|x = 1| then F:= F|x = 0|
9      else H(x):= DecisionHeuristic(F, F|x = 0|, F|x = 1|)
10     if H(x) > previous_H(x) then Best_x:= x;
11 until NoNewThingsLearned
12 return(F, Best_x)
```

4.4.3 Additional reasoning

Because LOOKAHEAD on many selected variables is already expensive, adding extra reasoning may help to increase its utility and improve performance, at little additional cost.

Local learning If a lookahead on x leads to y, then $x \to y \equiv \neg x \vee y$. As implied by the 'local' label, this new knowledge is only valid in the current context and should be removed before backtracking.

Autarky reasoning An *autarky* is a partial assignment φ that satisfies all clauses it touches. A simple example is a literal that is never contradicted in a formula, a *pure literal*: if x occurs, but $\bar{x}$ does not, then assigning $x_1 \leftarrow \texttt{true}$ will trivially satisfy all clauses containing x without the risk of falsifying clauses elsewhere. Satisfied clauses can be removed from an instance $\mathcal{F}$, leaving the reduced instance $\mathcal{F}'$ equisatisfiable with its original. E.g., in $\{(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})\}$ x_2 occurs only in negated form, therefore $x_2 \leftarrow \texttt{false}$ satisfies two clauses, reducing the instance to $\{(\overline{x_1} \vee x_3)\}$. Now the reduced instance contains two new autarkies: the pure literals x_3 and $\overline{x_1}$. Other forms of autarkies are harder to spot, requiring more processing to detect their presence. LOOKAHEAD already performs a more in-depth investigation than CDCL, which means detection for specific autarkies can be added relatively cheaply.

4.5 Incremental SAT solving

Before solving a problem, SAT solvers typically perform a number of bookkeeping and preprocessing steps, such as allocating data structures and removing autarkies. This preliminary work involves a start-up cost whenever the solver starts working on a new instance. When solving many similar problems incurring the same initialization costs for every problem can become prohibitive, especially if the raw solve time for a problem is small, as is often the case on satisfiable instances. In addition to avoiding recurring start-up costs, incremental SAT solving [16] makes it possible to reuse knowledge gained from one instance towards solving a similar problem later on.

The reuse is enabled by splitting an instance into two parts: an instance, followed by a set of assumptions. The problem instance consists of clauses as normal, the assumptions are a set of unit clauses. In order for a problem to be satisfiable, all clauses must be true, therefore specifying unit clauses effectively means we assign values to literals. Thus feeding the solver with a starting assignment prior to working on the problem. After the solver has worked through the specified problem, it removes all assumptions. It then reads in an optional list of additional clauses and new assumptions which it adds to the existing clause database before solving again. Because learned clauses do not depend on the assumptions under which they were discovered, they can be reused when solving subsequent instances; variable and value statistics can also be carried over to the next session.

Restating a SAT formula in this way allows us to transform a complex problem into—possibly many—smaller subproblems. A list of subproblems can be solved in sequence, allowing the reuse of knowledge gained along the way, or in parallel, spreading the workload across multiple cores. Given enough subproblems both strategies can be combined.

Care should be taken to ensure that subsequent subproblems differ as little from each other as practicable so as to maximize reuse potential. Also, when splitting a large problem into many incremental subproblems it is important to ensure all parts are of equal difficulty. Measuring the hardness of a SAT instance is itself an NP-complete problem, making this an aspirational goal [41]. LOOKAHEAD solvers perform a more thorough analysis of a formula than CDCL solvers, which makes them ideal to split up large problems that do not exhibit an obvious partition scheme.

Encoding incremental formulas The `icnf` file format is an adaptation of the common `cnf` file format used in the annual SAT competition by Wieringa [73]. It replaces the `p cnf ...` header with a `p inccnf` line with no numbers. This is followed by one or more clauses in the usual format, mixed with assumption lines. Each assumption line starts with an '`a`' followed by one or more non-zero integers terminated by a zero. At first glance, an assumption line looks like a clause, however, instead of a disjunction of literals it represents a conjunction of unit clauses.

Listing 4.3: Example of an incremental `icnf` file

```
1    c Optional comment lines start with a c and may appear anywhere in the file
2    p inccnf
3    1 3 0
4    - 3 1 0
5    a -1 0
6    -2 -5 3 0
7    -1 2 0
8    a 5 -3 0
```

Every file starts with a `p inccnf` header, which does not specify the number of clauses so that data can be fed to the solver as a continuous stream via standard input. Reading of data will pause after every assumption line as the solver digests all clauses read thus far. It then adds the last assumption line read to the instance, discarding the previous assumption line, and solves the combined instance. For example, the problem up to line 5 of listing 4.3 above resembles formula $\mathcal{F}_1$, whereas the full input resembles formula $\mathcal{F}_2$ in equation (4.3) below.

$$\begin{aligned} \mathcal{F}_1 &= (1 \vee 2) \wedge (\bar{3} \vee 1) \wedge \bar{1} \\ \mathcal{F}_2 &= (1 \vee 2) \wedge (\bar{3} \vee 1) \wedge (\bar{2} \vee \bar{5} \vee 3) \wedge (\bar{1} \vee 2) \wedge 5 \wedge \bar{3} \end{aligned} \tag{4.3}$$

4.6 Parallelization of SAT

At first glance, SAT looks embarrassingly parallel. However, the volatile nature of the clause database and the ever changing structure of the search tree means that there is little scope for many threads to cooperate without excessive locking.

In the 2016 international SAT competition, parallel solvers running on 48 cores ran slower than sequential solvers [1]. Attempts have been made to implement SAT on a GPU, but to our knowledge no *complete* solver exists for this platform that outperforms sequential solvers. Dal Palù et al. manage to significantly speed up *unit propagation* on their GPU implementation [12]. Meyer et al. implement a SAT solver on a GPU, but are forced to strip out all modern enhancements, relying solely on the GPUs massive thread-count [54]; This leaves sequential solvers faster "by magnitudes".

In contrast, the CUBE AND CONQUER approach by Heule et al. does not try to micro-parallelize SAT, but uses a LOOKAHEAD solver to cut large problems up into smaller pieces, each of which is fed into an *incremental* CDCL solver. The inspection power of the delegating LOOKAHEAD solver is used to try and partition the main problem into parts of roughly equal difficulty. The receiving solvers are incremental versions of standard solvers. Making a sequential solver incremental is easy, because of the recursive nature of the underlying DPLL algorithm. CUBE AND CONQUER achieves linear speedup on a number of problems.

4.7 Summary

In this chapter, we have described a number of techniques used in modern SAT solvers. In general, CDCL solvers perform better on structured problems and LOOKAHEAD solvers better on hard random and mathematical instances.

Whilst all techniques described aid performance, there is a lack of deep understanding of how they interact. There is also no theory that predicts performance under a given set of circumstances.

In lieu of proper understanding the *portfolio solver* is used as a hack. Here multiple solvers are allowed to run in parallel on the same instance for a limited duration, after which the candidate that has made the most progress thus far is allowed to finish solving the instance.

Chapter 5: The GridWalker algorithm

In this chapter we describe our search algorithm: GridWalker. The purpose of GridWalker is to traverse a 3D lattice containing a pattern search description. The lattice has two spatial dimensions (called a *grid*) and one time dimension, matching Life's properties of space and time. This lattice corresponds to a *node* in the search tree. Every node holds a potential solution to the search for a pattern that fits the given criteria. GridWalker computes allowable ancestor constellations (at $t-1$) for a given future configuration (at $t=0$) using Conway's rules. It then tries to prune the number of ancestor states by eliminating contradictions between neighboring constellations.

GridWalker is invoked whenever a change is made to a grid that contains unknown pixels; it overlaps neighboring sub-lattices and removes any constellation from the grid that does not fit with adjacent overlapping sub-lattices. Because GridWalker only removes and never adds states it prunes the search tree, as early as possible, before employing more time consuming exploration techniques. All of the processing steps take constant time with very low overhead per operation.

In section 5.1 we describe a straightforward method to track the number of possible configurations. Section 5.2 details how GridWalker disperses conclusions spatially and temporally. Section 5.3 explains the Dpll tree traversal part of the algorithm, followed by section 5.4 which outlines how to implement GridWalker on a GPU. The final section contains a short summary.

5.1 Check of a pixel's neighborhood

Life is played on a 2-dimensional grid and thus we refer to cells in the universe as 'pixels'. Every pixel has 8 neighbors, as shown in figure 5.1.

NW	N	NE
W	core	E
SW	S	SE

Figure 5.1: The neighbors of a single pixel

The 3×3 neighborhood of a pixel at $t=0$ fully defines all possible immediate ancestor states of a single future pixel at $t+1$. An undefined pixel in the game at generation t has $2^9 = 512$ possible ancestor states in generation $t-1$. Simple enumeration of all states reveals that 140 of these states lead to a future live core pixel and 372 states lead to a dead core pixel. We can encode all possible ancestor states of a cell in a unary representation: a *set of bits*. We call this set a *slice*. The slice encodes all allowable states in a future pixels 3×3 ancestor lattice and contains 512 bits (64 bytes): one bit for every possible constellation of ancestor cells. Using a set of bits allows us to draw a number of conclusions as detailed in table 5.1. The constellation of a slice can be derived from known pixels in its immediate future. Sometimes the future state of a pixel is known, either because it is given in the starting configuration, or because it is deduced by the search algorithm. At other times, the state of a future pixel is unknown. If known, we constrain the past slice for that pixel to 140 or 372 states as listed above; otherwise, we only limit the allowable states in a slice if a neighboring overlapping slice has less than the maximum number of 512 allowable states.

Encoding of a slice The encoding of a slice follows the standard binary way of counting. The first element of the set matches 000000000, the second 000000001, and the last 111111111 (511). The least significant bit (associated with pixel 0) flip flops at each increment and every next bit changes with a lower frequency, exactly like bits of increasing significance do in standard binary encoding. Thus every element of the set encodes 256 `on` and 256 `off` states for 9 pixels with 2 possible states. The ordering of the pixels increases from left to right first and then from top to bottom.

pixel positions

0	1	2
3	4	5
6	7	8

encoding per constellation

pixel/constellation → ↓	0	1	2	3	4	5	6	7	···	504	505	506	507	508	509	510	511
0:	0	1	0	1	0	1	0	1	···	0	1	0	1	0	1	0	1
1:	0	0	1	1	0	0	1	1	···	0	0	1	1	0	0	1	1
2:	0	0	0	0	1	1	1	1	···	0	0	0	0	1	1	1	1
...	...																
8:	0	0	0	0	0	0	0	0	···	1	1	1	1	1	1	1	1

Figure 5.2: Encoding of a slice as a set of bits. A 3×3 grid can have $2^9 = 512$ possible constellations, each denoted by a bit in the slice. If its corresponding bit is set then the constellation is allowed, if unset, it is forbidden. The mapping of a bit to a constellation follows from the ordering of the pixels as shown to the left. Because pixel 8 is the most significant pixel, its first 256 constellations encode a dead pixel and the last 256 encode a live one. Other pixels alternate in line with their significance; e.g., pixel 0 flip-flops at every index. Every pixel has an equal number of alive and dead assignments.

The encoding of a slice allows us to draw a number of conclusions as outlined in table 5.1.

Table 5.1: Implications that can be derived from a slice

Full slice	**Meaning**
All bits set	All configurations are possible
Some bits unset	The configurations associated with unset bits are forbidden
All bits unset	There is a conflict; this state cannot exist
Only a single bit set	All 9 ancestor pixels conform to the state denoted by the bit position

As can be seen in figure 5.2 every ancestor pixel in the slice has a different encoding. We can create a mask for a given pixel in the `on` *state and count the number of allowed configurations for this* `on` *pixel. We do this by taking the conjunction of the slice and the relevant mask and then perform a bit count. The negation of the* `on` *mask is the* `off` *mask. The conjunction of the* `off` *mask and the slice will yield the bit count for the* `off` *state. If one of these two states has a zero bit count than that state is disallowed for the given pixel, forcing it to be 'stuck' in the opposite state.*

Masked slice	**Meaning**
No bits set in `on` masked slice	Force core pixel to off
No bits set in `off` masked slice	Force core pixel to on

Combining overlapping slices Each bit of the bitset maps to a state. If the bit is `on`, the corresponding state is permitted. The set of permitted states is called a *pass*. Disallowed states are denoted by `off` bits. These are called *holes*. Holes perform the same function as negations do in SAT solving; Negations allow for contradictions: take for example the clause 'A = b and not(b)', where b is a boolean variable. For a problem in SAT to be 'satisfiable' (`SAT`) every clause therein must have a possible assignment so that it evaluates to *true*. Clause 'A' above will always resolve to *false* and thus forces any problem that contains it to be 'unsatisfiable' (`UNSAT`).

The goal is to reduce the total number of allowed states, which is the same thing as increasing the number of holes, by propagating holes in a given slice to its neighbors. If we are looking for `SAT` then, ideally, we want to reduce the number of allowed states s_i in all slices to 1, which will yield a concrete specimen. The total number of solutions S to investigate is the product of all allowed states $S = \prod_{i=1}^{n} s_i$. This number grows quickly if even a small number of slices has $s_i > 1$. When looking for `UNSAT`, a single slice with zero allowed states suffices to prove that no solution exists.

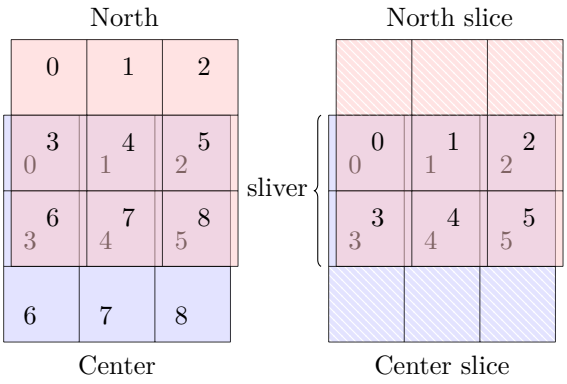


Figure 5.3: Overlapping two slices to form a sliver

Slivers Holes are transferred to neighboring slices by exploiting the fact that slices overlap. The 2×3 overlap between two adjacent 3×3 slices is called a *sliver* as shown in figure 5.3. Having 6 pixels, the sliver occupies $2^6 = 64$ bits. In order to extract a sliver from a slice we need to remove the non-overlapping pixels. Later, we expand the sliver back into a slice by adding the three removed pixels in again.

The method used to add pixels depends on its index and is detailed in table 5.2.

Table 5.2: Adding pixels to a sliver

Position to add pixels	**Method**
At lower indexes	Use Morton encoding [57] to bitwise interleave new pixels, starting with the highest indexed pixels.
At higher indexes	Double the sliver, adding an extra copy of itself to the end. Start with the lowest indexed pixels.
At intermediate indexes	Double the period of higher indexed pixels by copying sections of bits equal to the run length of the inserted pixel.

Obviously, the final case in table 5.2 is the general formula and inserting at lower and higher indexes are special cases thereof. The process is illustrated graphically in figures 5.4 and 5.5. Due to the encoding of slices (see figure 5.2) the run length is always a power of 2. Every pixel that is added to a sliver doubles the number of bits by duplicating existing ones. This is done by interleaving runs of bits with run lengths equal to the bit index, a run length of $2^0 = 1$ for pixel 0 and $2^8 = 256$ for pixel 8.

Removing pixels from slices to form slivers follows the opposite process. For example, as can be seen in figure 5.3, when overlapping slices using slivers the three non-overlapping bits in each slice are excluded. This reduces 512 initial states in the full slice down to 64 states in the overlapping sliver. This reduction is performed by taking the encoding of the pixel into account such that every run of bits representing off or '0' is disjointed with the following run of bits representing on or '1'. For example, for pixel 0 every even bit is `OR`'ed with every odd bit; for pixel 8 the run of bits 0..255 is `OR`'ed with the run 256..511. Doing so has the convenient side effect of renumbering the remaining bits such that the encoding in both slivers is the same (see figure 5.3).

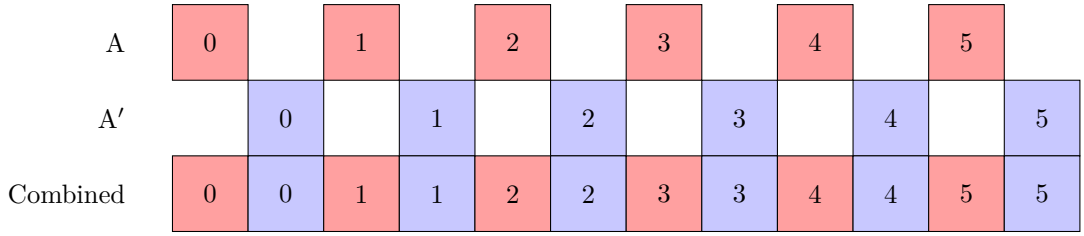


Figure 5.4: Bitwise interleaving with run length = 1 (Morton encoding)

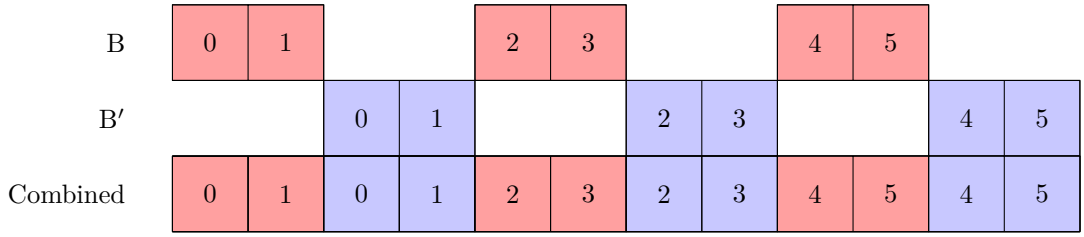


Figure 5.5: Bitwise interleaving with run length = 2

Overlapping slices As detailed above, removing different pixels requires distinct transformations. Take for example the confrontation of a slice with its northern neighbor using slivers as shown in figure 5.3. The northern slice needs to remove pixels 0, 1, 2 which involves folding $2^3 = 8$ consecutive bits into one. The center slice to the south removes pixels 6, 7, and 8 to produce its child sliver. The two resulting slivers are then `and`'ed into one sliver, which is expanded into two slice-masks by adding the appropriate pixels for each parent slice. Each mask is then conjoined with its grandparent slice, thereby transferring holes between the two neighbors.

5.2 Hole propagation algorithm

Now that we know how to use slivers to transfer data between neighboring slices, we can describe the algorithm used for the transfer of implications from the current grid to past, present, and future grids. Remember that a slice is the set of constellations for all 3×3 grids that are allowable predecessors for a given future core pixel. A slice centered at coordinate x, y, t is thus associated with the pixel$[x, y, t + 1]$ in the future grid. Conversely, a core pixel$[x, y, t]$ links to a slice$[x, y, t - 1]$ in the past grid as shown in figure 5.6.

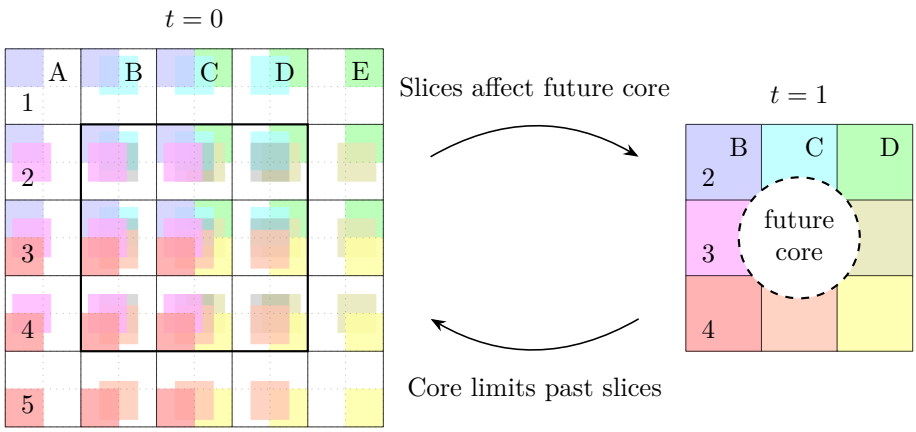


Figure 5.6: The past slices form the set of allowed predecessors for future pixels.

Any changes to a core at generation t will affect the slices at t that intersect it, because they denote the same pixels at the same point in time, as well as slices at $t-1$ in the past grid. Changes to a slice at $t-1$ will affect its embedded core at the same instant if any pixel becomes forced. In addition, because Life is forward deterministic, a given pixel at t affects zero or more future pixels at $t+1$. Any given pixel at t thus affects three consecutive generations at $t \in -1, 0, 1$. Note, the coloring of a future core pixel in figure 5.6 matches the coloring of the nine past pixels of its associated slice.

Propagation of holes in the current grid By overlapping slices—using slivers—we can transfer holes between neighboring slices. If this leads to a 'stuck' pixel we can propagate the effects thereof further backwards. Once this leads to a sufficiently low number of allowed states we can try and propagate the effects forward.

Hole propagation to the future grid If the number of allowable states in two adjacent non-overlapping slices become small enough then the algorithm will calculate all possible future cores arising from the union of allowed constellations in the two or more slices. This is done by calculating all patterns allowed by the slices a single generation into the future. Only the pixels in the future inner light cone are fully determined by the past and thus only those are taken into account; this means that for every (current) slice as starting point for the calculation, only a single future *core* pixel will be used. The number of future pixels considered will thus equal the number of input slices. All different outcome patterns are `or`'ed together. If the number of outcomes, given n output pixels, is less than 2^n there is a hole which will be propagated to the relevant intersecting future slices. This process is visualized in figure 5.7.

This process is most efficient when the size of the future core is 3×3 because then no translation is needed to update the future slice. It will, however, also work when there are fewer future core pixels. Holes in future cores are transferred to all slices that fully encapsulate the core. Slices that partially overlap the core gain holes only indirectly via the normal transfer mechanism using overlapping slivers.

An example using four overlapping slices (A, B, C, and D), where every slice has 509 holes (3 passes), is shown in figure 5.7 below. *Note that due to interaction between the pixels, the number of distinct constellations is not necessarily equal to the product of the number of constellations per constituent slice.*

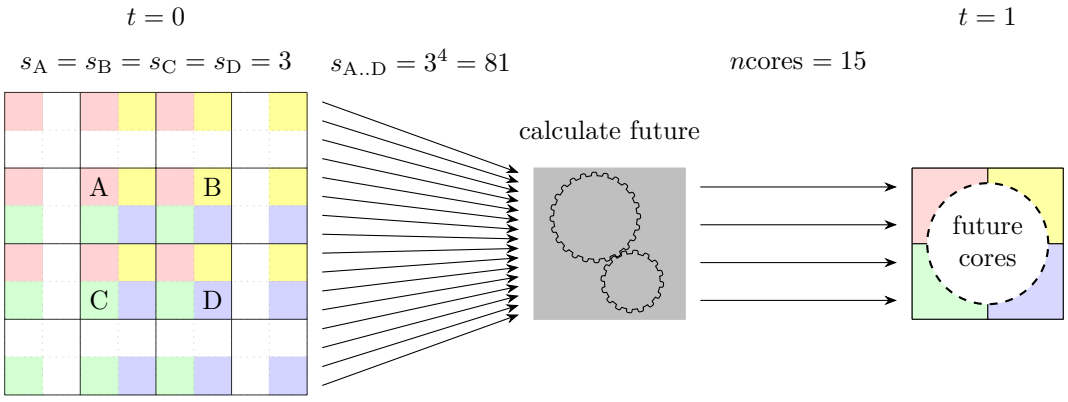


Figure 5.7: If the number of allowed states s_i for slice i is small enough that the total number of states in a block $\prod_{i=1}^{n} <$ treshhold, then it becomes feasible to calculate all possible constellations and eliminate impossible future cores. In this example $2^4 = 16$ future cores are possible. Let us assume only 15 distinct cores are calculated, then 1 constellation can be eliminated. Because a slice has 9 pixels and this core holds only 4, every slice that fully encapsulates it will gain up to $2^{(9-4)} = 32$ holes.

algorithm The high-level algorithm for entailments can be summarized in pseudo-code as follows:

Listing 5.1: Processing of entailments

```
repeat
    for each Grid in Grids do //Propagate holes, keep track of changes
        Grid.WalkCurrentGrid
        //If there are changes in slices then update future cores
        if Grid.Slices.ChangeCount > 0 then Grid.ProcessSlicesToFutureCores
        //If there are changes in cores then update past slices
        if Grid.Cores.ChangeCount > 0 then Grid.ProcessCoresToPastGrid
until Grids.ChangeCount = 0
```

Each of the three methods for propagation of changes, inside the current grid, to the future grid, and to the past grid are detailed in listings 5.2 to 5.4.

Listing 5.2: Progagate changes in the current grid

```
procedure WalkCurrentGrid
    for each Pixel do
        if Slice[Pixel] has holes then
            Combine Slice with neighbor slices
        if Slice has changed then
            Core[Pixel].Update      //Update the future core pixel
```

Listing 5.3: Progagate changes in the future grid

```
procedure ProcessSlicesToFutureCores
    for each changed Slice do
        c:= Slice.Count;  //number of allowed states
        rc:= Slice.Right.Count; lc:= Slice.Left.Count
        if (c < 8) and ( ((rc * c) < 16) or ((lc * c) < 16) ) then
            if ((rc * c) < 16) then
                FutureCores:= CalculateCores(Slice, Slice.Right)
                FutureGrid.Cores[Slice.Coordinate].Eliminate(not(FutureCores))
            if ((lc * c) < 16) then
                FutureCores:= CalculateCores(Slice, Slice.Left)
                FutureGrid.Cores[Slice.Coordinate].Eliminate(not(FutureCores))
```

Listing 5.4: Progagate changes to the past grid

```
procedure ProcessCoreToPastGrid
    for each changed Core do
        PastGrid.Slice:= PastGrid.Slice and LookupTable[Core]
```

Initialization of the grid Every grid in the collection of grids is a rectangular array of virtual pixels. Every virtual pixel is represented by a pair of perpendicular slices and a core. The arrangement of the grids depends on the kind of pattern in the search as detailed in table 5.3. We set up the next and previous pointers in every grid to match its search type.

Table 5.3: Configuration of grids for different searches

Search type	**grid count**	**grid configuration**
Garden of Eden	1	Search grid
Oscillator	p: oscillation period	A circular array$[0..p-1]$ of grids
Spaceship	p: oscillation period	Same as oscillator, grid$[p]$ is an alias of grid$[0]$ with translated coordinates depending on the spaceship's direction of travel.
Predecessor of known pattern	t: generations to search	A linear array$[0..t-1]$: grid$[t-1]$ contains the given pattern, all other grids are in its past.

The default state of a pixel is unknown, which means initially no slice has holes. In order to delimit a pattern we surround every grid with a border of zero pixels; for most patterns this border will be rectangular but it may also have an irregular shape. If the search begins with a starting pattern then we input that as well. We initialize the cores and slices to match the given pixels in all affected grids. After initialization we pick the low hanging fruit by working through all entailments as detailed in listing 5.1. When no more consequences can be inferred we employ speculative searching to explore non-trivial entailments.

The node contains a 3D array of grids with one time and two spatial dimensions. Every 3D grid array becomes a node in the search tree and the algorithm works through this tree as detailed in listings 5.2 to 5.4.

5.3 Speculative searching

This section describes the second part of the search program. Its purpose is to limit the number of possible solutions to evaluate. It does this by speculatively forcing a slice in a grid to a single constellation and testing to see if this forces a contradiction elsewhere in the grid.

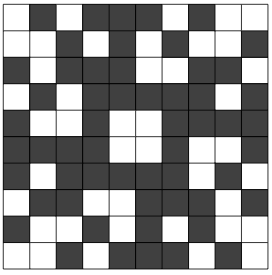


Figure 5.8: A 10×10 Garden of Eden pattern, *Adapted from Hartman et al. [37]*

5.3.1 What's in a slice?

A slice is the set of all allowable ancestors for a future core pixel. Starting with the Garden of Eden pattern in figure 5.8, we run the GridWalker algorithm until no further reductions in possible states per slice occur.

As can be seen in figure 5.9, the number of (known) allowable 3×3 ancestor slices for every pixel decreases depending on the number of iterations of the solver algorithm. If we only consider the future state of the core pixel for every ancestor grid there is only a single bit of information available which means that we either have 372 states leading to a future dead pixel or 140 states leading to its opposite.

A: single pixels

372	140	372	140	140	140	372	140	372	372
372	372	140	372	140	372	140	372	372	140
140	372	140	140	140	372	372	140	140	372
372	140	372	140	140	140	140	140	372	140
140	372	372	140	372	372	140	140	140	140
140	140	140	140	372	372	140	372	372	140
140	372	140	140	140	140	140	372	140	372
372	140	140	372	372	140	140	140	372	140
140	372	372	140	372	140	372	140	372	372
372	372	140	372	140	140	140	372	140	372

B: after applying GridWalker

268	132	161	115	119	121	163	130	242	268
242	186	112	159	112	164	119	175	186	132
130	175	117	108	111	176	171	117	112	161
163	119	171	118	117	121	118	108	159	115
121	164	176	121	200	200	117	111	112	119
119	112	111	117	200	200	121	176	164	121
115	159	108	118	121	117	118	171	119	163
161	112	117	171	176	111	108	117	175	130
132	186	175	119	164	112	159	112	186	242
268	242	130	163	121	119	115	161	132	268

Figure 5.9: Allowable states for a Garden of Eden pattern based on A: future core single pixels, B: after solving state A using the GridWalker algorithm; a minimal slice, with 108 constellations, is highlighted in red.

Panel A in figure 5.10 shows the number of states each slice can have due to its core pixel. Panel B shows the number of (known) allowable states after GridWalker has eliminated states in every slice that are incompatible with neighboring overlapping slices. The improvement from panel A to panel B takes 7 rounds where all north-south and east-west neighbor slices are overlapped. There is no need to overlap diagonal neighbors because the same result is attained using two rounds of orthogonal overlapping. The result is a dramatic reduction in states that need to be evaluated.

The number of states cannot be further reduced using GridWalker, however, we can sample every state speculatively and see if this creates a contradiction—i.e., a slice with zero allowed states—elsewhere in the grid. If it does, then that state cannot exist and can be removed from consideration. Luckily, there is no need to sample every possible state S ($S = \prod_0^{n-1} c_i$, where n = number of slices and c_i = number of allowable states c for slice i). In order for the pattern to exist, every slice within it must be valid. If we can invalidate a single slice in the grid, the entire pattern has no ancestors. Thus the slice with the lowest number of allowed states can be enumerated whilst removing all configurations that introduce contradictions. After this reduction, GridWalker is applied again to propagate any consequences thereof; speculative exploration continues until no further progress can be made.

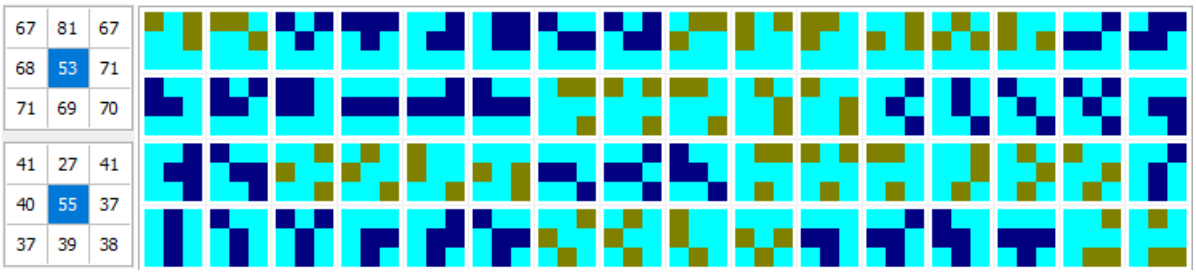


Figure 5.10: The minimal slice (highlighted in red) in figure 5.9-B has 108 possible states, 53 have the center pixel set (top-left, shown with dark blue pixels) and 55 have the center pixel unset (bottom-left, shown with bronze pixels).

The exploration is performed by forcing the slice to only have a single constellation and then applying the GridSolve procedure (see listing 5.6) on the grid. If the constellation is allowed then it should not produce any contradictions elsewhere in the grid; a contradiction is reached if the number of allowed states for any slice falls to zero. If we enumerate all possible constellations in that slice then some produce contradictions in the first round that GridSolve is applied. Others require multiple recursive applications of the reduction process (see listing 5.5). The process is illustrated in figure 5.11. If a slice has no valid constellations, that proves the entire pattern has no ancestors and is thus an orphan.

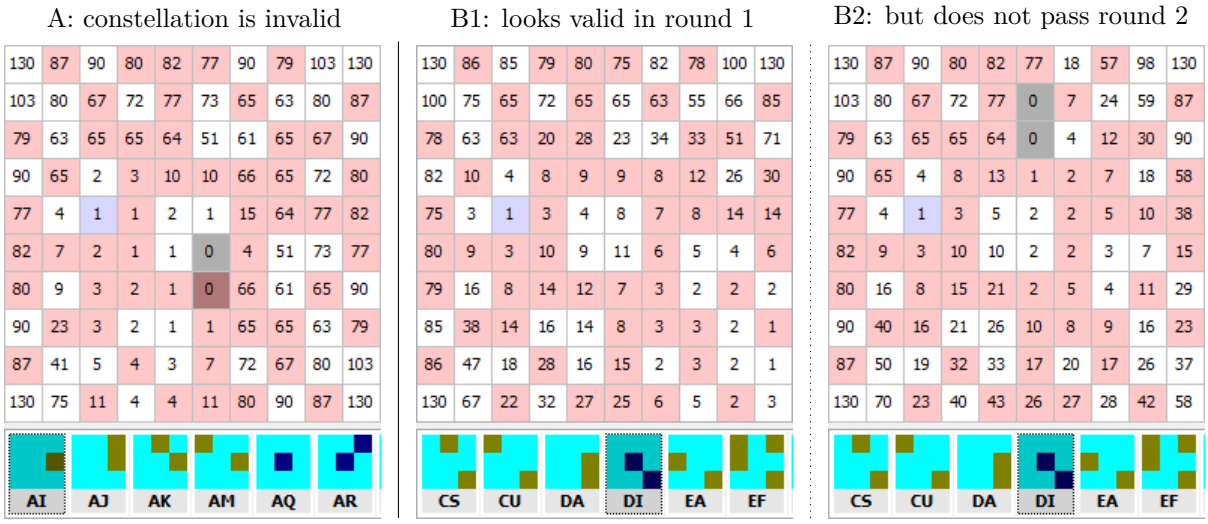


Figure 5.11: Solving states by speculative exploration. A: The highlighted slice (blue), forced to state `AI` (bottom-left) produces contradictions immediately. B: Forcing the highlighted slice to state `DI` (bottom-middle) does not produce problems in the first round of applying GridWalker (B1), but is shown to be impossible after a second round of solving (B2).

Listing 5.5: Find a Garden of Eden pattern

```
1  function Grid.GetUniqueSolution
2      //Find the slice with the lowest constellation count >=2.
3      MinSlice:= Self.GetMinSlice  //If none found, there is a unique solution
4      if (MinSlice = nil) and Self.ValidateLifeCube then return ValidSolution //SAT
5      Clone:= Self.Clone //Keep a clone so we can reset the grid for the next probe.
6      for Constellation in MinSlices do
7          Self.ForceSingleConstellation(MinSlice, Constellation)
8          if Self.GridSolve = IsValid then  //Return if UNSAT
9              Result:= Self.GetUniqueSolution //Recursive search for a solution.
10             if Result.IsValid then return ValidSolution //Early out @ unique solution
11         if (i < (MinCount-1)) then Clone.Overwrite(Self) //Reset the grid.
12     return GardenOfEdenFound //All alternatives are invalid, return UNSAT.
```

In order to perform this search automatically, we use a recursive method, outlined in pseudo code in listing 5.5 above. Before entering this method to search for a unique solution, or proof that none exists, we first perform a GridSolve (see listing 5.6 below) in order prune the grid. We then choose as a pivot the slice with the lowest constellation count $512 \geq n \geq 2$ (line 3).

If all slices in the grid have a constellation count of one (line 4) then we have a unique pattern that precedes our input pattern (an ancestor), meaning the input pattern is *not* a Garden of Eden pattern. If not, then we investigate each constellation of our pivot slice. If all of them are invalid, then we have found a Garden of Eden pattern. First we make a clone of the current grid (line 5), so that changes can be reverted later. Keep in mind that this is a recursive search, so there is actually a stack of clones matching the recursive calls. We then loop through all constellations (line 6) in the pivot and reduce the pivot slice so that only a single constellation in that slice is valid (line 7). A call to GridSolve (line 8) eliminates all states from the grid that are incompatible with this change.

If at this point no contradiction has been found then we recursively repeat finding a minimal slice (line 9); this pivot slice will not be the same as the earlier pivot, because a pivot must have a constellation

count ≥ 2, and we have just forced the previous pivot to have a count of 1. This recursion can thus only end in a satisfying assignment (a valid ancestor for the input pattern) or a contradiction. In the first case the search ends because we have found a counter example to the thesis that the input pattern is an orphan. In the latter case, we step back one recursion level and move on the to next candidate in the list to investigate. In order to move on, the grid is first reset to its starting point (line 11) and then a new candidate is selected for the next iteration of the loop (line 6). If, having gone through all candidates in the loop, no candidate yields a satisfying assignment then the pattern must be an orphan.

Because the algorithm performs a depth first search, it quickly finds a solution for those patterns that are satisfiable (i.e., not an orphan). This is fortunate, because the vast majority of patterns will not be Garden of Edens and we want to get through the non-hits as fast as possible.

Listing 5.6: Reduce a grid

```
1  function Grid.GridSolve;
2  repeat
3      ChangeCount:= 0
4      Self.LoopDirection:= not(Self.LoopDirection)
5      for point in Self do //loop through all coordinates
6          Status:= Self.SliverSolve(point);
7          if (Status.IsInvalid) then return Status //early out if grid is invalid
8          ChangeCount:= ChangeCount + Status; //+1 if changed, +0 if not changed
9  until (ChangeCount = 0)
```

The GridSolve method loops though all slices in the grid, looping forward on even loops and backwards on odd loops (line 4). It calls SliverSolve for every slice (line 6). SliverSolve confronts the given slice with its neighbors to the north and east going forward, and south and west going back, updating each slice as needed and returns a status detailing if a contradiction was found (line 7) and if any slices where altered. It tracks the number of changes (line 8) and keeps going until a full sweep is made with no more changes to the grid line 9.

5.3.2 Speculative evolution

In addition to the speculative exploration of individual slice constellations in the same generation there is the even more fine-grained option of speculatively evolving a subsection of the past pattern. In order to confirm the truth of the `SAT` conclusion the candidate solution can be verified using the LifeCube algorithm.

Because every slice has a unique 3×3 constellation, the grid can be trivially transferred to a bitmap. This pattern bitmap at $t = -1$ is fed into LifeCube and the output at $t = 0$ is compared against the bitmap of the input pattern using an `xor` operation. If all known pixels in the calculated result and the input pattern match, the `SAT` result is a true positive; if not it is discarded.

Bulk validation We also explored a slightly more complicated version of this code that would allow the verification of a grid that has more than one constellation. However, this would increase the complexity of the basic algorithm significantly, without a commensurate increase in performance. The problem is that translating the slice data in the grid to the bitmap format that LifeCube can digest needs a lot of computational effort, negating the bulk of the performance gain. The simple algorithm achieves the same result by performing a few extra recursions of the main solving procedure. Because no overly deep nested recursions where observed, we decided to keep not deploy this option.

Such bulk validation can be done by combining the constellations of multiple neighboring slices into a sub-pattern and calculating each possible sub-pattern forward a single generation. The resulting inner future light cone can be compared against the known parts of the future generation and any mismatches recorded.

This operation has already been implicitly performed for the future core pixel of a 3×3 slice, but not for larger past patterns. We can take a larger past pattern, e.g., a 5×5 subpattern made up by combining 9 slices. Taking as an example the extreme southeast square of pane B1 in figure 5.11, the number of subpatterns to evaluate is $(3 \times 2 \times 3 \times 2 \times 5 \times 2 \times 3) = 1\,080$. If none of these constellations' futures matches the pattern, then we can declare that no ancestor exists for that future; if, however, only a subset produces a mismatch valuable information can still be gleaned. Each of the non-unique slices (i.e., having an allowed state count > 1) has a fraction of the total count associated with one of its constellations. For example, the slice in the extreme southeast corner has 3 allowable constellations, each of which translates to 360 subpatterns. The same goes for the other non-unique slices, but the distribution differs per slice. If a given constellation of a slice is invalidated by all futures associated with it, that constellation can be removed from the slice.

5.4 GridWalker on a GPU

Even though a complete implementation of the GridWalker algorithm on a GPU is out of scope for this project, an outline of such an algorithm is presented here.

The most time consuming step ($95\%^{+}$) of GridWalker is the GridSolve routine. This simply loops over all slices in a grid and overlaps orthogonal neighbors until no more changes occur, or until an invalid slice is detected. In the CPU version, a sweep metaphor is used where we try to stabilize everything behind the broom and have all volatility—the dirt—occur in front on the broom. This minimizes the number of sweeps needed, but it also adds a serial dependency.

Maximizing concurrency If we are willing to increase the number of sweeps somewhat (see also section 6.2.1), this dependency can be broken and many slices can be calculated in parallel. In order to achieve this we break the grid up into a checkerboard configuration with white and black squares. On even sweeps white squares are processed from top-left to bottom-right. On odd sweeps black squares are processed in the same direction. The number of threads that are allowed to work on the same grid is limited so that they can either work on white slices at the bottom of the grid or black slices at the top, but never on slices of different colors that might interact. This is a common design pattern to increase the inherent parallelism of a problem. In addition, groups of 4 threads (8 such groups reside in every 32-thread warp) work on different parts of the same slice. For example, when working on a 10×10 grid there are $50 \times 4 = 200$ white or black slice-parts to work on. If we employ 128 threads (32 groups of 4) and keep on sweeping in the same direction in lockstep, they will never hinder one another. The 4 threads in a group work together on a slice using the GPU's shuffle intrinsics for share data between threads, this is fast and does not require external storage.

Fast memory access Just like a CPU has cores, which can execute multiple threads, a GPU has streaming multiprocessors (SM) that contain multiple warps (groups of 32 threads that execute in lockstep). Every SM has its own shared memory, primarily intended for data exchange between threads in the same SM. This memory also happens to be very fast to access, much faster than global memory. In an NVidia GTX 980 there is 48KiB of shared data per SM containing 1 024 threads. Storing grid data in shared memory means 48 bytes are available per thread. A slice takes up 64 bytes, 4 threads collaborate on a slice, but a group of 4 threads can only work on a white or a black slice, not both, and a

little separation is needed so different sweeps do not interfere. This means that we either overflow shared memory slightly, or idle a few threads. Idling 6% (utilizing 94%) of threads suffices to fully utilize this fast cache without having to spill data to slow global memory. Apart from slice data only a single status variable per grid to signal stabilization or unsatisfiability of the grid needs to stored in this cache. No lookup table or other external data is needed.

Before processing, a work unit copies a grid from global memory to shared memory. It then performs all processing either internally in registers or in shared memory. Afterwards the result is communicated using a single status variable with three possible values: unique solution (`SAT`), `UNSAT`, or grid stabilized (more work needed). Only in the latter case is the processed grid transferred back to global memory. In the first case, a confirmation calculation needs to be performed for which we have a GPU implementation (see section 3.2); this can be done in-place in shared memory.

Communication between CPU and GPU The CPU uploads sufficient grids to the GPU to keep it busy initially. Recalling listing 5.5, the GridSolve routine processes many grids that are the same except for a single altered slice. This means that only changes need to be transferred, which lightens the load. In the same vein, if a search description is similar to a previous one, only the delta is transmitted. This should suffice to keep the traffic between the CPU and the GPU from becoming a bottleneck.

5.5 Summary

GridWalker is the algorithm used to resolve logical consequences of a given pattern search description. When searching for a pattern, every generation t under consideration is stored in a *grid*. Grids do not store the pixels directly; instead the state of pixels is stored in two redundant data structures: *cores*, which store the current state of a single pixel tracking 4 possible states per pixel ('on', 'off', 'don't care' and 'unknown'). Additionally, the grid at $t-1$ contains the past configuration for the cores of generation t in so called *slices*. A slice covers a 3×3 sub-lattice of pixels and records all possible predecessors of a future pixel. Every future core pixel is fully defined by its 3×3 immediate past. Slices are sets of bit used to track the $2^9 = 512$ allowable states the pixel covered by that slice can be in. If a slice is empty, (the empty set), there is a contradiction which means the current pixel cannot have a past, which in turn means the current grid has no valid ancestors. All bits set, (the full set), means all constellations of predecessor pixels are possible. A partial set means that a subset of past pixel constellations is forbidden. The disallowed configurations are called *holes*. Because every slice overlaps with multiple neighboring slices we can exchange holes between them by calculating the intersection between two slices, a *sliver*. The sliver will pick up holes from both slices and will redistribute them back to the parent slices.

Advantages The GridWalker algorithm incorporating these operations is significantly faster than using plain Sat solvers as shown in table 6.1, because past logical implications are deduced using grid logic and future consequences are calculated using a fast Game of Life enumeration algorithm; spatial implications in the current generation are propagated using simple boolean operations. The search space is thus significantly pruned before the more time consuming parts of the search described in section 5.3.1 and section 5.3.2 start. In addition GridWalker's compact grid representation uses very little memory as can be seen in table 6.3.

Chapter 6: Experiments

In order to evaluate if we have answered our research question—*"can we speed up the search for Life patterns using a novel search engine?"*—we ran a number of experiments. To the best of our knowledge, the currently best performing search algorithms consist of SAT solvers, more specifically solvers that have won awards in the yearly SAT solver competition and therefore these form the basis for comparison. In all tests where individual times are listed, the lowest measured run time out of 10 tries is shown. The lowest is taken and not the average, because the minimal time has a lower bound, whereas a spurious longer run time has no upper bound. Some external factor might interfere, throwing the measurements off and skewing the averages unfairly.

6.1 Logic Life Search

Oscar Cunningham recently wrote a Python suite called LOGIC LIFE SEARCH [11], which can translate a given Life search pattern into CNF format and feed this into a variety of SAT solvers.

LOGIC LIFE SEARCH (hereafter: LLS) is able to search for orphans (aka Garden of Eden patterns), oscillators, spaceships, and still lives in a large number of Life-like CAs, using a number of command line arguments.

We used the development branch, dated 23 December 2018, and ran tests with the following SAT solvers:

- MAPLE_LCM_DIST by Fan Xiao et al. [75]: gold medal on the no limits track of the 2017 SAT competition.
- COMINISATPS PULSAR by Chanseok Oh [61]: gold medal on the main track of the 2017 SAT competition.
- LINGELING by Armin Biere [6]. This is the version committed to the SAT competition 2018, labelled 'bcj' and dated 17 May 2018. LINGELING is the solver Cummingham recommends be used with LLS.
- GLUCOSE 4.1 and its parallel invocation GLUCOSE-SYRUP 4.1 by Laurent Simon [66]: the 2016 versions of the well known gold standard. This is used to sanity check the results of the other solvers.

The first two solvers won gold awards in the 2017 SAT competition, albeit in different categories. To keep the test setup simple we did not select a parallel CUBE AND CONQUER solver; the current version of GRIDWALKER also works strictly in sequential mode. This makes for a fair comparison.

The above solvers are compared against the stock version of GRIDWALKER. In the next section we discuss a number of potential optimizations to the base algorithm.

6.1.1 Comparing GridWalker against Sat solvers

Using the Logic Life Search wrapper, we can evaluate different solvers so long as they conform to the command line interface of either MiniSat, glucose, or Lingeling.

lls takes a relatively long time to translate a pattern file into a CNF description that one of the commandeered solvers can work with. This means that we cannot use this setup to automatically search for new Garden of Eden patterns. In order to enable a fair comparison, we discount both the transcoding time from lls as well as the trivial transcoding time from GridWalker timings. In the former case this is easy, because every solver states the time in (milli-)seconds it took to solve the problem. In the latter case, we start the timer after transcoding has completed. Disk access time does not apply to any of the tests, because all the disk I/O is handled by the lls frontend, which is excluded from the timings.

We first run a competition with some small patterns, to see which Sat solvers perform best and then process some larger patterns using the winners for the first test.

Solving small patterns The five solvers discussed, show the following results for a number of known Garden of Eden patterns as well as a number of known non-orphans:

Table 6.1: Comparison of search algorithms.
All timings are in milliseconds, taking the lowest time out of 10 runs, running on an Intel Core i7 3820 with 4 cores (8 threads) at 3.9GHz with 32GiB of RAM. The Sat solvers ran under the Linux subsystem of Windows 10; GridWalker runs on Windows 10 on the same machine. Patterns #2, #3, and #4 are variations of pattern #1. The alterations relative to #1 are shown in orange. The final pattern has a number of undefined pixels shown in green. The fastest Sat solver results are displayed in bold. The penultimate column shows the speedup GridWalker achieves over the fastest Sat solver. The rightmost column lists a number of remarks, which are explained in the main text below.

Pattern	GoE?	CoMini-SAT	glucose	glucose-syrup	lingeling	Maple	Grid-Walker	GW vs SAT	notes
	Yes	150.82	154.35	**126.72**	237.50	1 814.57	8.09	15.7×	
	Yes	132.66	134.90	**121.13**	173.33	1 637.11	8.75	13.8×	A
	No	**52.06**	66.77	56.61	78.25	2 177.34	2.67	19.5×	
	No	90.26	**83.10**	122.05	179.93	1 900.67	0.24	346.3×	B
	No	37.29	**25.16**	36.03	45.46	30.22	0.49	51.3×	C
	Yes	**91.97**	95.25	109.97	148.22	2 524.72	3.55	25.9×	
	Yes	103.30	100.19	**97.73**	135.54	2 342.10	16.16	6.0×	
	Yes	**98.68**	99.86	102.12	135.54	1 955.40	15.04	6.6×	
Average		95.13	**94.95**	96.55	141.72	1 797.77	6.87	13.8×	

The results show that CoMiniSat and both versions of Glucose exhibit very similar performance, whereas Maple performs dismally on most benchmarks. Lingeling lags slightly behind the three fast solvers. Each of the fast Sat solvers are quite close in performance, as is evident in the average of

the timings. Although Glucose-syrup is a fast solver, this feat was achieved by running 8 threads in parallel, whereas every other contestant ran on a single thread. Using single-threaded solvers and running multiple searches in parallel on different cores is clearly a better use of computing power. GridWalker easily outperforms all Sat solvers.

A. The reason the Sat solvers work faster on pattern #2 than pattern #1 is unknown, although it is remarkable that all solvers performed better on the larger pattern. We speculate that the CNF encoding of pattern #2 has more redundancy, which is generally beneficial for Cdcl solvers. The reason GridWalker performs slower is simple: the larger the grid is, the more work each iteration of GridSolve has to perform to stabilize the sub-grids it works on. The difference in grid size is 10% and the difference in solve time is 8%.

B. The remarkable performance improvement by GridWalker between pattern #3 and #4 is easy to explain. The basic orphan in row #1, has 108 subpatterns to investigate, as discussed in section 5.3.1 and figure 5.11. In pattern #3 and #4 the minimal constellation count is the same, but located in a different slice. GetUniqueSolution walks through these constellations in sequence and pattern #4 happens to have a solution early in the sequence whereas pattern #3 has many unsatisfiable subpatterns prior to a satisfiable subpattern. In fact, pattern #4 presents a solution on the 4^{th} try, and pattern #3 needs 40 tries to get to a solution. The good result is an artifact of the order in which the potential solutions are sampled.

C. Both MiniSAT and Glucose on which the three fast solvers are based, are good all-round solvers without obvious weak points on highly redundant[1] problem encodings. Maple is specifically designed to beat these solvers on a number of tests in the Sat competition. The design choices seem to put this solver off balance in this application. The large performance deviations that solvers exhibit is of course the reason portfolio solvers are so popular. By trying many solvers the best solver can be applied to the problem.

Solving larger patterns In order to ascertain what happens to the run time of the Sat solvers and GridWalker when the number of cells to investigate increases, a batch of 20×20 and 40×40 patterns are subjected to the same test as before. The results are shown in table 6.2.

[1] The Sat competition uses the keyword 'industrial problems'.

Table 6.2: In the second test two batches 16 Garden of Eden patterns each, combined with 4 non-Garden of Eden patterns are used to compare run time between three SAT solvers and GRIDWALKER; all times are in milliseconds, the best performing SAT solver is highlighted in bold. The leftmost batch contains patterns with a 20×20 bounding box and the rightmost batch of patterns fit snugly inside a 40×40 grid. The last two columns show the gains GRIDWALKER makes vs the fastest SAT solver. Note that in order to prevent displaying '0.0×' in the bottom row, the gain is shown as a loss here, marked by a '-' prefix.

#	GoE?	Glucose		Comini-	Grid-	Glucose		Comini-	Grid-	Gain	
		stock	syrup	SAT	Walker	stock	syrup	SAT	Walker	20×20	40×40
1	no	**133**	160	423	53	2 224	**599**	721	129	2.5×	4.7×
2	no	**133**	157	451	53	2 226	**600**	1 220	128	2.5×	4.7×
3	no	**130**	153	426	53	2 222	**601**	697	141	2.4×	4.3×
4	no	**131**	156	448	53	2 224	**603**	1 461	128	2.5×	4.7×
5	yes	887	**634**	853	1 046	1 303	1 370	**1 280**	1 902	0.6×	0.7×
6	yes	1 240	**651**	912	1 887	**1 556**	2 366	1 697	3 511	0.3×	0.4×
7	yes	730	**667**	783	330	1 336	1 744	**1 159**	667	2.0×	1.7×
8	yes	1 209	**798**	1 226	1 162	**1 887**	2 095	2 229	1 487	0.7×	1.3×
9	yes	726	**571**	823	332	1 320	1 740	**1 299**	773	1.7×	1.7×
10	yes	662	**492**	665	240	**1 120**	1 380	1 314	618	2.1×	1.8×
11	yes	875	**605**	914	1 050	**1 324**	1 375	1 513	1 899	0.6×	0.7×
12	yes	1 241	**718**	906	1 848	1 557	2 377	**1 369**	3 519	0.4×	0.4×
13	yes	1 228	**713**	1 182	126	**1 319**	2 818	1 622	203	5.6×	6.5×
14	yes	800	**543**	719	228	1 163	1 379	**974**	422	2.4×	2.3×
15	yes	1 217	**671**	1 161	128	**1 319**	2 812	1 401	204	5.2×	6.5×
16	yes	966	**614**	915	82	**1 267**	1 606	1 405	143	7.5×	8.9×
17	yes	1 228	**958**	1 190	166	1 740	1 761	**1 664**	1 865	5.8×	0.9×
18	yes	784	**597**	715	77	**1 090**	1 283	1 414	1 281	7.7×	0.9×
19	yes	1 263	**790**	1 256	166	1 769	1 715	**1 573**	2 828	4.7×	0.6×
20	yes	949	**732**	903	1 542	**1 354**	1 673	1 852	52 327	0.5×	**-38.6**×
Total		16 532	**11 379**	16 873	10 622	**31 320**	31 896	27 865	74 175		

Table 6.2 shows that GRIDWALKER runs out of steam when confronted with larger patterns. The SAT solvers slow down much less. Note the final row, where GRIDWALKER is stuck in a pathological case and takes a disproportionate amount to solve the instance. We suspect that the strategy of always processing the entire grid until it stabilizes before recursing into a deeper level starts working against our solver. It might be better to invoke a mixed strategy where a subsection of the grid is iterated to a limited depth, so as to reduce the number of states that need investigating. This is where the SAT solvers strike a better balance. On the satisfiable instances GRIDWALKER holds up rather better, although the gains shown in table 6.1 using small patterns are much reduced.

Memory consumption The next experiment aims to compare memory usage of different solvers vs GridWalker. In order to maximize cache-efficiency GridWalker uses as little memory as possible.

Table 6.3: Memory utilization comparison of search algorithms. Unless otherwise specified, numbers indicate reported maximum memory consumption in megabytes. The final column lists the factor of space savings achieved by GridWalker compared to the most memory efficient Sat solver.

Pattern	GoE?	CoMini-SAT	glucose	glucose-syrup	lingeling	Maple	Grid-Walker	GW vs SAT
	Yes	97.3	93.7	389.0	1.8	106.8	37KiB	50×
	Yes	98.2	94.8	388.9	2.0	107.7	42KiB	49×
	No	96.4	93.4	388.1	1.8	106.9	106KiB	17×
	No	96.7	93.6	388.4	1.8	106.8	31KiB	59×
	No	96.4	93.3	388.3	2.1	95.2	201KiB	11×
	Yes	96.5	93.7	387.8	1.8	108.5	12KiB	154×
	Yes	96.6	93.8	387.5	1.8	108.3	48KiB	38×
	Yes	96.1	93.0	387.3	1.8	103.2	50KiB	36×

As can be seen in table 6.3 above, the various Sat solvers seem to need an inordinate amount of memory to solve a simple problem. Upon examination of the source code for this solvers, it turns out that most solvers allocate large blocks of memory for their internal data structures before starting their operations, so as to not incur the overhead of having to reallocate memory often; in fact all these solvers include a custom memory manager. The memory consumption shown is largely preallocation of blocks, not actual usage. Low memory consumption is one of the design goals of lingeling [5], which is why it preallocates less storage. GridWalker only requires a fraction of the memory reportedly used by Sat solvers. This is due to four factors. Firstly it has a more compact representation of the problem. Secondly, it performs a depth-first search which minimizes the memory needed. All Sat solvers here use clause learning, meaning that they create an expanding database of newly learned clauses, requiring ample storage, GridWalker does not learn clauses, it starts out with all the clauses it needs (encoded as a truth table in the form of slices) and forgets constellations instead. Finally, when GridWalker backtracks, it cleans up the data, shrinking its memory footprint. Therefore in the search for an orphan, where there is only one generation to consider, the memory requirement is roughly equal to grid size × recursion depth.

The lean memory usage of GridWalker is an artifact of the fact that the orphan search is organized as a depth-first tree traverse. On a more general pattern search for oscillators, spaceships, and such, a breadth-first or iterative deepening search is better suited. This would entail a much larger memory requirement, although we still expect it to be smaller than the Sat solvers' due to a more efficient problem encoding and 'forgetting'.

6.2 GridWalker optimizations

There are a number of improvements that can be made to the basic algorithm. Below we measure and discuss those that showed promise. Two of these optimizations are incorporated in the basic algorithm, to note: reversing the sweep direction and repeatedly overlapping slices until stabilization occurs. In these two cases we discuss what happens when these improvements are disabled. The potential optimizations that are not included by default are: the use of an interaction database and keeping track of changes. For these variants we describe the effect of their inclusion.

6.2.1 Reversing the sweep direction

In the basic algorithm, the GridSolve method sweeps the grid, from the top-left to the bottom-right on odd sweeps and the reverse direction on even sweeps. If we disable this optimization and simply perform the sweep from the top every time, it takes between 2× and 3× more sweeps to stabilize the grid.

The extra work in the mono-directional case happens because pushing changes forward propagates changes faster than pulling them back. The sweep only pulls back a single slice, but pushes forward changes in all slices not yet processed. In addition, because the algorithm is tuned to pushing changes, the area most recently visited will be less stable, thus more fruitful to process. The disadvantage of the single sweep is that it processes large areas that have already been stabilized without introducing new changes.

As a visual metaphor, imagine a lazy teenager assigned to cleaning an old floor; he will try to redistribute the dust into the cracks so as to technically conform with the request, without actually removing any dirt. Sweeping back and forth is a simple and effective strategy.

The result of this simple intervention is shown in table 6.4 below.

6.2.2 Repeatedly overlapping the same slices

A sweep works from top-left to bottom-right (and visa versa). We first confront a northeast slice with its neighbor to the west and update either of them if there are any changes. Then we overlap the slice with its neighbor to the south and do the same. If the northeast slice has changed in the second N-S comparison, this might affect its western neighbor later on. (Obviously, all directions are reversed in the returning sweep).

Since all the data required to evaluate the potential change is already in the cache, it is better to perform the overlap immediately, rather than waiting for the next sweep.

Theoretically, changes to center slices can continue repeatedly, which is why we keep running the E-W, N-S comparisons until no more changes to the center slice occur. Further local propagation after the first additional overlap rarely occur; no instances of deeper propagations thereafter have been observed.

From table 6.4 it is clear that alternating sweeps outperforms the simpler mono-directional sweep by a wide margin (100% improvement on average). The effect of repeatedly overlapping the same slice before moving on to the next is less clear and may well fall within the margin of measurement error. The reason for including this complication in the basic algorithm is explained in the next section where the effect of tracking changes is discussed.

6.2.3 Tracking changes

From the initial sweep of a grid to its eventual stabilization takes on average 5 to 6 sweeps with alternating directions. Because the algorithm halts when the final sweep yields no changes, the last sweep is always superfluous. Furthermore, observing the change counts per sweep yields an interesting pattern: the first few sweeps have large change counts (90%$^+$ of all slices undergo change). Then the number sometimes drops to below 10%, only to quickly increase above 50% before finally rushing to zero. If a large portion of sweeps only change a few slices, it might better to keep track of changed slices and only process those slices that have seen a (recent) change nearby. When tracking changes, it is beneficial to repeatedly overlap slices until no more changes occur, so as to keep updates as localized as possible. To improve comparability of both versions this slight complication is included in the basic algorithm.

To keep track of changes, we add a `set of bit` that records 'dirty' slices in the grid. At the start of the first sweep all slices are marked as such. If the slice does not change after a sweep, its corresponding dirty bit is cleared; otherwise, it is set. In addition, if a slice has changed, we also invalidate a number of slices next to it that might interact with the dirty slice.

Table 6.4: The effect of three simple optimizations: sweep the grid from alternating directions, repeatedly overlap the same slices, and only process changed slices. Columns 3..6 show the time taken in milliseconds to solve a pattern using respectively: the plain algorithm, which includes dual sweep and repeated overlaps (**basic**), sweeping the grid mono-directionally (**single**), a simpler algorithm that does not repeat slice overlaps (**no repeat**), and the tracking of changes in slices (**track**). The last three columns show by what factor basic outperforms single, loses against no repeat, and the varying effects of tracking changes.

Pattern	**GoE?**	**Optimization strategy**				**Is basic better than?**		
		basic	**single**	**no repeat**	**track**	**single**	**no repeat**	**track**
	Yes	8.09	14.77	**8.04**	8.68	1.8×	-0.6%	+7.3%
	Yes	**8.75**	15.22	**8.75**	9.64	1.7×	0.0%	+10.2%
	No	2.67	5.12	**2.64**	2.81	1.9×	-1,1%	+5.2%
	No	**0.24**	0.46	**0.24**	**0.24**	1.9×	0.0%	0.0%
	No	0.49	0.70	0.48	**0.29**	1.4×	-2.0%	-40%
	Yes	3.55	8.24	**3.51**	4.34	2.3×	-1.1%	+22%
	Yes	16.16	43.63	15.89	**14.80**	2.7×	-1.6%	-8.4%
	Yes	15.04	21.58	14.77	**12.03**	1.4×	-1.8%	-20.0%
Average		6.87	13.72	6.79	**6.60**	yes: 2.0×	no: -1.2%	y/n: -3.9%

Although we have tried to minimize the overhead of the change tracking using CPU intrinsic operations it turns out that this variation does not improve running time consistently, and in some instances hurts performance as can be seen in the final column of table 6.4.

The reason for this is that in order to not to corrupt the solution, i.e., not skip over relevant data, we need to add extra neighboring slices to the change list in addition to the dirty slices themselves.

This unavoidable step increases the number of slices under investigation by about 50%, limiting the improvement just enough to no longer make it a clear winner. Measurement of a few data points does not yield enough information to make an informed decision, therefore a larger pattern collection was sampled containing 125 000 randomly generated patterns. The results are shown in figures 6.1 and 6.2, where a large number of non-orphans are processed. When searching for novel patterns, the fast bulk of candidates will be misses and on these patterns tracking changes speeds up operations significantly.

6.2.4 Using a database of slice overlappings

In any initial pattern there are only 3 possible kinds of slices, those where the (future) core pixel is either `on`, `off`, or `unknown`. As these slices are confronted with their neighbors, their constellations change. Processing a large number of patterns, means that the same manipulations are repeated over and over.

To ascertain if it is beneficial to cache interactions between slices two dictionaries (hash maps) are used, one for N-S and another for E-W interactions, as well as a database for slices and a hash map to exclude duplicate entries from the slice database. As long as it is cheaper to lookup the result of an interaction between two slices in the dictionary, it should be possible to improve upon full calculation. Calculating the interaction between two slices means reducing two 64-byte slices into two 8-byte slivers and overlapping both into one. If there are changes the single sliver is expanded back into one or two slices—different from their parents—which are conjuncted with their grandparents.

lookup An interaction lookup involves hashing a pair of 32-bit slice identifiers and looking up the result in a hash table. The result of the overlapping is also cached. Instead of storing 64-byte slices in a grid, we now store 32-bit *slice identifiers*. The slice database is initially seeded with all 512 possible unity slices (slices denoting a single allowed constellation), as well as the canonical `on`, `off`, and `unknown` slices. All other slices are computed on-the-fly and added to the database as needed. We can store $2^{32} \times 64\text{bytes} = 256\text{GiB}$ of slice data in the database before running out of identifiers. The N-S and E-W dictionaries accept a slice-identifier pair as key and return a (possibly) different slice identifier pair as well as a change status variable. Because the unity slices have known identifiers, testing to see if a grid has a unique solution (i.e., is filled with unity slices) amounts to testing if all its slice-ids fall within 0..511.

The dictionary is updated incrementally while solving grids. This means that an ever increasing number of slices is added, hopefully reducing the need for recalculations.

storage If the total storage requirement of the database (dictionary and slice store) grows too large, data might get swapped out to hard disk. This would kill any speed advantage. This is prevented by keeping track of the number of times dictionary keys and slice store entries are requested. As RAM fills up we purge the entries with a low number of hits; a simple least frequently used (LFU) cache eviction scheme [52]. The test system used has 32GiB of RAM, hence the eviction scheme was not tested. The poor performance of the database did not necessitate optimization of the—admittedly sub-optimal—cache strategy. The database does have a useful side-effect: it allows us to track the number of manipulations needed to solve a pattern as can be seen in table 6.5. The statistics gathered may be useful for future optimizations.

We had hoped that using the database would give us anywhere from 0% to 100% speedup, but instead it yields much slower performance than plain GridWalker on the first run and slightly slower performance on the second run of the same pattern as shown in table 6.5. Despite the fact that hardly any slice data (64 bytes per slice) is touched and only interaction data is queried in a hash table ($2 \times 4 = 8$ bytes per interaction), using a database is still slower than calculating slice interactions from scratch, even in the best possible case.

Table 6.5: Optimization using a slice interaction database.
Unless otherwise stated, columns list number of operations performed. The explanation of the columns is as follows: Starting with an empty database, the first investigation of a pattern will require all interactions and all slices to be added (columns C, F, and I) to the slice, N-S, and E-W interaction database respectively. This adds considerable overhead, because not only does the algorithm need to do all the hard work of processing slices on the 1st run (column D) before an interaction can be stored, it also needs to query interactions (columns G and J) as well as calculate hash-keys of the slice data (column D again) to prevent duplicate slices from being added. Even on the first run of a pattern search some interactions recur, shown as a percentage gain (columns H and K) (calculated as 1 − (added/queried)). All this extra work leads to the worst possible case (column L) relative to the plain algorithm.
The picture looks much better (column M) on 2nd runs: an insignificant number of slice lookups is needed (column E) for enumeration of slice constellations in GetUniqueSolution (see section 5.3); interactions need only be queried (columns G and J), leading to a fixed 100% gain percentage (not shown). Still, even this best *conceivable* case does not improve on the run time of plain Gridwalker (column M).

A	B	C	D	E	F	G	H	I	J	K	L	M
Pattern	**GoE?**	**Slice queries**			**N-S matches**			**E-W matches**			**Slowdown**	
		added	**1st run**	**2nd**	**added**	**queried**	**gain**	**added**	**queried**	**gain**	**worst**	**best**
	Yes	58 114	309 136	278	46 108	72 617	36%	42 736	73 573	42%	3.88×	1.18×
	No	18 435	95 362	91	14 707	24 251	39%	13 536	24 639	45%	4.25×	1.21×
	Yes	63 108	331 614	276	49 749	79 402	37%	46 881	81 217	42%	4.42×	1.26×
	No	2 257	10 015	10	1 263	2 039	38%	1 222	2 113	42%	4.34×	1.00×
	No	826	3 738	32	443	4 404	90%	378	4 619	92%	1.75×	1.00×
	Yes	28 278	155 202	134	20 967	29 866	30%	20 847	30 940	40%	4.50×	1.12×
	Yes	96 420	540 707	477	81 593	147 166	45%	78 678	153 778	49%	3.75×	1.35×
	Yes	64 075	361 766	913	53 568	134 822	60%	50 925	140 875	64%	3.43×	1.17×
Average		18%	100%	0.1%	54%	100%	46%	50%	100%	50%	3.79×	1.16×

Part of the reason is that the slice interactions are coded as highly optimized assembly language using CPU intrinsic operations. Data in a single localized array is a very cache-friendly arrangement. The database on the other hand uses a standard linear probe hash table with a hash function that is fast, but not specifically tuned to the key data. Also, hash tables are notoriously cache-unfriendly.

However, if the database approach is to overtake plain brute-force calculation, it needs to achieve a low rate of new additions in relation to queries. To find new patterns, a lot of candidates need to be investigated. If the data stored in the database due to earlier candidates helps to avoid calculations later on, then optimizing the hash table might be worthwhile. Unless this learning effect occurs, a speedup remains out of reach. To investigate this, a search is performed on 125 000 potential Garden of Eden patterns in a 10×10 grid with diagonal mirror symmetry. To maximize possible learning effects, the candidates differ little from one iteration to the next, but there are no recurring patterns.

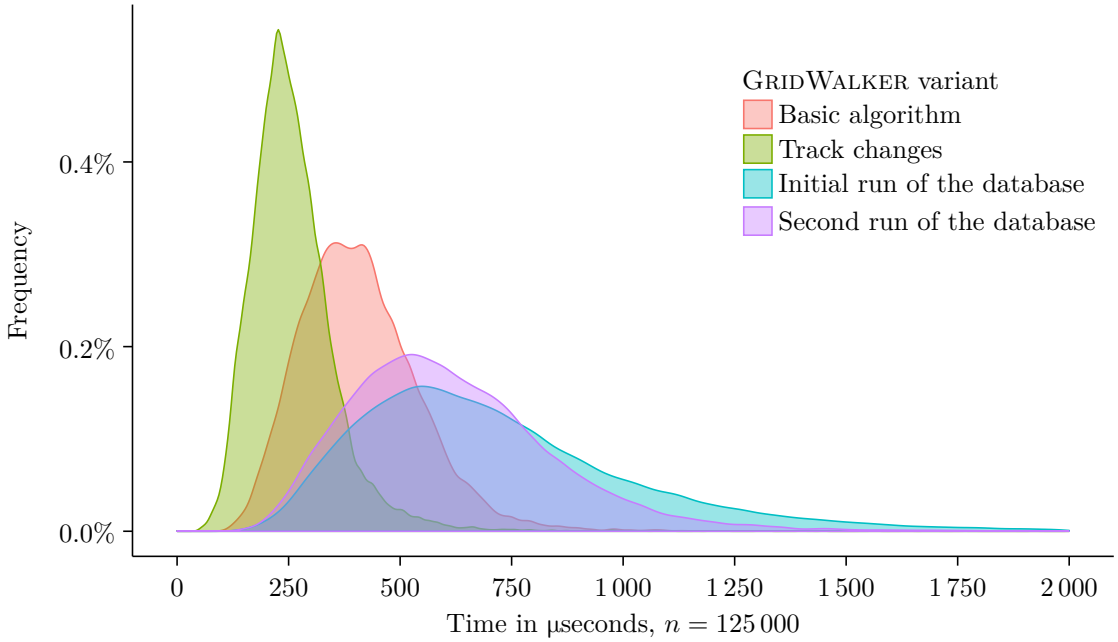


Figure 6.1: A density graph for 125 000 orphan searches using different variants of the GridWalker algorithm. The horizontal axis shows the time spend on the solve. The long tail is arbitrarily cut-off at $t = 2$ milliseconds, excluding 1 377 observations out of 125 000. Figure 6.2 discusses the nature of these extremes. The graph clearly shows what was ambiguous in table 6.4: tracking changes significantly outperforms the basic algorithm. It confirms the findings from table 6.5 that the slice database is significantly slower. The two curves for the best-case (purple) and realistic (blue) scenario for the database show little difference.

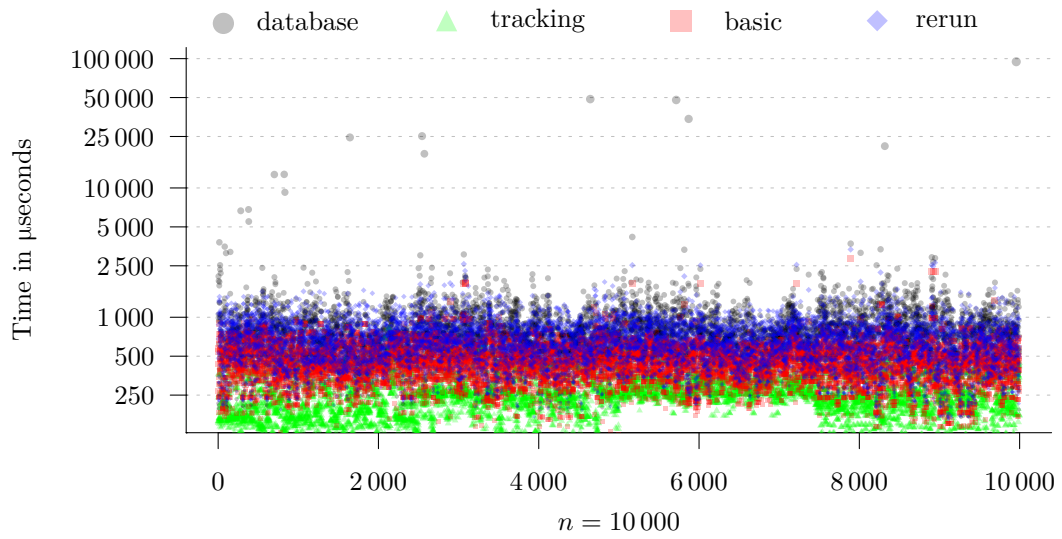


Figure 6.2: A scatter plot detailing the differences between three variants of the GridWalker algorithm. The database in normal operation is shown in grey, the same code is rerun a second time on a database where all interactions are already cached; this is used to ascertain the best case scenario for this algorithm. This mode of operation is shown in blue. The basic algorithm is displayed in red and the variant that tracks changes is shown in green. One thing that stands out is the huge spikes in run time by the database at regular intervals. This is due to the resizing of its hash-table when it reaches maximum occupancy. Clearly visible are the increasingly longer reallocation times; these are not outliers, but are integral to the hash table. The bottom of the graph is dominated by tracking-green; with the exception of a small red/blue appendix near $n = 6\,000$. Particularly hard patterns to solve stand out as spikes in the plot. All runs are independent of one another and were run on the same hardware at different times.

In order to understand why even the best case for the database algorithm performs so poorly, the run time data for the first 10 000 searches is detailed in the scatter plot below.

Our hypothesis is that because there are 2^{512} possible slices and $2^{512^2} > 10^{308}$ potential interactions between them, there are not enough recurrences of interactions to make caching worthwhile. This means that the average running time is actually much closer to the worst case than the unrealistic best case. And even the—admittedly sub-optimized—best case is slower than the simple algorithm.

There is a sliver lining to this cloud, caching data in a database complicates parallelization, whereas brute-force calculations are easy to run in parallel.

6.3 Incremental search

All the SAT solvers used in section 6.1.1 are meant to be used on problems that take minutes to hours; in this context, a small start-up cost is inconsequential. However, when running these programs on many easily satisfiable problems the initialization can dwarf the actual solve time. To minimize this effect some tests were rerun on an incremental version of MINISAT. Because satisfiable (Non-GoE) patterns vastly outnumber unsatisfiable (orphans), all test runs were made with non-orphans.

In order to generate the input data for the incremental solvers we updated the source code of Cunningham's LOGIC LIFE SEARCH to generate incremental `icnf` files.

The changes made are as follows: first the initial input pattern is replaced with a random string of `width` × `height` bits. This is supplemented with 100 000 additional Gray codes [32] derived from each preceding pattern. This ensures that each subsequent pattern only differs from the previous by a single pixel. Patterns with fewer than 40% or more than 80% `on` cells are discarded, because they are very unlikely to contain an orphan. This matches the work of Hartman et al. [37] where they used an incremental SAT solver to show that orphans in a 14×14 grid were normally distributed with a mean around 116 ($\approx 60\%$) `on` pixels.

An incremental solver will retain any learned clauses between assumptions. Old assumptions are thrown away after every run. Clauses can only be added. There is a trick to disable a clause though. By adding one of the literals in a clause as an assumption, that clause will be satisfied; this causes the satisfied clause to be temporarily disabled, because any clause known to be true is—eventually—removed from the database holding clauses still to be resolved as part of the normal processing of the solver. We can thus transform the formula under investigation at will whilst retaining any knowledge gained along the way.

Optimizing incremental problems Utilizing a incremental solver is most efficient if changes between consecutive assumption sets are minimized. To demonstrate this we add assumptions for a number of random patterns to an orphan searching formula and task an incremental version of MINISAT[2] with solving each incrementally. More modern solvers, such as GLUCOSE, have replaced `icnf` text file support with a C API which is hard to link with the Python code in LOGIC LIFE SEARCH. Solving 100 000 random patterns takes about 100 seconds. If Gray-encoding is used to perform a similar search starting with a random constellation where each subsequent pattern differs from the previous by a single bit, then the search takes only 8 seconds; more than 10x faster. Obviously, the pattern sets in the two searches do not overlap fully, but it does highlight the point that it pays off to order the assumption lines for maximum performance. This is illustrated in figure 6.3. GRIDWALKER lags behind the optimal scenario for the incremental solver, albeit that its run time does not depend on the ordering of the patterns, the performance difference for Gray-coded data and random data is less than 5%.

[2] MINISAT 2.2.0 from `http://www.minisat.se/` patched with Siert Wieringa's `icnf` patch at `http://www.siert.nl/icnf/minisat-2.2.0-for-icnf.patch`

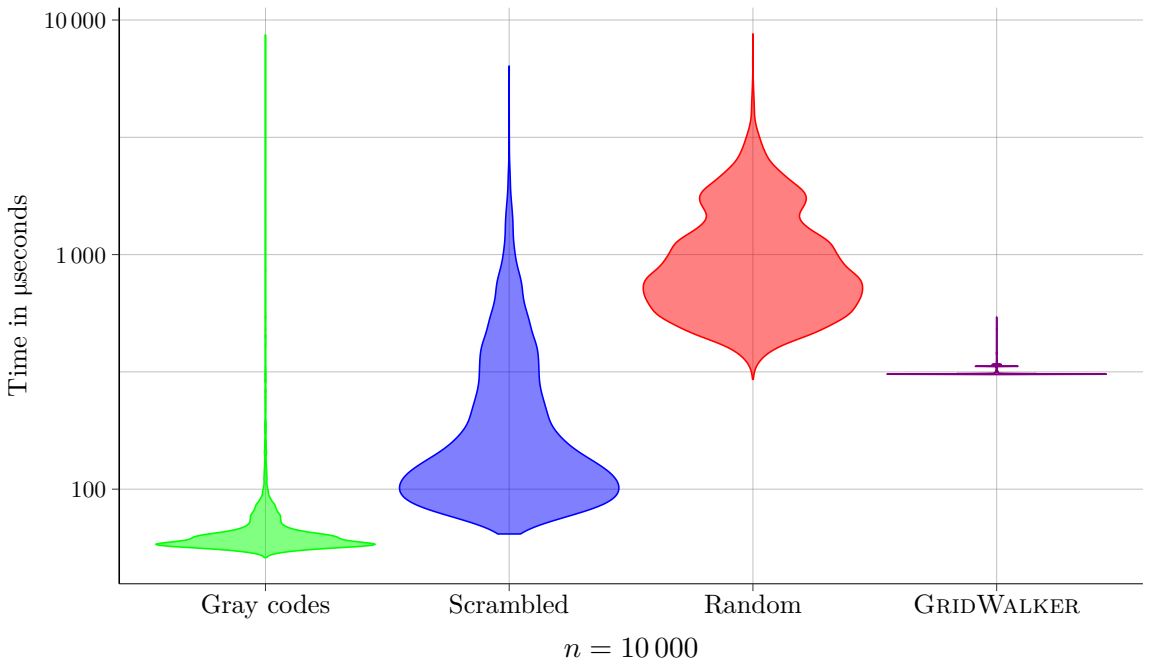


Figure 6.3: A violin plot detailing the differences between GridWalker and three incremental Sat solving strategies for 10 000 patterns. Three `icnf` files were prepared offline and fed to an incremental version of the MiniSat solver; The solve time—which excludes disk I/O—for each pattern was recorded. The fastest method ('Gray codes' in green) is to ensure that successive patterns differs from their predecessors as little as possible by using Gray encoding. Processing the same data in randomized order ('Scrambled' in blue) results in a 2× slowdown. Both GridWalker in tracking changes mode (see section 6.2.3) (purple) processing the Gray-coded dataset and solving random patterns ('Random' in red), taken from a different population than the Gray codes, run more than 10× slower than the optimal strategy. Because the Gray-coded data is so self-similar our algorithm walks the same path in many of them, resulting in a very narrow band of solve times. All violin plots are rendered using 'constant width', rather than the more common 'constant area', but the number of observations in each is the same.

6.4 LifeCube on a GPU

In order to test the performance of the LifeCube algorithm on the GPU, the future generations for all possible 7×7 patterns were calculated. For practical reasons we did not remove mirrored or rotated patterns. This required processing $2^{49} \approx 563$ trillion items. Procesing using an NVidia GTX1080 with 8GiB of VRAM took 22 hours, meaning that the algorithm can calculate over 7 billion such patterns per second. This is fast, even considering the fact that these are small patterns and there was no need to handle neighboring blocks.

Chapter 7: Conclusion

7.1 Results

It turns out that modern SAT solvers are hard to beat. Considering the nearly 100-fold overhead these solvers incur because they cannot inherently 'know' the rules of the Game of Life, they perform their search very efficiently. This project is based on a corollary to Wolpert & Macready's 'No Free Lunch' theorem [74] (*a search engine specifically tuned to a problem can always outperform a general solution*). The GRIDWALKER algorithm is the implementation thereof.

Paraphasing the research question: *'can we outperform modern* SAT *solvers when searching for patterns?'*, the answer is: 'Sometimes'. For small patterns the winner is clear, but as the patterns grow the brute-force approach built into our algorithm starts showing its limitations. For larger patterns GRIDWALKER sometimes outperforms its rivals, but there are pathological cases where it underperforms to a large degree.

When processing many satisfiable patterns the SAT solver can utilize an incremental scheme, here knowledge gained from processing previous patterns is reused, resulting in a significant speedup. This approach does not have an obvious analog in GRIDWALKER, which means that its performance remains unchanged, whilst the incremental SAT solver gains a 100× speedup.

Our implementation of the LIFECUBE algorithm was developed as a validation tool for the solver. It is useful in and of itself as a fast Life pattern enumerator, because it outperforms other algorithms on the GPU by a wide margin.

7.2 Contributions

1. The LIFECUBE implementation on the CPU outperforms Tom Rokicki's QLIFE algorithm by a factor 100. The GPU implementation thereof performs a 1 000× faster still. Although it is still an $O(N)$ algorithm. It can be a useful part of a search application.

2. The GRIDWALKER algorithm is a fast algorithm for finding patterns. It is narrowly outperformed by award-winning incremental SAT solvers, but we believe it can be substantially improved by a GPU implementation.

7.3 Limitations

Additional speedup needed for new discoveries In the current implementation and with the computing resources at hand, we have been unable to find new Garden of Eden patterns that improve on the state-of-art as per February 2019. In order to improve the lower bound from its current 6×6 we need to investigate over $\frac{2^{36}}{7.66} > 8 \cdot 10^9$ patterns. Assuming the average search takes about 1 ms, this needs about 100 CPU-days. A tenfold improvement in executing time would allow running such a query using a single machine within reasonable time. This applies even more strongly in the search for smaller—for some version of minimal—Garden of Eden patterns.

Weaknesses vs general SAT solvers GridWalker performs a breadth-first search of the clauses provided. This means that if there is a long dependency chain in the conclusions derived from a given set of clauses, then GridWalker will perform extra unnecessary work getting to the bottom of this chain. At every step along the way it will work through an entire grid of clauses. A modern SAT solver only works on a few clauses directly involved in the implication chain.

Scaling issues with large patterns As patterns under investigation grow larger, the brute-force approach starts to work against the algorithm. This is especially visible in large Garden of Eden patterns.

7.4 Future work

7.4.1 Possible improvements to the GridWalker algorithm

Currently the algorithm overlaps two neighboring 3×3 slices to form 2×3 slivers. However, every slice is really overlapped by three neighbors in the N-S or E-W directions. If we split up every 64-bit 2×3 sliver into two 8-bit 1×3 splinters, then considerable time can be saved in the change tracking version of the algorithm. The reason is that the extra information gained would allow the code to see whether a change propagated from slice A to slice B would affect slice C further down. If not, then there is no need to add slice C to the work-list. Some extra time, but no extra space, is needed to calculate and analyze splinters, but this should be more than offset by time saved due to a lower number of slices in the work list.

7.4.2 Generalize the search beyond Garden of Eden patterns

The current implementation only investigates a single generation into the past, i.e., against the forward deterministic grain of Life. This is arguably the hardest part, but generalizing the search algorithm to investigate multiple generations forwards and backwards would allow searching for interesting and useful—for Life-enthusiasts—patterns such as spaceships, oscillators, and other building blocks. There are, however, already many programs that do this, their only drawback being that they are never fast enough, which is what this project aimed to remedy.

7.4.3 Implementation on a GPU

In section 5.4 we outline an algorithm for performing the most time consuming part of the search on a GPU. With the intention of speeding up the search sufficiently to warrant the additional development effort. We intend to publish a paper and implementation doing exactly that. Considering that 5% of the execution time is not spent in parts that can 'easily' be run on a GPU, further optimization is still needed on those parts if the speedup is to exceed 20×.

7.4.4 Generalize the solver beyond Conway's Life

The solver is fine-tuned to Life and similar CA's, where every cell operates somewhat independently from others. This makes it unsuited for problems with long dependency chains. However, not every problem that can be encoded into SAT has this dependency problem. It might be worthwhile checking if the solver can be generalized to work on SAT instances where its breadth-first grid-based approach works in its favor.

Acknowledgments

Working on a masters' thesis is a large undertaking and many people have helped me along the way. A couple of them I wish to thank here.

First, I would like to thank my supervisor, Professor James Gain. For his excellent assistance and encouragement. His involvement made this project much more enjoyable and improved the quality thereof considerably.

Next, I would like to thank my fellow students. More specifically, Paul Wanjohi who is an apt hacker and has helped me out more than a few times; Ulysee Vimont for our many discussions and his assistance in whiteboarding me out of blind alleys; I also want to acknowledge Courtney Pitcher, Jacob Clarkson, Bryan Davis, Timothy Gwynn, Gerald Balekaki, Konrad Rapp, Charles Fitzhenry, and Tanweer Khatieb for their contributions.

Samuel Chetty and Craig Balfour are doing a sterling job in system administration, which is much appreciated.

I thank Brian DeRenzi and Mellisa Densmore for bringing me up to speed with statistics and R; Thomas Meyer for his teachings on propositional logic and for helping me secure funding; Geoff Nitschke for transferring his insights into genetic algorithms; Hussein Suleman for explaining the finer points of search; and Michelle Kuttel for her instructions on data visualization.

This work was funded, in part, by the South African Counsel for Scientific and Industrial Research (CSIR).

Finally I would like to thank Sonwabo for always making me smile when I come home and my Love, Jenny, for her love, support and commitment. Thank you for putting your life on hold so I can pursue my academic ambitions.

Appendix A: Life terminology

Mostly taken from LifeWiki: [43]

Conway's life, Life John Conway's cellular automaton with 2 states using the Moore neighborhood with rule B3/S23 as first described by Gartner [23].

Moore neighborhood (also: neighborhood, surroundings) Every cell has 8 neighbors in a two-dimensional square lattice; formally: all points at a 2-dimensional Chebyshev distance of 1 from that cell. The Moore neighborhood can be extended in the time dimension in which case we talk of the p_n or range n Moore neighborhood.

Figure A.1: The Moore neighborhood of a single cell

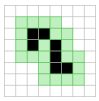


Figure A.2: The Moore neighborhood of a pattern

Figure A.3: The range 2 Moore neighborhood of a single cell

zone of influence The neighborhood to which a pattern can extend its influence in n generations. Note that a pattern without movement or growth does not extend its influence beyond its immediate neighborhood.
Because cells at the corner are starved of neighbors their influence only extents to $\sqrt{2}$ pixels every 2 generations on average.

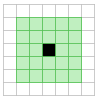


Figure A.4: The p2 Moore neighborhood of a single cell

Figure A.5: The p2 zone of influence of a single cell

generation A discrete unit of time. Every generation marks one pass where the rules of the automaton are applied to all cells in its universe.

rule B3/S23 A cell in the Moore neighborhood can either be dead (0) or alive (1). The state of a cell in generation $n+1$ depends solely on the configuration at generation n; all changes are applied simultaneously in discrete time intervals called generations. The state of a cell at time $n+1$ remains unchanged from its state in generation n if surrounded by 2 live cells. It will be alive at $n+1$ if surrounded by 3 live cells at time n and it will be dead at $n+1$ in all other cases.

hit (or target pattern) any pattern that meets the given search requirements.

bounding box the smallest rectangular array of cells that entirely contains a pattern.

population the number of live cells in a (specific generation of) a pattern. If we consider all the generations of a pattern then the we take count of the generation with the least number of cells as its population.

vacuum a region of space entirely consisting of dead cells.

pattern any finite configuration of cells surrounded by a vacuum in its sphere of influence that the automaton operates on.

sampling search a non-exhaustive algorithm that takes samples of the search space and returns either: a set of samples that meet its parameters, or the empty set as output. In order to speed up the search it is often limited to a given bounding box.

background the set of all patterns that are not hits.

void any part of the background that, to a search algorithm, is indistinguishable from random noise.

near miss a pattern that would exhibit the behavior required if only a mythical demon would flip a small number of its cells in a small number of generations. The fewer bit flips required, the closer the pattern is to being a hit. This demon can sometimes be replaced by adding a perturbing pattern that perturbs the original pattern's evolution in a such a manner that the combination turns from a miss into a hit.

Life engineering combining patterns with known properties in such a way that the combination forms a pattern with novel behavior. This is often done by hand, but can be automated.

still-life a pattern containing multiple live cells that remains the unchanged from one generation to the next.

Garden of Eden, orphan a pattern that does not have an ancestor. As such it cannot evolve from any starting pattern, it can only be created into existence. Note that an orphan cannot be a `still-life`, because a still-life is its own ancestor.

oscillator a pattern whose evolution repeats itself after a fixed number of generations. An oscillator with period 1 is called a *still-life*, an oscillator that has moved between two oscillations is called a spaceship, an oscillator that produces spaceships is called a gun, and a spaceship that produces guns is called a breeder.

p the period of oscillation: The number of generations it takes for an oscillator or spaceship to repeat itself.

true x any pattern x that is not trivially constructible from simpler sub-patterns. True oscillator: an oscillator where at least 1 cell oscillates with the full period (e.g. a pseudo-P34 oscillator can be made by combining any p17 with a p2; no true P34 oscillator is known).

Hershel a specific pattern that will self-replicate if given the correct perturbation. Because it produces a lot of active 'junk' in the course of its evolution it will be consumed unless special circuitry – a conduit – is put in place to clean up the debris. Hershel conduits form the basis of many complex patterns in Life.

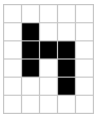


Figure A.6: Hershel

conduit A largely static enveloping pattern that perturbs a smaller pattern moving through it, so that it moves in a different direction or evolves other than it would without the envelope. The conduit restores itself after the vehicle moving through it has passed. The minimum time distance that two vehicles can follow each other through a conduit is called its period; this in spite of the fact that many conduits are still-lifes.

Hershel track A Herschel track or circuit is a series of Herschel `conduits` or other components, connected by placing them so that the output `Herschels` from early conduits become the input Herschels for later conduits, for an example see figure A.7.

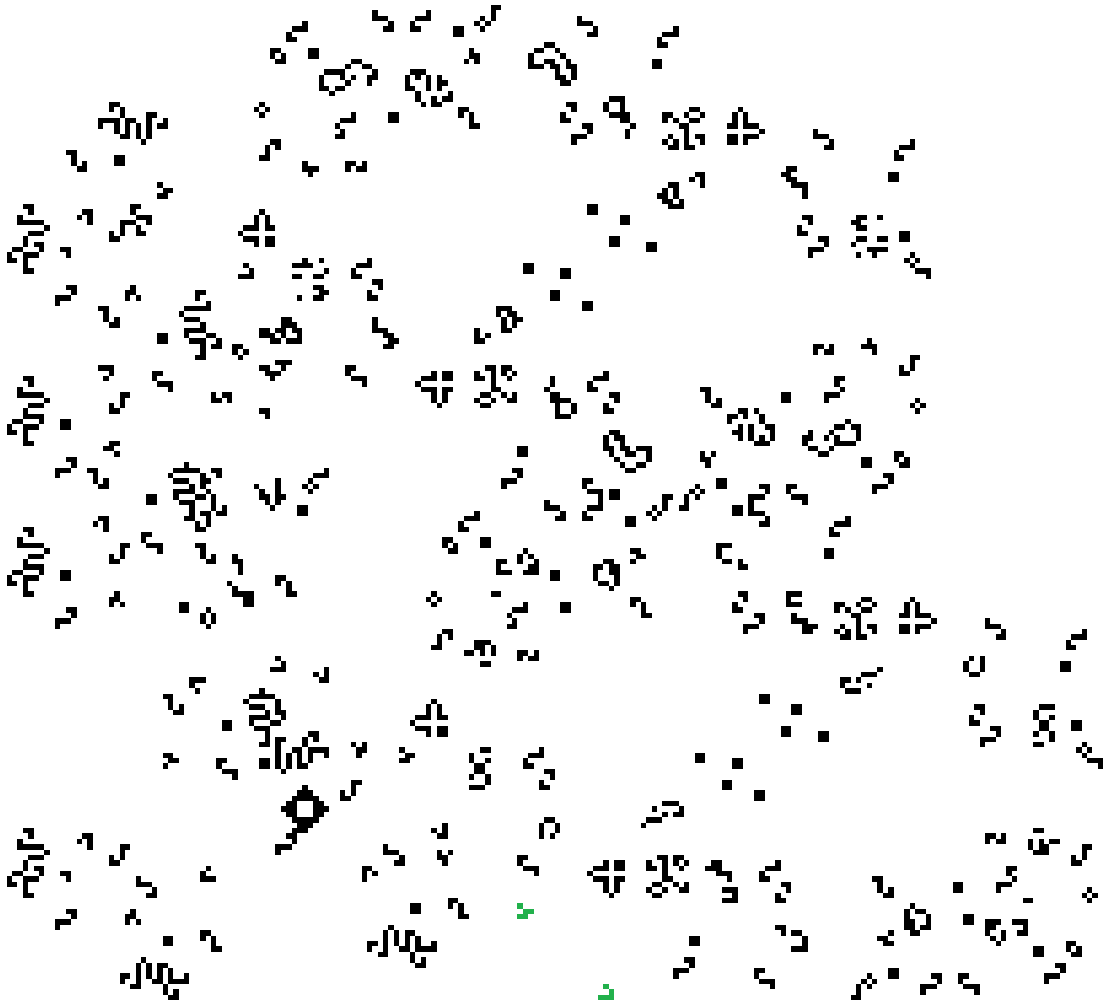


Figure A.7: A true p61 glidergun built using Hershel conduits by Chris Cain and Luka Okanishi in April 2016. A stream of gliders (shown in green) can be seen exiting the gun from the south.

slow salvo Patterns can be constructed by crashing gliders into things. To avoid timing issues, a slow salvo only releases the second glider after the effects from the first glider collision have stabilized.

universal constructor A pattern that can construct any pattern that can be created by colliding gliders. The path and timing of the gliders is encoded by a 'tape' read by a stream of gliders. The program on tape can be simplified using a slow salvo to build the target pattern.

elementary A `pattern` that cannot be reduced to a number of smaller subpatterns

knightship A `spaceship` traveling at an oblique angle.

Appendix B: Software

The software developed as part of this thesis is available at: `https://github.com/jbontes/Life64` The software is released under the open-source MIT license. The details can be read at `https://github.com/jbontes/Life64/license`.

Bibliography

[1] Balyo, T., Heule, M. J. H., and Järvisalo, M. SAT Competition 2016: Recent Developments. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.* (2017), pp. 5061–5063.

[2] Bell, D. Spaceships in Conway's Life. URL: `http://members.tip.net.au/~dbell/`, 1992. Accessed: 2017-04-01.

[3] Bell, D. I. Lifesrc 3.8 - a program that finds oscillators. URL: `http://members.canb.auug.org.au/~dbell/`, 2001. Accessed: 2017-04-01.

[4] Berlekamp, E. R., Conway, J. H., and Guy, R. K. What is Life? In *Winning Ways for Your Mathematical Plays*, vol. 2. Academic Press, 1982, pp. 817–849.

[5] Biere, A. Lingeling Essentials – Design and Implementation Aspects. Presented at the 5$^{\text{th}}$ workshop on Pragmatics of SAT 2014.

[6] Biere, A. Lingeling as per SAT Competition 2018. URL: `https://github.com/arminbiere/lingeling`, 2018. Accessed: 2018-12-23.

[7] Braunstein, A., Mézard, M., and Zecchina, R. Survey propagation: an algorithm for satisfiability. *Computing Research Repository ArXiv cs.CC/0212002* (2002).

[8] Buckingham, D. J., and Callahan, P. B. Tight bounds on periodic cell configurations in Life. *Experimental Mathematics 7*, 3 (1998), 221–241.

[9] Chapman, P. New results from Glue 2. URL: `http://b3s23life.blogspot.com/2006/02/new-results-from-glue-2.html`, 2006. Accessed: 2018-08-14.

[10] Cook, S. A. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), ACM, pp. 151–158.

[11] Cunningham, O. Logic-Life-Search – Cellular automata search program. URL: `https://github.com/OscarCunningham/logic-life-search/tree/develop`, 2018. Accessed: 2018-11-01.

[12] Dal Palù, A., Dovier, A., Formisano, A., and Pontelli, E. CUD@SAT: Sat solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence 27*, 3 (2015), 293–316.

[13] Davis, M., Logemann, G., and Loveland, D. A Machine Program for Theorem-Proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[14] Davis, M., and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the ACM 7*, 3 (1960), 201–215.

[15] Dershowitz, N., Hanna, Z., and Nadel, A. Towards a Better Understanding of the Functionality of a Conflict-Driven SAT Solver. In *International Conference on Theory and Applications of Satisfiability Testing* (2007), Springer, pp. 287–293.

[16] Eén, N., and Sörensson, N. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science 89*, 4 (2003), 543–560.

[17] Eker, S. Garden of Eden. URL: http://wwwhomes.uni-bielefeld.de/achim/orphan_11th.html, 2017. Accessed: 2018-11-01.

[18] Eppstein, D. Searching for spaceships. In *More Games of No Chance*, R. J. Nowakowski, Ed., vol. 42. Cambridge University Press, 2002, pp. 433–452.

[19] Eppstein, D. Life Search Programs. URL: https://www.ics.uci.edu/~eppstein/ca/search.html, 2004. Accessed: 2017-04-01.

[20] Fog, A. Optimization Manual 3 The microarchitecture of Intel, AMD and VIA CPUs. URL: http://www.agner.org/optimize/microarchitecture.pdf, 2017. Accessed: 2017-04-01.

[21] Fog, A. Optimization Manual 4 Instruction tables. URL: https://www.agner.org/optimize/instruction_tables.pdf, 2017. Accessed: 2017-04-01.

[22] Ganesh, B. From SAT to SMT. URL: https://ece.uwaterloo.ca/%7Evganesh/TEACHING/F2013/SATSMT/lectures/lecture4.pdf, 2011. Accessed: 2018-11-01.

[23] Gardner, M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American 223*, 4 (1970), 120–123.

[24] Gardner, M. Mathematical Games: On Cellular Automata, Self-Reproduction, The Garden of Eden and the Game of 'Life'. *Scientific American 224*, 2 (1971), 112–117.

[25] Gibson, M. J., Keedwell, E. C., and Savić, D. A. An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. *Journal of Parallel and Distributed Computing 77* (2015), 11–25.

[26] Gosper, W. R. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena 10*, 1-2 (1984), 75–80.

[27] Goucher, A. Catagolue census. URL: https://catagolue.appspot.com/census/b3s23, 2015. Accessed: 2017-04-01.

[28] Goucher, A. LifeLib: A fast C++ library for simulation and manipulation of patterns in cellular automata. URL: https://gitlab.com/apgoucher/lifelib#how-is-lifelib-structured, 2018. Accessed: 2018-05-11.

[29] Goucher, A. slmake: Tools for self-constructing circuitry in cellular automata. URL: https://gitlab.com/apgoucher/slmake, 2018. Accessed: 2018-05-11.

[30] Goucher, A. P., and Rokicki, T. A rather satisfying winter. URL: https://cp4space.wordpress.com/2018/03/11/a-rather-satisfying-winter/, 2018. Accessed: 2018-03-27.

[31] Goucher, A. P., and Rokicki, T. Announcement of the discovery of the first Elementary knightship. URL: http://www.conwaylife.com/forums/viewtopic.php?t=3303, 2018. Accessed: 2018-03-27.

[32] Gray, F. Pulse code communication, 1953. US Patent 2,632,058.

[33] Greene, D. Announcement of the linear propagator. URL: http://www.conwaylife.com/forums/viewtopic.php?f=2&t=1006&p=9901#p9901, 2013. Accessed: 2017-04-01.

[34] Gulwani, S., Srivastava, S., and Venkatesan, R. Program Analysis as Constraint Solving. *ACM SIGPLAN Notices 43*, 6 (2008), 281–292.

[35] Gustafson, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (1988), 532–533.

[36] Guy, M. J. T., and Bourne, S. URL: https://rosettacode.org/wiki/Conway%27s_Game_of_Life/ALGOL_68.

[37] Hartman, C. P., Heule, M. J., Kwekkeboom, K., and Noels, A. Symmetry in Gardens of Eden. *The Electronic Journal of Combinatorics 20*, 3 (2013), P16.

[38] Hensel, A. About my Conway's Game of Life Applet. URL: http://www.ibiblio.org/lifepatterns/lifeapplet.html, 2001. Accessed: 2017-04-01.

[39] Heule, M. March: Towards a Look-ahead SAT solver for General Purposes. Master's thesis, TU Delft, 2004.

[40] Heule, M. J., Kullmann, O., and Marek, V. W. Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer. In *International Conference on Theory and Applications of Satisfiability Testing* (2016), Springer, pp. 228–245.

[41] Heule, M. J., and van Maaren, H. Look-Ahead Based SAT Solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, ch. 5, pp. 155–184.

[42] Heule, M. J. H., Kullmann, O., Wieringa, S., and Biere, A. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing* (2012), K. Eder, J. Lourenço, and O. Shehory, Eds., Springer, pp. 50–65.

[43] Johnston, N. Welcome to LifeWiki: the wiki for Conway's Game of Life. URL: http://conwaylife.com/wiki/Main_Page, 2008. Accessed: 2017-04-01.

[44] Johnston, N. The Online Life-Like CA Soup Search. URL: http://www.conwaylife.com/forums/viewtopic.php?f=7&t=131&p=570, 2009. Accessed: 2017-04-01.

[45] Karafyllidis, I. Acceleration of cellular automata algorithms using genetic algorithms. *Advances in Engineering Software 30*, 6 (1999), 419–437.

[46] Knuth, D. E. *The Art of Computer Programming. Volume 4, Fascicle 1, Bitwise Tricks & Techniques : Binary Decision Diagrams.* Addison-Wesley, 2009.

[47] Knuth, D. E. *The Art of Computer Programming. Volume 4, Fascicle 6, Satisfiability.* Addison-Wesley, 2015.

[48] Korf, R. E. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence 27*, 1 (1985), 97–109.

[49] León, P. A., Martínez, G. J., and Chapa-Vergara, S. V. Complex Dynamics in Life-like Rules Described with de Bruijn Diagrams: Complex and Chaotic Cellular Automata. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on* (2012), IEEE, pp. 245–251.

[50] Li, C. M., and Anbulagan, A. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the 15th international joint conference on Artifical intelligence* (1997), vol. 1, Morgan Kaufmann Publishers Inc., pp. 366–371.

[51] Li, C. M., and Anbulagan, A. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the 15th international joint conference on Artificial intelligence* (1997), vol. 1, Morgan Kaufmann Publishers Inc., pp. 366–371.

[52] Maffeis, S. Cache Management Algorithms for Flexible Filesystems. *ACM SIGMETRICS Performance Evaluation Review 21*, 2 (1993), 16–25.

[53] Marques-Silva, J. P., and Sakallah, K. A. GRASP—A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design* (1996), IEEE, pp. 220–227.

[54] Meyer, Q., Schönfeld, F., Stamminger, M., and Wanka, R. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *2010 International Conference on High Performance Computing and Simulation (HPCS)* (2010), IEEE, pp. 306–313.

[55] Miéville, China Tom. *The City & The City*. Macmillan, 2009.

[56] Moore, E. F. Machine models of self-reproduction. In *Proceedings of Symposia in Applied Mathematics* (1962), vol. 14, American Mathematical Society, pp. 17–33.

[57] Morton, G. M. A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing. *Unpublished manuscript* (1966).

[58] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference* (2001), ACM, pp. 530–535.

[59] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., and Simon, L. Impact of Community Structure on SAT Solver Performance. In *International Conference on Theory and Applications of Satisfiability Testing* (2014), Springer International Publishing, pp. 252–268.

[60] Ninagawa, S., Yoneda, M., and Hirose, S. $1/f$ fluctuation in the "Game of Life". *Physica D: Nonlinear Phenomena 118* (1998), 49–52.

[61] Oh, C. COMiniSatPS Pulsar. URL: `https://baldur.iti.kit.edu/sat-competition-2017/solvers/nolimits/COMiniSatPS_Pulsar_no_drup.zip`, 2017. Accessed: 2018-12-23.

[62] Roberts, S. *Genius At Play: The Curious Mind of John Horton Conway*. Bloomsbury Publishing USA, 2015.

[63] Sapin, E., Bull, L., and Adamatzky, A. A Genetic Approach to Search for Glider Guns in Cellular Automata. In *2007 IEEE Congress on Evolutionary Computation* (2007), IEEE, pp. 2456–2462.

[64] Sapin, E., Bull, L., and Adamatzky, A. Genetic Approaches to Search for Computing Patterns in Cellular Automata. *IEEE Computational Intelligence Magazine 4*, 3 (2009), 20–28.

[65] Simkin, M. LifeCube: Very fast Conway Game Of Life iterator based on AVX. URL: `https://github.com/simsim314/LifeCube`, 2016. Accessed: 2017-04-01.

[66] Simon, L. The Glucose SAT Solver. URL: `http://www.labri.fr/perso/lsimon/downloads/softwares/glucose-syrup-4.1.tgz`, 2017. Accessed: 2018-12-23.

[67] Skolem, T. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Bewiesbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen. *Videnskapsselskapet Skrifter, I. Matematisk-naturvidenskabelig Klasse 4* (1920), 1–36. English translation in Heijenoort J. van (ed.). *From Freege to Gödel. A Source Book in Mathematical Logic*, (1967) Harvard University Press, pp. 1879–1931.

[68] Sörensson, N., and Eén, N. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization. In *Proceedings of SAT 2005: Eighth International Conference on Theory and Applications of Satisfiability Testing* (2005), Springer, pp. 502–518.

[69] Šuhajda, K. Hersrch: A Herschel track search program. URL: `http://entropymine.com/jason/life/software/hersrch_20040327.zip`, 2004. Accessed: 2017-04-01.

[70] Trevorrow, A., Goucher, A. P., Greene, D., and Rokicki, T. G. Golly For exploring cellular automata like the Game of Life. URL: `https://sourceforge.net/projects/golly/files`, 2016. Accessed: 2017-04-01.

[71] Tseytin, G. S. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic Part 2* (1968), 115–125.

[72] Von Neumann, J., and Burks, A. W. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, 1966.

[73] Wieringa, S., Niemenmaa, M., and Heljanko, K. Tarmo: A framework for parallelized bounded model checking. In *Workshop on Parallel and Distributed Methods in verifiCation (PDMC), Proceedings of the 8th International* (2009), vol. 14, pp. 62–76.

[74] Wolpert, D. H., and Macready, W. G. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation 1*, 1 (1997), 67–82.

[75] Xiao, F., Luo, M., Li, C.-M., Manyaà, F., and Lü, Z. Maple_LCM_Dist as per SAT Competition 2017. URL: `https://github.com/msoos/Maple_LCM_Dist`, 2017. Accessed: 2018-12-23.