Computing in the fractal cloud: modular generic solvers for SAT and Q-SAT variants

Denys Duchier, Jérôme Durand-Lose *, and Maxime Senot

LIFO, Université d'Orléans,
B.P. 6759, F-45067 ORLÉANS Cedex 2.

Abstract. Abstract geometrical computation can solve hard combinatorial problems efficiently: we showed previously how Q-SAT —the satisfiability problem of quantified boolean formulae— can be solved in bounded space and time using instance-specific signal machines and fractal parallelization. In this article, we propose an approach for constructing a particular *generic* machine for the same task. This machine deploies the Map/Reduce paradigm over a discrete fractal structure. Moreover our approach is *modular*: the machine is constructed by combining modules. In this manner, we can easily create generic machines for solving satifiability variants, such as SAT, #SAT, MAX-SAT.

Keywords. Abstract geometrical computation; Signal machine; Fractal; Satisfiability problems; Massive parallelism; Model of computation.

1 Introduction

Since their first formulations in the seventies, problems of Boolean satisfiability have been studied extensively in the field of computational complexity. Indeed, the most important complexity classes can be characterized —in terms of reducibility and completeness— by such problems *e.g.* SAT for NP [4] and Q-SAT for PSPACE [20]. As such, it is a natural challenge to consider how to solve these problems when investigating new computing machinery: quantum, NDA and membrane [19], optical [12], hyperbolic spaces [17]... (for an overview of the status of NP-complete problems under several physical assumptions, see [1]).

This is the line of investigation that we have been following with *signal machines*, an abstract and geometrical model of computation. We showed previously how such machines were able to solve SAT [6] and Q-SAT [7] in bounded space and time and in quadratic *collision depth*, a model-specific time complexity measure defined by the maximal number of consecutives collisions, which is better suited to the strong parallelism of signal machines. But in both cases, the machines were instance-specific *i.e.* depended on the formula whose satifiability was

* This work was partially supported by the ANR project AGAPE, ANR-09-BLAN-0159-03.

to be determined. The primary contribution of the present paper is to exhibit a particular *generic* signal machine for the same task: it takes the instance formula as an input encoded (by a polynomial-time Turing machine) in an initial configuration. This single machine replaces the whole family of machines designed previously (one machine for each formula) and formulae are now represented by inputs (as for classical algorithms) instead of being encoded by signals and rules of machines. We further improve our previous results by describing a *modular* approach that allows us to easily construct generic machines for other variants of SAT, such as #SAT or MAX-SAT. We also introduce a new technical gadget, the *lens device*, which automatically scales by half any beam of signals. These constructions are in cubic collision depth instead of quadratic for the previous instance-specific solutions.

The model of signal machines, called *abstract geometrical computation*, involves two types of fundamental objects: dimensionless *particles* and *collision rules*. We use here one-dimensional machines: the space is the Euclidean real line, on which the particles move with a constant speed. Collision rules describe what happens when several particles collide. By representing continuous time on a vertical axis, we obtain two-dimensional *space-time diagrams*, in which the motion of the particles are represented by lines segment called *signals*. Signal machines can simulate Turing machines and are thus Turing-universal [10]. Under some assumptions and by using the continuity of space and time, signal machines can simulate analog models such as computable analysis [9] and they can even be super-Turing by embedding the black hole model (forecasting a black-hole in signal machines is highly indecidable as shown in [8]).

Other geometrical models of computation exist: colored universes [14], geometric machines [13], optical machines [18], interaction nets [16], piece-wise constant derivative systems [2], tilings [15]...

All these models, including signal machines, belong to a larger class of models of computation, called *unconventional*, which are more powerful than classical ones (Turing machines, RAM, `while`-programs...). Among all these abstract models, the model of signal machines distinguishes itself by realistic assumptions respecting the major principles of physics —finite density of information, respect of causality and bounded speed of information— which are, in general, not respected all at the same time by other models. Nevertheless, signal machines remain an abstract model, with no a priori ambition to be physically realizable, and is studied for theoretical issues of computer sciences such as computability power and complexity measures.

As signal machines take their origins in the world of cellular automa , not only can they be a powerful tool for understanding the latter's complex behaviors, implementing computations [11] and proving universality [3] but they too can be viewed as a massively parallel computational device. This is the approach proposed here: we put in place a fractal compute grid, then use the Map/Reduce paradigm to distribute the computations, then aggregate the results.

The Map/Reduce pattern, pioneered by Lisp, is now standard in functional programming: a function is applied to many inputs (map), then the results are

aggregated (reduce). Google extended this pattern to allow its distributed computation over a grid of possibly a thousand nodes [5]. The idea is to partition the input (petabytes of data) into chunks, and to process these chunks in parallel on the available nodes. When solving combinatorial problems, we are also faced with massive inputs; namely, the exponential number of candidate solutions. Our approach is to distribute the candidates, and thus the computation, over an unbounded fractal grid. In this way, we adapt the map/reduce pattern for use over a grid with fractal geometry.

Our contribution in this paper is three fold: first, we show how Q-SAT can be solved in bounded space and time using a *generic machine*, where the input (the formula) is simply compiled into an initial configuration. This improves on our previous result where the machine itself depended on the formula. Second, we propose the first architecture for fractally distributed computing (the *fractal cloud*) and give a way to automatically shrink the data into this structure by means of a *lens device*. Third, we show how generic machines for many variants of SAT can be assembled by composing independent modules, which naturally emerged from the generalization of our previous family of machines into a single machine solving Q-SAT. Each module can be programmed and understood independently. We also discuss notions and choices of complexity measures which strongly depend of the considered model of computation, and we argue that *collision depth*, a time complexity measure introduced in [6], is more relevant to signal machines. The collision depth of the given construction is cubic in the size of the input formula and space complexity is exponential.

The paper is structured as follow. Signal machines are introduced in Section 2. Section 3 presents the fractal tree structure used to achieve massive parallelism and how general computations can be inserted in the tree. Section 4 details this implementation for a Q-SAT solver and Section 5 explains how some variants of satisfiability problems can be solved with the same approach. Complexities are discussed in Section 6 and conclusions and remarks are gathered in Section 7.

2 Definitions

Signal machines are an extension of cellular automata from discrete time and space to continuous time and space. Dimensionless signals/particles move along the real line and rules describe what happens when they collide.

Signals. Each *signal* is an instance of a *meta-signal*. The associated meta-signal defines its *speed*. Figure 1 presents a very simple space-time diagram. Time is increasing upwards and the meta-signals are indicated as labels on the signals. Generally, we use over-line arrows to indicate the direction and speed of propagation of a meta-signal. For example, $\overrightarrow{\mathbf{a}}$ and $\overleftarrow{\mathbf{a}}$ denote two different meta-signals: the first travels to the right at speed 1, while the other travels to the left at speed -3. $\mathbf{w}$ and $\mathbf{a}$ are both stationary meta-signals.

Collision rules. When a set of signals collide *i.e.* when they are at the same spatial location at the same time, they are replaced by a new set of signals according

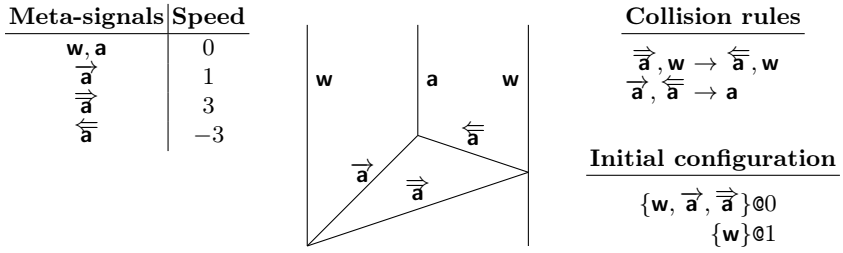


Fig. 1. Geometrical algorithm for computing the middle

to a matching collision rule. A rule has the form: $\sigma_1, \ldots, \sigma_n \to \sigma'_1, \ldots, \sigma'_p$ where all σ_i are meta-signals of distinct speeds as well as σ'_j (two signals cannot collide if they have the same speed and outcoming signals must have different speeds). A rule matches a set of colliding signals if its left-hand side is equal to the set of their meta-signals. By default, if there is no exactly matching rule for a collision, the behavior is defined to regenerate exactly the same meta-signals. In such a case, the collision is called *blank*. Collision rules can be deduced from space-time diagrams as on Fig. 1 where they are also listed on the right.

Definition 1. *A* **signal machine** $\mathcal{M}$ *is a triplet* $\mathcal{M} = (M, S, C)$ *where* M *is a finite set of meta-signals,* $S : M \to \mathbb{R}$ *is the speed function which assigns a real speed to each meta-signal and* C *is the set of collision rules.*
An **initial configuration** c_0 *is a finite set* $c_0 = \{(\sigma_i, x_i) \mid \sigma \in M \text{ and } x \in \mathbb{R}\}$. *For a signal* σ_i *at initial position* x_i*, we also note* $\sigma_i @ x_i$.

A signal machine is executed starting from an initial configuration which corresponds to the input. The evolution of a signal machine can be represented geometrically as a *space-time diagram*: space is always represented horizontally, and time vertically, growing upwards. The geometrical algorithm displayed in Fig. 1 computes the middle: the new **a** is located exactly halfway between the initial two **w**.

3 Computing in the fractal cloud

Constructing the fractal. The fractal structure that interests us is based on the simple idea of computing the middle illustated in Figure 1. We just indefinitely repeat this geometrical construction: once space has been halved, we recursively halve the two halves, and so on. This is illustrated in Figure 4(a) (due to format restrictions, most part of the figures are postponed at the end of the paper and are also given in bigger size in Appendix), and can be generated by the following rules*:

$$\mathbf{w}, \overleftarrow{\mathbf{a}} \to \mathbf{w}, \overrightarrow{\mathbf{a}} \qquad\qquad \overrightarrow{\mathbf{a}}, \overleftarrow{\mathbf{a}} \to \overleftarrow{\mathbf{a}}, \overleftarrow{\mathbf{a}}, \mathbf{a}, \overrightarrow{\mathbf{a}}, \overrightarrow{\mathbf{a}}$$

* For brevity, we will always omit the rules which can be obtained from the others by symmetry. We refer to Appendix for more details.

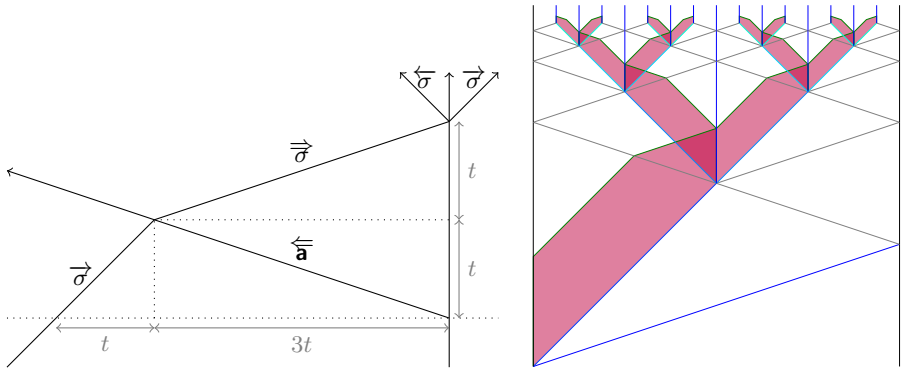
using $\{\mathbf{w}, \overrightarrow{\mathbf{a}}, \overrightarrow{\overrightarrow{\mathbf{a}}}\}$@0 and $\{\mathbf{w}\}$@1 as the initial configuration. This produces a stack of levels: each level is half the height of the previous one. As a consequence, the full fractal has width 1 and height 1.

Distributing a computation. The point of the fractal is to recursively halve space. At each point where space is halved, we position a stationary signal (a vertical line in the space-time diagram). We can use this structure, so that, at each halving point (stationary signal), we split the computation in two: send it to the left with half the data, and also to the right with the other half of the data.

The intuition is that the computation is represented by a *beam* of signals, and that stationary signals split this beam in two, resulting in one beam that goes through, and one beam that is reflected.

Unfortunately, a beam of constant width will not do: eventually it becomes too large for the height of the level. This can be clearly seen in Fig. 4(b).

The lens device. The lens device narrows the beam by a factor of 2 at each level, thus automatically adjusting it to fit the fractal (see Fig. 2). It is implemented by the following meta-rule: *unless otherwise specified, any signal* $\overrightarrow{\sigma}$ *is accelerated by* $\overleftarrow{\overleftarrow{\mathbf{a}}}$ *and decelerated and split by any stationary signal* $\mathbf{s}$.



(a) Narrows by 2 (b) Effect on propagation

Fig. 2. The lens device

Generic computing over the fractal cloud. With the lens device in effect, generic computations can take place over the fractal by propagating a beam from an initial configuration. We write `[(`$\overrightarrow{\sigma_n} \dots \overrightarrow{\sigma_1}$`)spawn]` for an initial configuration with a sequence $\overrightarrow{\sigma_n} \dots \overrightarrow{\sigma_1}$ of signals disposed from left to right on the space line. Geometrically, it can easily be seen that, in order for the beam to fit through the first level, the sequence $\overrightarrow{\sigma_n} \dots \overrightarrow{\sigma_1}$ must be placed in the interval $(-\frac{1}{4}, 0)$.

Modules. A *module* is a set of signals which correspond to a given task. We describe a module by the parametric abstraction defining its instance-specific contribution to the initial configuration in the form `[module]` $= \overrightarrow{\sigma_n} \dots \overrightarrow{\sigma_1}$ and

by the collision rules describing how this module interacts with other modules (rules omitted by lack of space can be found in Appendix).

Stopping the fractal. For finite computations, we don't need the entire fractal. The `[until(`n`)]` module can be inserted in the initial configuration to cut the fractal after n levels have been generated. We set : `[until(`n`)]` $= \overrightarrow{\mathsf{z}}\overrightarrow{\zeta}^{n-1}$.

$$\begin{array}{lll}
\overrightarrow{\zeta}, \overleftarrow{\mathsf{a}} \to \overleftarrow{\mathsf{a}_\circ}, \overrightarrow{\zeta} & \overrightarrow{\zeta}, \mathsf{a} \to \mathsf{a}_\circ & \overrightarrow{\zeta}, \mathsf{a}_\circ \to \overleftarrow{\zeta}, \mathsf{a}_\circ, \overrightarrow{\zeta} \\
\overrightarrow{\mathsf{z}}, \overleftarrow{\mathsf{a}} \to \overleftarrow{\mathsf{a}_\circ}, \overrightarrow{\mathsf{z}} & \overrightarrow{\mathsf{z}}, \overleftarrow{\mathsf{a}_\circ} \to \overleftarrow{\mathsf{a}}, \overrightarrow{\mathsf{z}} & \overrightarrow{\mathsf{z}}, \mathsf{a}_\circ \to \overleftarrow{\mathsf{z}}, \mathsf{a}, \overrightarrow{\mathsf{z}} \\
\overrightarrow{\mathsf{z}}, \mathsf{a} \to \mathsf{a}, \overrightarrow{\mathsf{z}} & \overrightarrow{\mathsf{z}}, \overrightarrow{\mathsf{a}} \to \overrightarrow{\mathsf{a}_\circ} & \overrightarrow{\mathsf{a}}, \overleftarrow{\mathsf{a}_\circ} \to \emptyset
\end{array}$$

Table 1. Stopping the fractal.

The subbeam $\overrightarrow{\zeta}^{n-1}$ are the inhibitors for $\overrightarrow{\mathsf{z}}$. One inhibitor is consumed at each level, after which $\overrightarrow{\mathsf{z}}$ takes effect and turns $\overleftarrow{\mathsf{a}}$ into $\overleftarrow{\mathsf{a}_\circ}$ which finally annihilates the constructing signals via the rule $\overrightarrow{\mathsf{a}}, \overleftarrow{\mathsf{a}_\circ} \to \emptyset$, bringing the fractal to a stop. Thus, a computation `[(`$\overrightarrow{\sigma_n} \ldots \overrightarrow{\sigma_1}$`[until(`$n$`)])spawn]` uses only n levels. It can be seen geometrically that, for the collision of $\overrightarrow{\mathsf{z}}$ with $\overrightarrow{\mathsf{a}}$ to occur before the latter meets with $\overleftarrow{\mathsf{a}}$, $\overrightarrow{\mathsf{z}}$ must initially be placed in $(-\frac{1}{6}, 0)$.

4 A modular Q-SAT solver

Q-SAT is the satisfiability problem for quantified Boolean formulae (QBF). A QBF is a closed formula of the form $\phi = Q_1 x_1 Q_1 x_2 \ldots Q_n x_n \quad \psi(x_1, x_2, \ldots, x_n)$ where $Q_i \in \{\exists, \forall\}$ and ψ is a quantifier-free formula of propositional logic. A classical recursive algorithm for solving Q-SAT is:

$$\begin{aligned}
\mathsf{qsat}(\exists x\ \phi) &= \mathsf{qsat}(\phi[x \leftarrow \mathsf{false}]) \vee \mathsf{qsat}(\phi[x \leftarrow \mathsf{true}]) \\
\mathsf{qsat}(\forall x\ \phi) &= \mathsf{qsat}(\phi[x \leftarrow \mathsf{false}]) \wedge \mathsf{qsat}(\phi[x \leftarrow \mathsf{true}]) \\
\mathsf{qsat}(\beta) &= \mathsf{eval}(\beta)
\end{aligned}$$

where β is a ground Boolean formula. This is exactly the structure of our construction: each quantified variable splits the computation in 2, $\mathsf{qsat}(\phi[x \leftarrow \mathsf{false}])$ is sent to the left and $\mathsf{qsat}(\phi[x \leftarrow \mathsf{true}])$ to the right, and subsequently the recursively computed results that come back are combined (with $\vee$ for $\exists$ and $\wedge$ for $\forall$) to yield the result for the quantified formula. This process can be viewed as an instance of *Map/Reduce*, where the *Map* phase distributes the combinatorial exploration of all possible valuations across space using a binary decision tree, and the *Reduce* phase collects the results and aggregates them using quantifier-appropriate Boolean operations. Our Q-SAT solver is modularly composed as follows (modules `decide`, `map:sat`, and `reduce:qsat` are described below):
`[([reduce:qsat(`$Q_1 x_1 \ldots Q_n x_n$`)][map:sat(`ψ`)][decide(`n`)][until(`n`+1)])spawn]`

4.1 Setting up the decision tree

For a QBF with n variables, we need 1 level per variable, and then at level $n+1$ we have a ground propositional formula that needs to be evaluated. Thus, the

first module we insert is $\texttt{[until}(n+1)\texttt{]}$ to create $n+1$ levels. We then insert $\texttt{[decide}(n)\texttt{]}$ because we want to use the first n levels as decision points for each variable. This is simply achieved by taking $\texttt{[decide}(n)\texttt{]} = \overrightarrow{\alpha}^n$ (one signal $\overrightarrow{\alpha}$ per level) with the following rules: $\overrightarrow{\overrightarrow{\alpha}}, \mathbf{a} \to \mathbf{x}$ (turning stationary signal into assigning ones) and $\overrightarrow{\overrightarrow{\alpha}}, \mathbf{x} \to \overleftarrow{\alpha}, \mathbf{x}, \overrightarrow{\alpha}$ (splitting the remaining $\overrightarrow{\alpha}$ for next levels).

4.2 Compiling the formula

The intuition is that we want to compile the formula into a form of inverse polish notation to obtain executable code using postfix operators. At level $n+1$ all variables have been decided, and have become $\overrightarrow{\mathbf{t}}$ or $\overrightarrow{\mathbf{f}}$. The ground formula, regarded as an expression tree, can be executed bottom up to compute its truth value: the resulting signal for a subexpression is sent to interact with its parent operator. The formula is represented by a beam of signals: each subformula is represented by a (contiguous) subbeam. A subformula that arrives at level $n+1$ starts evaluating when it hits the stationary $\mathbf{a}$. When its truth value has been computed, it is reflected so that it may eventually collide with the incoming signal of its parent connective.

Compilation. For binary connectives, one argument arrives first, it is evaluated, and its truth value is reflected toward the incoming connective; but, in order to reach it, it must cross the incoming beam for the other argument and not interact with the connectives contained therein. For this reason, with each subexpression, we associate a beam $\overrightarrow{\gamma}^k$ of inhibitors that prevents its resulting truth value from interacting with the first k connectives that it crosses. We write $C[\![\psi]\!]$ for the compilation of ψ into a contribution to the initial configuration, and $\| \psi \|$ for the number of occurrences of connectives in ψ. The following scheme of compilation produces an initial configuration wich has a size —the number of signals— at most quadratic in the size s of the input formula. Clearly, for each node (*i.e.* symbol) of the formula, only a linear number of inhibitor signals $\overrightarrow{\gamma}$ can be added, so $C[\![\psi]\!]$ is composed by at most $\mathcal{O}(s \cdot s) = \mathcal{O}(s^2)$ signals. The compilation is done by induction on the formula:

$$
\begin{aligned}
C[\![\psi]\!] &= C[\![\psi]\!]^0 \\
C[\![\psi_1 \wedge \psi_2]\!]^k &= \overrightarrow{\wedge}\ \overrightarrow{\gamma}^k\ C[\![\psi_1]\!]^0\ C[\![\psi_2]\!]^{\|\psi_1\|} \\
C[\![\psi_1 \vee \psi_2]\!]^k &= \overrightarrow{\vee}\ \overrightarrow{\gamma}^k\ C[\![\psi_1]\!]^0\ C[\![\psi_2]\!]^{\|\psi_1\|} \\
C[\![\neg\psi]\!]^k &= \overrightarrow{\neg}\ \overrightarrow{\gamma}^k\ C[\![\psi]\!] \\
C[\![x_i]\!]^k &= \texttt{[var}(x_i)\texttt{]}\ \overrightarrow{\gamma}^k
\end{aligned}
$$

Variables. We want variable x_i to be decided at level i. This can be achieved using $i-1$ inhibitors. For variable x_i, the idea is to protect $\overrightarrow{\overrightarrow{\mathbf{x}}}$ from being assigned into $\overleftarrow{\mathbf{f}}$ and $\overrightarrow{\mathbf{t}}$ until it reaches the i^{th} level. This is achieved with a stack of $i-1$ signals $\overrightarrow{\overrightarrow{\beta}}$: at each level, the first $\overrightarrow{\overrightarrow{\beta}}$ turns the stationary signal $\mathbf{x}$ into $\mathbf{x}_\circ$(the

non-assigning version of $\mathbf{x}$) and disappears. The following $\overrightarrow{\overrightarrow{\beta}}$ and $\overrightarrow{\overrightarrow{\mathbf{x}}}$ are simply split, $\overrightarrow{\overrightarrow{\mathbf{x}}}$ taking $\mathbf{x}_\circ$ back into $\mathbf{x}$. After the first $i-1$ levels, all the $\overrightarrow{\beta}$ have been consumed so that $\overrightarrow{\overrightarrow{\mathbf{x}}}$ finally collides directly with $\mathbf{x}$ and splits into $\overleftarrow{\mathbf{f}}$ going left and $\overrightarrow{\mathbf{t}}$ going right. The variable x_i is initially coded by: $\texttt{[var}(x_i)\texttt{]} = \overrightarrow{\mathbf{x}}\overrightarrow{\beta}^{i-1}$.

$$\overrightarrow{\overrightarrow{\beta}}, \mathbf{x} \to \mathbf{x}_\circ \qquad\qquad \overrightarrow{\overrightarrow{\beta}}, \mathbf{x}_\circ \to \overleftarrow{\beta}, \mathbf{x}_\circ, \overrightarrow{\beta}$$
$$\overrightarrow{\overrightarrow{\mathbf{x}}}, \mathbf{x} \to \overleftarrow{\mathbf{f}}, \mathbf{x}, \overrightarrow{\mathbf{t}} \qquad\qquad \overrightarrow{\overrightarrow{\mathbf{x}}}, \mathbf{x}_\circ \to \overleftarrow{\mathbf{x}}, \mathbf{x}, \overrightarrow{\mathbf{x}}$$

Table 2. Coding and assigning variables.

Evaluation. When hitting $\mathbf{a}$ at level $n+1$, $\overrightarrow{\overrightarrow{\mathbf{t}}}$ is reflected as $\overleftarrow{\mathbf{T}}$, and $\overrightarrow{\overrightarrow{\mathbf{f}}}$ as $\overleftarrow{\mathbf{F}}$: these are their *activated* versions which can interact with incoming connectives to compute the truth value of the formula according to the rules below (for $\wedge$; other connectives are similar, *cf* Appendix). See Fig. 5(a) for an example.

$$\overrightarrow{\overrightarrow{\mathbf{t}}}, \mathbf{a} \to \overleftarrow{\mathbf{T}}, \mathbf{a} \qquad \overrightarrow{\overrightarrow{\mathbf{f}}}, \mathbf{a} \to \overleftarrow{\mathbf{F}}, \mathbf{a} \qquad \overrightarrow{\overrightarrow{\gamma}}, \mathbf{a} \to \overleftarrow{\gamma_+}, \mathbf{a}$$
$$\overrightarrow{\overrightarrow{\wedge}}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\overrightarrow{\wedge_+}} \qquad \overrightarrow{\overrightarrow{\mathbf{f}_+}}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\overrightarrow{\mathbf{f}}} \qquad \overrightarrow{\overrightarrow{\wedge_+}}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\overrightarrow{\mathbf{t}}}$$
$$\overrightarrow{\overrightarrow{\wedge}}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\overrightarrow{\mathbf{f}_+}} \qquad \overrightarrow{\overrightarrow{\mathbf{f}_+}}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\overrightarrow{\mathbf{f}}} \qquad \overrightarrow{\overrightarrow{\wedge_+}}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\overrightarrow{\mathbf{f}}}$$
$$\overrightarrow{\overrightarrow{\wedge}}, \overleftarrow{\gamma_+} \to \overrightarrow{\overrightarrow{\wedge_\circ}} \qquad \overrightarrow{\overrightarrow{\wedge_\circ}}, \overleftarrow{\mathbf{T}} \to \overleftarrow{\mathbf{T}}, \overrightarrow{\overrightarrow{\wedge}} \qquad \overrightarrow{\overrightarrow{\wedge_\circ}}, \overleftarrow{\mathbf{F}} \to \overleftarrow{\mathbf{F}}, \overrightarrow{\overrightarrow{\wedge}}$$

Table 3. Evaluation rules for $\wedge$ connective.

Storing the results. In order to make the result easily exploitable by the *Reduce* phase, we now store it with a signal $\overrightarrow{\mathbf{s}}$ as the stationary signal at level $n+1$; it replaces $\mathbf{a}$, which becomes a signal $\mathbf{t}$ or $\mathbf{f}$. The complete *Map* phase is implemented by: $\texttt{[map:sat}(\psi)\texttt{]} = \overrightarrow{\mathbf{s}}\, C[\psi]$ (rules are given in Tab. 2 and Tab. 3).

4.3 Aggregating the results

As explained earlier, the results for an existentially (resp. universally) quantified variable must be combined using $\vee$ (resp. $\wedge$).

Setting up the quantifiers. We turn the decision points of the first n levels into quantifier signals. Moreover, at each level, we must also take note of the direction in which the aggregated result must be sent. Thus $\exists_L$ represents an existential quantifier that must send its result to the left.
For this, we set: $\texttt{[reduce:qsat:init}(Q_1x_1 \cdots Q_nx_n)\texttt{]} = \overrightarrow{Q_n} \ldots \overrightarrow{Q_1}$.

$$\mathbf{x}, \overleftarrow{\overrightarrow{\exists}} \to \exists_R \qquad \overrightarrow{\overrightarrow{\exists}}, \mathbf{x} \to \exists_L \qquad \mathbf{x}, \overleftarrow{\overrightarrow{\forall}} \to \forall_R \qquad \overrightarrow{\overrightarrow{\forall}}, \mathbf{x} \to \forall_L$$

Table 4. Setting up the quantifiers and the direction of the results.

Aggregating the results. Actual aggregation is initiated by $\overrightarrow{\mathbf{c}}$ and then executes according to the rules given in Fig 5(b). We just have $\texttt{[reduce:qsat:exec]} = \overrightarrow{\mathbf{c}}$.
The complete *Reduce* phase is implemented by
$\texttt{[reduce:qsat}(Q_1x_1 \cdots Q_nx_n)\texttt{]} = \texttt{[reduce:qsat:exec]}\texttt{[reduce:qsat:init}(Q_1x_1 \cdots Q_nx_n)\texttt{]}$

5 Machines for SAT variants

Similar machines for variants of SAT can be obtained easily, typically by using different modules for the *Reduce* phase. All the details of modules and rules can be found in Appendix B.

ENUM-SAT. Returning all the satisfying assignments for a propositional formula ψ can be achieved easily by adding a module which stores satisfying assignments as stationary beams and which annihilates the non-satisfying ones.

#SAT. Counting the number of satisfying assignments for ψ can be achieved using a module implementing a binary adder with signals.

MAX-SAT. It consists in finding the maximum number of *clauses* that can be satisfied by an assignment. Here we must count (with the previous adder module) the number of satisfied clauses rather than the number of satisfying assignments, and then stack a module for computing the max of two binary numbers.

6 Complexities

As mentioned in Sect. 1, we implement algorithms for satisfiability problems on signal machines in order to investigate the computational power of our abstract geometrical model of computation and to compare it to others. As we shall see, for such comparisons to be meaningful, the way complexity is measured is essential and must be adapted tot he nature of the computing machine.

Since signal machines can be regarded as the extension of cellular automata from discrete to continous time and space, it might seem natural to measure time (resp. space) complexity of a computation using the height (resp. width) of its space-time diagram. But, in our applications to SAT variants, these are bounded and independant of the formula: the *Map* phase is bounded by the fractal, and, by symmetry, so is the *Reduce* phase. Indeed, in general, by an appropriate scaling of the initial configuration, a finite computation could be made as small as desired. Thus, height and width are no longer pertinent measures of complexity.

Instead, we should regard our construction as a massively parallel computational device transforming inputs into outputs. The input is the initial configuration at the bottom of the diagram, and the output is the truth value signal coming out at the top of the whole construction, as seen in Fig. 3 for formula $\exists x_1 \forall x_2 \forall x_3 \ (x_1 \wedge \neg x_2) \vee x_3$ [**] . The transformation is performed in parallel by many threads: a thread here is an ascending path through the diagram from an input to the output, and the operations executed by the thread are the collisions occurring on this path.

[**] All the diagrams used as examples in the paper were generated by Durand-Lose's software, implemented in Java, and corresponds to a run of our Q-SAT solver for the running example.

Formally, we view a space-time diagram as a directed acyclic graph of collisions (vertices) and signals (arcs) oriented according to causality. Time complexity is then defined as the maximal size of a chain of collisions *i.e.* the length of the longest path, and space complexity as the maximal size of an anti-chain *i.e.* the size of the maximal set of signals pairwise un-related. This model-specific measure of time complexity is called *collisions depth*.

For the present construction, if s is the size of the formula and n the number of variables, space complexity is exponential: during evaluation, 2^n independent computations are executed in parallel, each one involving less than s^2 signals, so that the total space complexity is in $\mathcal{O}(s^2 \cdot 2^n)$.

Regarding the time complexity: the initial configuration contains at most $\mathcal{O}(s^2)$ signals (from the compilation process as explained in Sect. 4, other modules adding only a linear number of signals). The primary contribution to the number of collisions along an ascending path comes, at each of the n levels, from the reflected beam crossing the incoming beam. Thus a thread involves $\mathcal{O}(n \cdot s^2)$ collisions, making the collision depth cubic in the size of the formula instead of quadratic for our previous family of machines [7]. So here the measure of the time complexity takes one more polynomial degree (from quadratic to cubic) when we get an algorithm which is independent of the input instead of an instance-dependent one. This gives us an idea of the price for genericity.

7 Conclusion

We showed in this paper that abstract geometrical computation can solve Q-SAT in bounded space and time by means of a single generic signal machine. This is achieved through massive parallelism enabled by a fractal construction that we call the *fractal cloud*. We adapted the Map/Reduce paradigm to this fractal cloud, and described a modular programming approach making it easy to assemble generic machines for SAT variants such as #SAT or MAX-SAT.

As we explained in Sect. 6, time and space are no longer appropriate measures of complexity for geometrical computations. This leads us to propose new definitions of complexity, specific to signal machines : time and space complexities are now defined respectively by the maximal sizes of a chain and an anti-chain, when the diagram is regarded as a directed acyclic graph. Time complexity thus defined is called *collision depth* and is cubic for the construction given here.

Although the model is purely theoretical and has no ambition to be physically realizable, it is a significant and distinguishing aspect of signal machines that they solve satifiability problems while adhering to major principles of modern physics —finite density and speed of information, causality— that are typically not considered by other unconventional models of computation. They do not, however, respect the quantization hypothesis, nor the uncertainty principle.

We are now furthering our research along two axes. First, the design and applications of other fractal structures for modular programming with fractal parallelism. Second, the investigation of computational complexity classes, both classical and model-specific for abstract geometrical computation.

References

1. Aaronson, S.: NP-complete problems and physical reality. SIGACT News. 36(1), 30–52 (2005)
2. Asarin, E., Maler, O.: Achilles and the Tortoise climbing up the arithmetical hierarchy. In: 15th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '95). pp. 471–483. No. 1026 in LNCS (1995)
3. Cook, M.: Universality in elementary cellular automata. Compl. Syst. 15(1), 1–40 (2004)
4. Cook, S.: The complexity of theorem proving procedures. In: 3rd Symp. on Theory of Computing (STOC '71). pp. 151–158. ACM (1971)
5. Dean, J., Ghemawat, S., Inc., G.: Map/Reduce: simplified data processing on large clusters. In: 6th Symp. on Operating Systems Design & Implementation (OSDI' 04). USENIX Association (2004)
6. Duchier, D., Durand-Lose, J., Senot, M.: Fractal parallelism: Solving SAT in bounded space and time. In: Cheong, O., Chwa, K.Y., Park, K. (eds.) 21st Int. Symp. on Algorithms and Computation (ISAAC '10). pp. 279–290. LNCS 6506, Springer (2010)
7. Duchier, D., Durand-Lose, J., Senot, M.: Massively parallel automata in Euclidean space-time. In: IEEE 4th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW '10). pp. 104–109. IEEE Computer Society (2010)
8. Durand-Lose, J.: Forecasting black holes in abstract geometrical computation is highly unpredictable. In: Cai, J.Y., Cooper, S.B., Li, A. (eds.) 3rd Int. Conf. on Theory and Applications of Models of Computations (TAMC '06). pp. 644–653. No. 3959 in LNCS, Springer (2006)
9. Durand-Lose, J.: Abstract geometrical computation and computable analysis. In: Costa, J., Dershowitz, N. (eds.) 8th Int. Conf. on Unconventional Computation 2009 (UC '09). pp. 158–167. LNCS 5715, Springer (2009)
10. Durand-Lose, J.: Abstract geometrical computation 4: small Turing universal signal machines. Theoret. Comp. Sci. 412, 57–67 (2011)
11. Fischer, P.: Generation of primes by a one-dimensional real-time iterative array. Jour. ACM 12(3), 388–394 (1965)
12. Goliaei, S., Jalili, S.: An optical solution to the 3-SAT problem using wavelength based selectors. The Journal of Supercomputing pp. 1–10 (2010)
13. Huckenbeck, U.: Euclidian geometry in terms of automata theory. Theoret. Comp. Sci. 68(1), 71–87 (1989)
14. Jacopini, G., Sontacchi, G.: Reversible parallel computation: an evolving space-model. Theoret. Comp. Sci. 73(1), 1–46 (1990)
15. Jeandel, E., Vanier, P.: π_0^1 sets and tilings. In: Ogihara, M., Tarui, J. (eds.) 8th Int. Conf. on Theory and Applications of Models of Compuation (TAMC '11). LNCS, vol. 6648, pp. 230–239. Springer (2011)
16. Mackie, I.: A visual model of computation. In: Kratochvíl, J., Li, A., Fiala, J., Kolman, P. (eds.) 7th Int. Conf. on Theory and Applications of Models of Computation (TAMC '10). LNCS, vol. 6108, pp. 350–360. Springer (2010)
17. Margenstern, M., Morita, K.: NP problems are tractable in the space of cellular automata in the hyperbolic plane. Theoret. Comp. Sci. 259(1–2), 99–128 (2001)
18. Naughton, T., Woods, D.: An optical model of computation. Theoret. Comput. Sci. 334(1-3), 227–258 (2005)
19. Păun, G.: P-systems with active membranes: Attacking NP-Complete problems. Jour. of Automata, Languages and Combinatorics 6(1), 75–90 (2001)

20. Stockmeyer, L., Meyer, A.: Word problems requiring exponential time. In: 5th ACM Symp. on Theory of Computing (STOC '73). vol. 16, pp. 1–9 (1973)

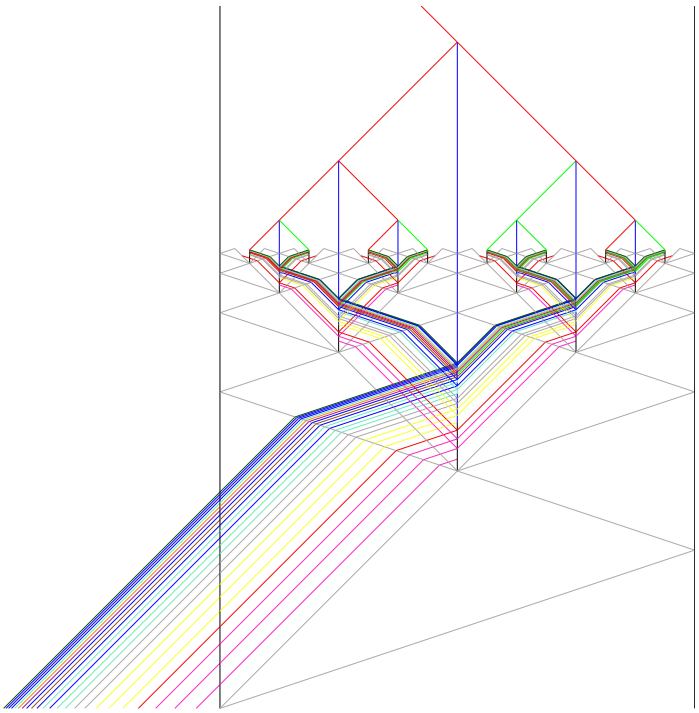
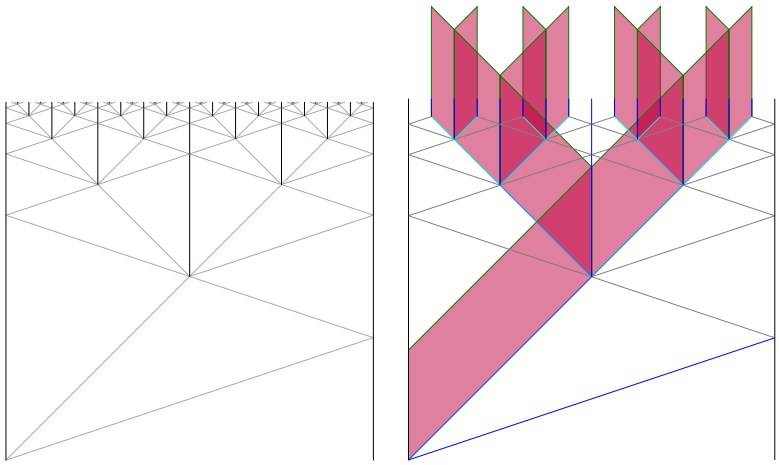
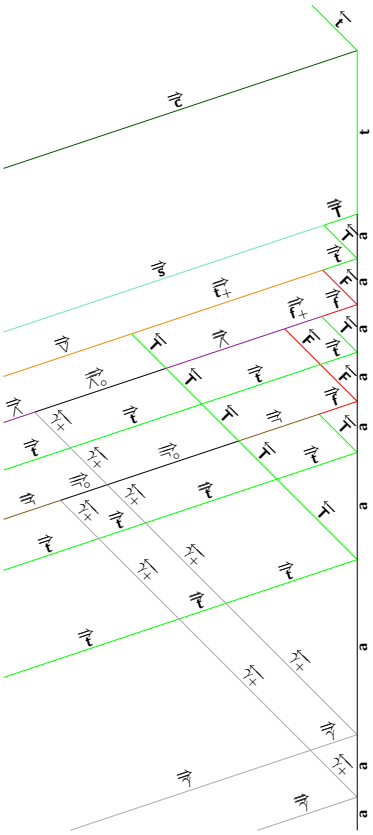


Fig. 3. The whole diagram.



(a) Constructing the fractal cloud (b) Distributing a computation

Fig. 4. Computing in the fractal cloud



(a) Evaluation
case $x_1 = x_2 = x_3 = \mathbf{t}$

$$[(\overrightarrow{\mathbf{c}}\ \overrightarrow{\forall}\ \overrightarrow{\forall}\ \overrightarrow{\exists}\ \overrightarrow{\mathbf{s}}\ \overrightarrow{\vee}\ \overrightarrow{\wedge}\ \overrightarrow{\mathbf{x}}\ \overrightarrow{\neg}\ \overrightarrow{\mathbf{x}}\ \overrightarrow{\beta}\ \overrightarrow{\mathbf{x}}\ \overrightarrow{\beta}\ \overrightarrow{\beta}\ \overrightarrow{\gamma}\ \overrightarrow{\gamma}\ \overrightarrow{\alpha}\ \overrightarrow{\alpha}\ \overrightarrow{\alpha}\ \overrightarrow{\mathbf{z}}\ \overrightarrow{\zeta}\ \overrightarrow{\zeta}\ \overrightarrow{\zeta}\ )\mathtt{spawn}]$$

(c) Initial configuration

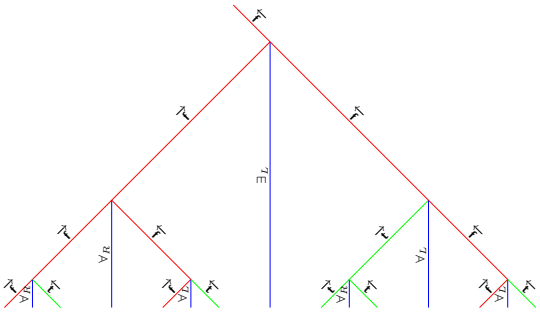
$$\overrightarrow{\mathbf{c}}, \mathbf{t} \to \overleftarrow{\mathbf{t}} \qquad \overrightarrow{\mathbf{c}}, \mathbf{f} \to \overleftarrow{\mathbf{f}}$$

$$\overrightarrow{\mathbf{t}}, \exists_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}} \qquad \overrightarrow{\mathbf{t}}, \forall_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}}$$

$$\overrightarrow{\mathbf{t}}, \exists_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{t}} \qquad \overrightarrow{\mathbf{t}}, \forall_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}}$$

$$\overrightarrow{\mathbf{f}}, \exists_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}} \qquad \overrightarrow{\mathbf{f}}, \forall_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{f}}$$

$$\overrightarrow{\mathbf{f}}, \exists_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}} \qquad \overrightarrow{\mathbf{f}}, \forall_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}}$$



(b) Aggregation

Fig. 5. Example $\exists x_1 \forall x_2 \forall x_3\ (x_1 \wedge \neg x_2) \vee x_3$

A Appendix: details of the modules

In the following, we generally define the collision rules only for one side, as the corresponding rules for the other side can be deduced by symetry. We give here the rules for all modules: first the modules for building and stopping the fractal tree, then the modules for setting and using the tree for satisfiability problems and finally the modules specific to each variants of SAT.

All the diagrams of the paper were generated by Durand-Lose's software, implemented in Java, and corresponds to a run of our Q-SAT solver for the formula:

$$\phi = \exists x_1 \forall x_2 \forall x_3 \ (x_1 \wedge \neg x_2) \vee x_3 \ .$$

A.1 The fractal cloud

Constructing the fractal: `[start]` $= \overrightarrow{\mathsf{a}}, \overrightarrow{\overrightarrow{\mathsf{a}}}$
The following rules on the left correspond to the bounce of $\overrightarrow{\overrightarrow{\mathsf{a}}}$ and $\overleftarrow{\overleftarrow{\mathsf{a}}}$ on the wall w. Rules on the right are the step of induction for starting the next level: the initial signals $\overrightarrow{\mathsf{a}}$ and $\overrightarrow{\overrightarrow{\mathsf{a}}}$ are duplicated on the right and the left, and the stationary signal a is created exactly at the middle of the previous stage. The result is given by Fig. 6.

$$\mathsf{w}, \overleftarrow{\overleftarrow{\mathsf{a}}} \to \mathsf{w}, \overrightarrow{\overrightarrow{\mathsf{a}}} \qquad \overrightarrow{\mathsf{a}}, \overleftarrow{\overleftarrow{\mathsf{a}}} \to \overleftarrow{\overleftarrow{\mathsf{a}}}, \overleftarrow{\mathsf{a}}, \mathsf{a}, \overrightarrow{\mathsf{a}}, \overrightarrow{\overrightarrow{\mathsf{a}}}$$
$$\overrightarrow{\overrightarrow{\mathsf{a}}}, \mathsf{w} \to \overleftarrow{\overleftarrow{\mathsf{a}}}, \mathsf{w} \qquad \overrightarrow{\overrightarrow{\mathsf{a}}}, \overleftarrow{\mathsf{a}} \to \overleftarrow{\overleftarrow{\mathsf{a}}}, \overleftarrow{\mathsf{a}}, \mathsf{a}, \overrightarrow{\mathsf{a}}, \overrightarrow{\overrightarrow{\mathsf{a}}}$$

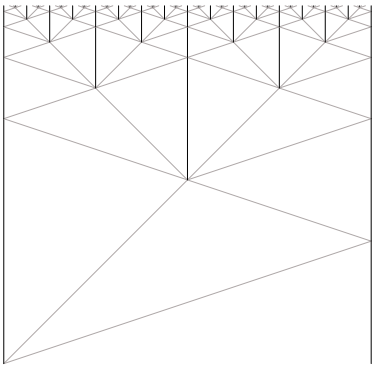


Fig. 6. The fractal tree.

Stopping the fractal: `[until(`$n+1$`)]` $= \overrightarrow{\overrightarrow{\mathsf{z}}}\overrightarrow{\zeta}^n$

The role of this module is to stop the fractal after $(n+1)$ levels — n levels for assigning the n variables and 1 level for the evaluation of the ground formula. For this, we use a stack of n signals $\overrightarrow{\zeta}$ and one signal $\overrightarrow{\overrightarrow{\mathsf{z}}}$. The $\overrightarrow{\zeta}$ signals are used both as a counter, one signal being killed at each level, and as inhibitors to the effect of $\overrightarrow{\overrightarrow{\mathsf{z}}}$. After

n levels, only $\overrightarrow{\mathbf{z}}$ remains and can stop the construction of the fractal at level $n+1$. Here are the corresponding rules:

$$\overrightarrow{\zeta}, \overleftarrow{\mathbf{a}} \to \overleftarrow{\mathbf{a}_\circ}, \overrightarrow{\zeta} \qquad \overrightarrow{\zeta}, \mathbf{a} \to \mathbf{a}_\circ \qquad \overrightarrow{\zeta}, \mathbf{a}_\circ \to \overleftarrow{\zeta}, \mathbf{a}_\circ, \overrightarrow{\zeta}$$
$$\overrightarrow{\mathbf{z}}, \overleftarrow{\mathbf{a}} \to \overleftarrow{\mathbf{a}_\circ}, \overrightarrow{\mathbf{z}} \qquad \overrightarrow{\mathbf{z}}, \mathbf{a}_\circ \to \overleftarrow{\mathbf{a}}, \overrightarrow{\mathbf{z}} \qquad \overrightarrow{\mathbf{z}}, \mathbf{a}_\circ \to \overleftarrow{\mathbf{z}}, \mathbf{a}, \overrightarrow{\mathbf{z}}$$
$$\overrightarrow{\mathbf{z}}, \mathbf{a} \to \mathbf{a}, \overrightarrow{\mathbf{z}} \qquad \overrightarrow{\mathbf{z}}, \overrightarrow{\mathbf{a}} \to \overrightarrow{\mathbf{a}_\circ} \qquad \overrightarrow{\mathbf{a}}, \overleftarrow{\mathbf{a}_\circ} \to$$

The lens effect: The general idea is that any signal $\overrightarrow{\mathbf{i}}$ is accelerated by $\overleftarrow{\mathbf{a}}$ and decelerated and split by any stationary signal $\mathbf{s}$. There are some special cases to handle for the deceleration and the split, some particular signals being stopped at this moment. But in all cases, signals are always accelerated by $\overleftarrow{\mathbf{a}}$. Figure 7 zooms on a split and illustrates the lens effect on the beam as well as the assignment of the top-most $\overrightarrow{\mathbf{x}}$.

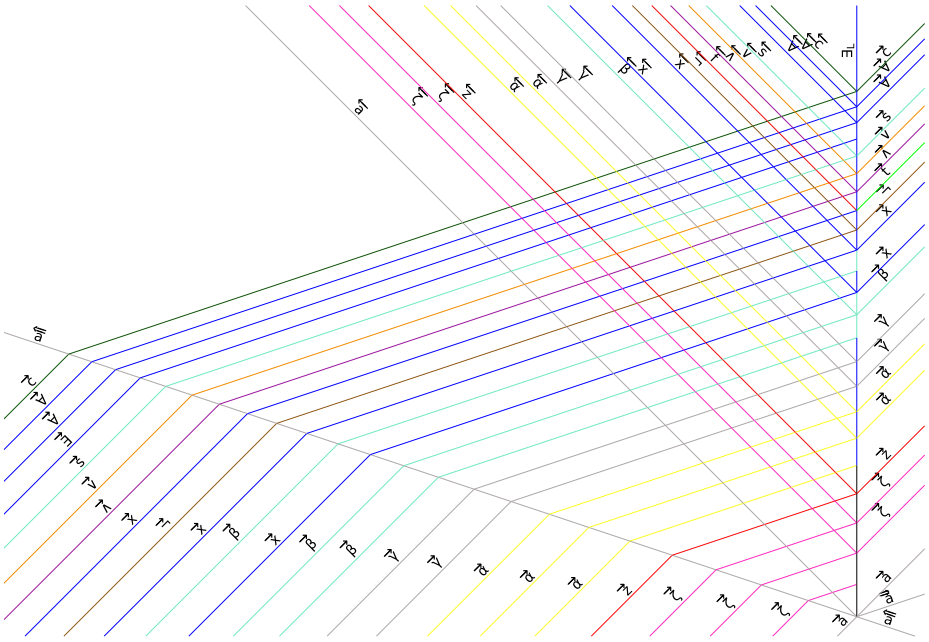


Fig. 7. Split and lens effect at the first level.

General case: for any stationary signal $\mathbf{s}$ (more exactly, $\mathbf{s}$ is either $\mathbf{a}$, $\mathbf{a}_\circ$, $\mathbf{x}$, $\mathbf{x}_\circ$, $\exists_L$, $\exists_R$, $\forall_L$ or $\forall_R$) and for any signal $\overrightarrow{\mathbf{i}}$ distinct of $\overrightarrow{\zeta}$ and $\overrightarrow{\mathbf{z}}$, we have:

$$\overrightarrow{\mathbf{i}}, \overleftarrow{\mathbf{a}} \to \overleftarrow{\mathbf{a}}, \overrightarrow{\mathbf{i}} \qquad\qquad \overrightarrow{\mathbf{i}}, \mathbf{s} \to \overleftarrow{\mathbf{i}}, \mathbf{s}, \overrightarrow{\mathbf{i}}$$

Case of $\overrightarrow{\zeta}$ and $\overrightarrow{\mathbf{z}}$: the rules for applying the lens effect to $\overrightarrow{\zeta}$ and $\overrightarrow{\mathbf{z}}$ are given previously in §.Stopping the fractal. The stationary signal involved here is $\mathbf{a}$, which kills the first $\overrightarrow{\zeta}$, becomes $\mathbf{a}_\circ$ and splits and decelerates the next signals $\overrightarrow{\zeta}$ and $\overrightarrow{\mathbf{z}}$, and then becomes $\mathbf{a}$ again.

Case of $\overrightarrow{\alpha}$: the first $\overrightarrow{\alpha}$ is stopped by the stationary signal a, which becomes x and decelerates and splits the next coming $\overrightarrow{\alpha}$.

$$\overrightarrow{\overrightarrow{\alpha}}, \mathsf{a} \to \mathsf{x} \qquad\qquad \overrightarrow{\overrightarrow{\alpha}}, \mathsf{x} \to \overleftarrow{\alpha}, \mathsf{x}, \overrightarrow{\alpha}$$

Case of quantifiers signals: the first quantifier signal, $\overrightarrow{\overrightarrow{\forall}}$ or $\overrightarrow{\overrightarrow{\exists}}$, colliding with a stationary x is stopped and turns x into $\forall_D$ or $\exists_D$ ($D \in \{R, L\}$). This is achieved by the rules:

$$\mathsf{x}, \overleftarrow{\overrightarrow{\exists}} \to \exists_R \qquad \overrightarrow{\overrightarrow{\exists}}, \mathsf{x} \to \exists_L \qquad \mathsf{x}, \overleftarrow{\overrightarrow{\forall}} \to \forall_R \qquad \overrightarrow{\overrightarrow{\forall}}, \mathsf{x} \to \forall_L$$

The next quantifier signals are decelerated and split by the new stationary signal $\forall_D$ or $\exists_D$. In the following rules, we have $D \in \{R, L\}$:

$$\overrightarrow{\overrightarrow{\exists}}, \exists_D \to \overleftarrow{\exists}, \exists_D, \overrightarrow{\exists} \qquad\qquad \overrightarrow{\overrightarrow{\exists}}, \forall_D \to \overleftarrow{\exists}, \forall_D, \overrightarrow{\exists}$$
$$\overrightarrow{\overrightarrow{\forall}}, \exists_D \to \overleftarrow{\forall}, \exists_D, \overrightarrow{\forall} \qquad\qquad \overrightarrow{\overrightarrow{\forall}}, \forall_D \to \overleftarrow{\forall}, \forall_D, \overrightarrow{\forall}$$

A.2 The tree for satisfiabilty problems

Activation of the decision tree: $\texttt{[init}(n)\texttt{]} = \overrightarrow{\alpha}^n$

$$\overrightarrow{\overrightarrow{\alpha}}, \mathsf{a} \to \mathsf{x} \qquad\qquad \overrightarrow{\overrightarrow{\alpha}}, \mathsf{x} \to \overleftarrow{\alpha}, \mathsf{x}, \overrightarrow{\alpha}$$

Representation of a variable: $\texttt{[var}(x_i)\texttt{]} = \overrightarrow{\mathtt{x}}\,\overrightarrow{\beta}^{i-1}$

As we explained in the paper, the variable x_i is represented by a stack of i signals: one signal $\overrightarrow{\mathtt{x}}$ and $i-1$ signals $\overrightarrow{\beta}$. The role of the signals $\overrightarrow{\beta}$ is to protect $\overrightarrow{\mathtt{x}}$ from being assigned before the i^{th} level. The first signal $\overrightarrow{\beta}$ of the stack is stopped at the next split, so that $i-1$ signals have disappeared just before the i^{th} stage. This is illustrated in Fig. 7 and modelised by the following rules:

$$\overrightarrow{\overrightarrow{\beta}}, \mathsf{x} \to \mathsf{x}_\circ \qquad\qquad \overrightarrow{\overrightarrow{\beta}}, \mathsf{x}_\circ \to \overleftarrow{\beta}, \mathsf{x}_\circ, \overrightarrow{\beta}$$
$$\overrightarrow{\overrightarrow{\mathtt{x}}}, \mathsf{x} \to \overleftarrow{\mathsf{f}}, \mathsf{x}, \overrightarrow{\mathsf{t}} \qquad\qquad \overrightarrow{\overrightarrow{\mathtt{x}}}, \mathsf{x}_\circ \to \overleftarrow{\mathtt{x}}, \mathsf{x}, \overrightarrow{\mathtt{x}}$$

Compilation of the formula: We propose here a recursive algorithm which takes as input an unquantified formula — a SAT-formula — and outputs the part of the initial configuration corresponding to the formula. In the following schemes of compilation, $\| \phi \|$ designates the number of occurences of Boolean connectives in formula ϕ.

$$\begin{aligned}
C[\phi] &= C[\phi]^0 \\
C[\phi_1 \wedge \phi_2]^k &= \overrightarrow{\wedge}\ \overrightarrow{\gamma}^k\ C[\phi_1]^0\ C[\phi_2]^{\|\phi_1\|} \\
C[\phi_1 \vee \phi_2]^k &= \overrightarrow{\vee}\ \overrightarrow{\gamma}^k\ C[\phi_1]^0\ C[\phi_2]^{\|\phi_1\|} \\
C[\neg\phi]^k &= \overrightarrow{\neg}\ \overrightarrow{\gamma}^k\ C[\phi] \\
C[x_i]^k &= \overrightarrow{\mathtt{x}}\ \overrightarrow{\beta}^{i-1}\ \overrightarrow{\gamma}^k
\end{aligned}$$

Evaluation: The rules for the evaluation follow the classical Boolean operations. We explained earlier that some inhibiting signals — the $\overrightarrow{\gamma}$ — are needed to allow the result of the first evaluated argument of a binary connective to traverse the beam of the other, as yet unevaluated, argument without reacting with the connectives contained therein, and only interact with its actual syntactical parent connective. Figure 8 displays the evaluation of the formula $\phi = \exists x_1 \forall x_2 \forall x_3\ (x_1 \wedge \neg x_2) \vee x_3$ for the case $x_1 = x_2 = x_3 = \mathbf{t}$.

$$\overrightarrow{\mathbf{t}}, \mathbf{a} \to \overleftarrow{\mathbf{T}}, \mathbf{a} \qquad \overrightarrow{\mathbf{f}}, \mathbf{a} \to \overleftarrow{\mathbf{F}}, \mathbf{a} \qquad \overrightarrow{\gamma}, \mathbf{a} \to \overleftarrow{\gamma_+}, \mathbf{a}$$

$$\begin{array}{lll}
\overrightarrow{\wedge}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\wedge_+} & \overrightarrow{\mathbf{f}_+}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{f}} & \overrightarrow{\wedge_+}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{t}} \\
\overrightarrow{\wedge}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{f}_+} & \overrightarrow{\mathbf{f}_+}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{f}} & \overrightarrow{\wedge_+}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{f}} \\
\overrightarrow{\vee}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{t}_+} & \overrightarrow{\mathbf{t}_+}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{t}} & \overrightarrow{\vee_+}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{t}} \\
\overrightarrow{\vee}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\vee_+} & \overrightarrow{\mathbf{t}_+}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{t}} & \overrightarrow{\vee_+}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{f}} \\
\overrightarrow{\neg}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{f}} & & \\
\overrightarrow{\neg}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{t}} & &
\end{array}$$

$$\overrightarrow{\wedge}, \overleftarrow{\gamma_+} \to \overrightarrow{\wedge_\circ} \qquad \overrightarrow{\vee}, \overleftarrow{\gamma_+} \to \overrightarrow{\vee_\circ} \qquad \overrightarrow{\neg}, \overleftarrow{\gamma_+} \to \overrightarrow{\neg_\circ}$$

$$\begin{array}{lll}
\overrightarrow{\wedge_\circ}, \overleftarrow{\mathbf{T}} \to \overleftarrow{\mathbf{T}}, \overrightarrow{\wedge} & \overrightarrow{\vee_\circ}, \overleftarrow{\mathbf{T}} \to \overleftarrow{\mathbf{T}}, \overrightarrow{\vee} & \overrightarrow{\neg_\circ}, \overleftarrow{\mathbf{T}} \to \overleftarrow{\mathbf{T}}, \overrightarrow{\neg} \\
\overrightarrow{\wedge_\circ}, \overleftarrow{\mathbf{F}} \to \overleftarrow{\mathbf{F}}, \overrightarrow{\wedge} & \overrightarrow{\vee_\circ}, \overleftarrow{\mathbf{F}} \to \overleftarrow{\mathbf{F}}, \overrightarrow{\vee} & \overrightarrow{\neg_\circ}, \overleftarrow{\mathbf{F}} \to \overleftarrow{\mathbf{F}}, \overrightarrow{\neg}
\end{array}$$

Storing the results: $\texttt{[store]} = \overrightarrow{\mathbf{s}}$

$$\begin{array}{ll}
\overrightarrow{\mathbf{s}}, \overleftarrow{\mathbf{T}} \to \overrightarrow{\mathbf{T}} & \overrightarrow{\mathbf{T}}, \mathbf{a} \to \mathbf{t} \\
\overrightarrow{\mathbf{s}}, \overleftarrow{\mathbf{F}} \to \overrightarrow{\mathbf{F}} & \overrightarrow{\mathbf{F}}, \mathbf{a} \to \mathbf{f}
\end{array}$$

B Appendix: modularity and satifiability problems

B.1 Q-SAT

To proceed to the aggregation process, we join the results coming from right and left two-by-two. This is done with a stationary signal indicating the type of operation to perform — a conjunction for $\forall$ and a disjunction for $\exists$ — and the direction of the resulting signal — left or right. The whole process is displayed in Fig. 9.

Setting up the reduce stage:

$$\texttt{[reduce:qsat:init}(Q_1 x_1 \cdots Q_n x_n)\texttt{]} = \overrightarrow{Q_n} \ldots \overrightarrow{Q_1}$$

$$\begin{array}{ll}
\mathbf{x}, \overleftarrow{\exists} \to \exists_R & \overrightarrow{\exists}, \mathbf{x} \to \exists_L \\
\mathbf{x}, \overleftarrow{\forall} \to \forall_R & \overrightarrow{\forall}, \mathbf{x} \to \forall_L
\end{array}$$

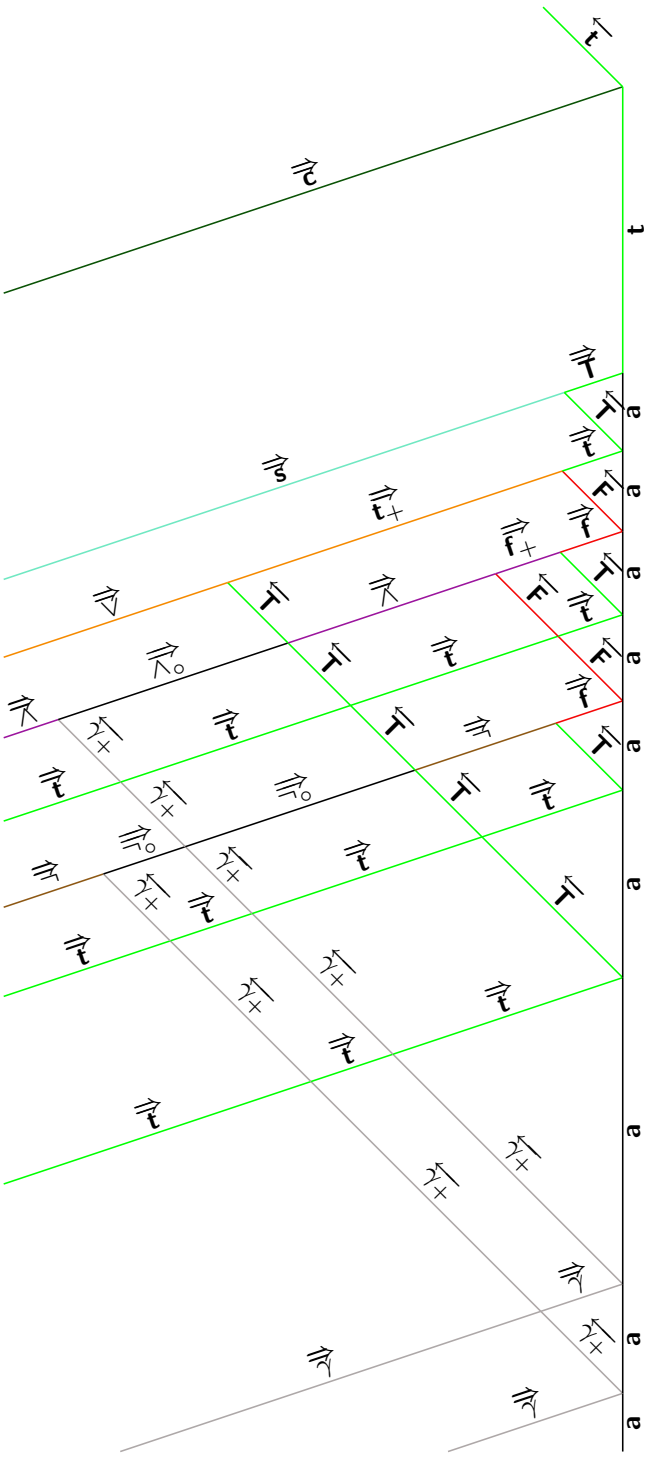


Fig. 8. Evaluation for $x_1 = x_2 = x_3 = \mathbf{t}$ in $\exists x_1 \forall x_2 \forall x_3\ (x_1 \wedge \neg x_2) \vee x_3$.

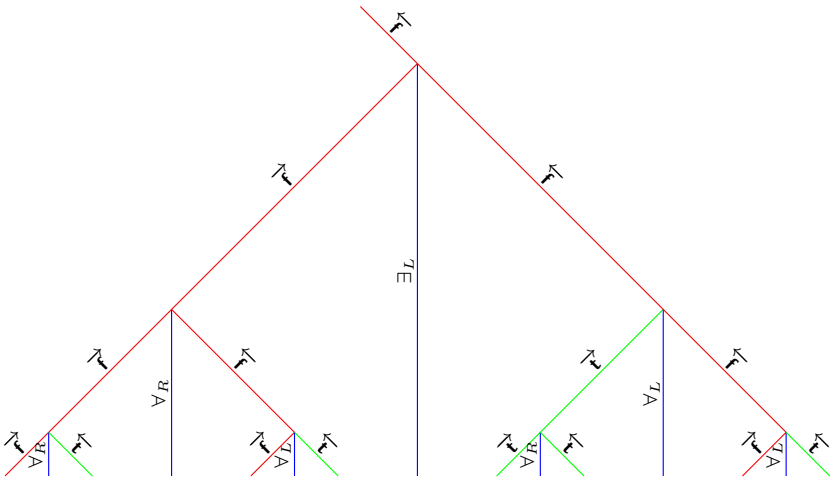


Fig. 9. Aggregation process.

Executing the reduce stage: `[reduce:qsat:exec]` $= \overrightarrow{\mathbf{c}}$

Initiation:

$$\overrightarrow{\mathbf{c}}, \mathbf{t} \to \overleftarrow{\mathbf{t}} \qquad\qquad \overrightarrow{\mathbf{c}}, \mathbf{f} \to \overleftarrow{\mathbf{f}}$$

Performing the disjunction:

$$\overrightarrow{\mathbf{t}}, \exists_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{t}}, \exists_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{f}}, \exists_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{f}}, \exists_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}}$$
$$\overrightarrow{\mathbf{t}}, \exists_R, \overleftarrow{\mathbf{t}} \to \overrightarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{t}}, \exists_R, \overleftarrow{\mathbf{f}} \to \overrightarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{f}}, \exists_R, \overleftarrow{\mathbf{t}} \to \overrightarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{f}}, \exists_R, \overleftarrow{\mathbf{f}} \to \overrightarrow{\mathbf{f}}$$

Performing the conjunction:

$$\overrightarrow{\mathbf{t}}, \forall_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{t}}, \forall_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}} \quad \overrightarrow{\mathbf{f}}, \forall_L, \overleftarrow{\mathbf{t}} \to \overleftarrow{\mathbf{f}} \quad \overrightarrow{\mathbf{f}}, \forall_L, \overleftarrow{\mathbf{f}} \to \overleftarrow{\mathbf{f}}$$
$$\overrightarrow{\mathbf{t}}, \forall_R, \overrightarrow{\mathbf{t}} \to \overrightarrow{\mathbf{t}} \quad \overrightarrow{\mathbf{t}}, \forall_R, \overrightarrow{\mathbf{f}} \to \overrightarrow{\mathbf{f}} \quad \overrightarrow{\mathbf{f}}, \forall_R, \overrightarrow{\mathbf{t}} \to \overrightarrow{\mathbf{f}} \quad \overrightarrow{\mathbf{f}}, \forall_R, \overrightarrow{\mathbf{f}} \to \overrightarrow{\mathbf{f}}$$

Putting all the modules together, we obtain for the running example the initial configuration shown by Fig. 10.

The global construction is displayed by Fig. 11

Fig. 10. Initial configuration.

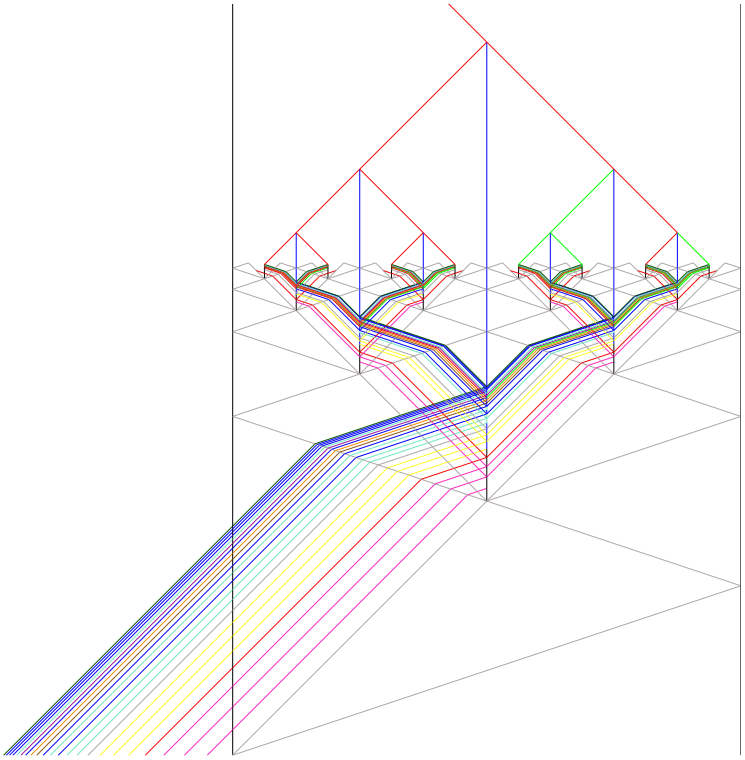


Fig. 11. The whole diagram.

B.2 #SAT

#SAT is the problem of counting the number of solutions of SAT. We recall that this problem is complete for the class #P *i.e.* the class of NP-problems for which their solutions can be counted is polynomial time.

To solve #SAT, as a SAT-formula is a special instance of a Q-SAT formula in which all quantifiers are existential, we can use our Q-SAT solver and we add a special module for counting the truth evalution of the formula during the aggregation process. The counting is performed by a binary adder.

Setting up the reduce stage: `[reduce:#sat:init]` $= \overrightarrow{+}^n$

$$\overrightarrow{+}, \mathbf{x} \to +^0_L \qquad\qquad \mathbf{x}, \overleftarrow{+} \to +^0_R$$

Executing the reduce stage: `[reduce:#sat:exec]` $= \overrightarrow{\epsilon_\circ}\overrightarrow{\delta_\circ}\overrightarrow{0_\circ}$

$$\begin{array}{lll} \overrightarrow{\delta_\circ}, \mathbf{t} \to \overleftarrow{\delta} & \overrightarrow{\epsilon_\circ}, \mathbf{t} \to \overleftarrow{\epsilon} & \overrightarrow{0_\circ}, \mathbf{t} \to \overleftarrow{1} \\ \overrightarrow{\delta_\circ}, \mathbf{f} \to \overleftarrow{\delta} & \overrightarrow{\epsilon_\circ}, \mathbf{f} \to \overleftarrow{\epsilon} & \overrightarrow{0_\circ}, \mathbf{f} \to \overleftarrow{0} \end{array}$$

Rules for the binary adder (for the R subscript, the rules for the L are similar, but with the output going to the left):

$$\overrightarrow{0}, +^0_R, \overleftarrow{0} \to +^0_R, \overrightarrow{0} \qquad \overrightarrow{0}, +^1_R, \overleftarrow{0} \to +^0_R, \overrightarrow{1}$$
$$\overrightarrow{1}, +^0_R, \overleftarrow{0} \to +^0_R, \overrightarrow{1} \qquad \overrightarrow{1}, +^1_R, \overleftarrow{0} \to +^1_R, \overrightarrow{0}$$
$$\overrightarrow{0}, +^0_R, \overleftarrow{1} \to +^0_R, \overrightarrow{1} \qquad \overrightarrow{0}, +^1_R, \overleftarrow{1} \to +^1_R, \overrightarrow{0}$$
$$\overrightarrow{1}, +^0_R, \overleftarrow{1} \to +^1_R, \overrightarrow{0} \qquad \overrightarrow{1}, +^1_R, \overleftarrow{1} \to +^1_R, \overrightarrow{1}$$

$$\overrightarrow{0}, +^0_R, \overleftarrow{\delta} \to +^0_R, \overrightarrow{0} \qquad \overrightarrow{0}, +^1_R, \overleftarrow{\delta} \to +^0_R, \overrightarrow{1}$$
$$\overrightarrow{1}, +^0_R, \overleftarrow{\delta} \to +^0_R, \overrightarrow{1} \qquad \overrightarrow{1}, +^1_R, \overleftarrow{\delta} \to +^1_R, \overrightarrow{0}$$

$$\overrightarrow{0}, +^0_R \to +^0_R, \overrightarrow{0} \qquad \overrightarrow{0}, +^1_R \to +^0_R, \overrightarrow{1}$$
$$\overrightarrow{1}, +^0_R \to +^0_R, \overrightarrow{1} \qquad \overrightarrow{1}, +^1_R \to +^1_R, \overrightarrow{0}$$

$$\overrightarrow{\delta}, +^0_R, \overleftarrow{\delta} \to +^0_R, \overrightarrow{\delta} \qquad \overrightarrow{\delta}, +^1_R, \overleftarrow{\delta} \to +^0_R, \overrightarrow{1}, \overrightarrow{\delta}$$
$$\overrightarrow{\delta}, +^0_R \to +^0_R, \overrightarrow{\delta} \qquad \overrightarrow{\delta}, +^1_R \to +^0_R, \overrightarrow{1}, \overrightarrow{\delta}$$

$$\overrightarrow{\delta}, \overleftarrow{\epsilon} \to \overleftarrow{\epsilon}, \overrightarrow{\delta} \qquad \overrightarrow{\epsilon}, +^0_R \to \overrightarrow{\epsilon}$$

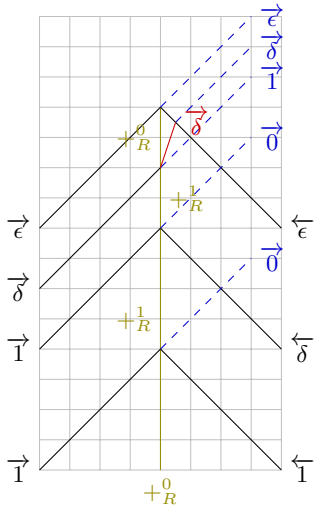


Fig. 12. Computing $3 + 1$

B.3 ENUM-SAT

ENUM-SAT is the problem of enumerating all the solutions for an instance of SAT: we want to know *all* the truth assignements of variables for which the formula is satisfiable. We can also consider a particular case of ENUM-SAT: the problem ONESOL-SAT, which consists in returning *only one* valuation satifying the formula, when the formula is satisfiable.

Reduce stage: $\texttt{[reduce:enumsat}(x_1 \dots x_n)\texttt{]} = \overrightarrow{\mathbf{v}}\,\texttt{[var}(x_1)\texttt{]} \dots \texttt{[var}(x_n)\texttt{]}\,\overrightarrow{\mathbf{v}}$

$$\overrightarrow{\mathbf{v}}, \mathbf{t} \to \overleftarrow{\mathbf{v}_+}, \mathbf{v} \qquad \overrightarrow{\mathbf{t}}, \overleftarrow{\mathbf{v}_+} \to \overleftarrow{\mathbf{v}_+}, \mathbf{t} \qquad \overrightarrow{\mathbf{v}}, \mathbf{f} \to \overleftarrow{\mathbf{v}_\circ} \qquad \overrightarrow{\mathbf{t}}, \overleftarrow{\mathbf{v}_\circ} \to \overleftarrow{\mathbf{v}_\circ}$$
$$\overrightarrow{\mathbf{v}}, \overleftarrow{\mathbf{v}_+} \to \mathbf{v} \qquad \overrightarrow{\mathbf{f}}, \overleftarrow{\mathbf{v}_+} \to \overleftarrow{\mathbf{v}_+}, \mathbf{f} \qquad \overrightarrow{\mathbf{v}}, \overleftarrow{\mathbf{v}_\circ} \to \emptyset \qquad \overrightarrow{\mathbf{f}}, \overleftarrow{\mathbf{v}_\circ} \to \overleftarrow{\mathbf{v}_\circ}$$

B.4 MAX-SAT

The problem MAX-SAT consists in, given k SAT-formulae, finding the maximum number of formulae satisfiable by the same valuation. This problem is NP-hard and is complete for the class APX — the class of problems approximable in polynomial time with a constant factor of approximation. The problem MAX-SAT can be extended by returning the valuation of variables that satisfies the greater number of formulae amoung the k ones.

We do not give the corresponding rules for solving MAX-SAT, we just describe the concerned modules. Each formula amoung the k formulae given in the input is compiled by the same method used previously, and the resulting arrangement of signals for each formula are placed end-to-end. This results in a beam of formulae composed by k subbeam, one for each formula. The evaluation process is then the same as seen previously.

To compare the number of satisfiable formulae for each valuation, we used the binary adder introduced for #SAT, that we combine with a module comparing two binary number. The reduce phase follows the same idea that for the other variants, except that after comparing two-by-two the number of formulae satisfiable, the greater number is transmitted to the next level of agregation for the next comparison. Then, at the top of the construction, we can read in the binary representation of the maximal number of satisfiable formulae.

If we also want the truth assignement that satisfies the greater number of formulae, we can easily devise a new module on the basis of the one used for ENUM-SAT.