

# SARRIGUREN: a polynomial-time complete algorithm for random $k$ -SAT with relatively dense clauses

Alfredo Goñi Sarriguren

Department of Computer Languages and Systems

Faculty of Informatics

University of the Basque Country UPV/EHU

Paseo Manuel de Lardizabal 1 (20018) Donostia, Spain

alfredo@ehu.es

April 8, 2025

## Abstract

SARRIGUREN, a new complete algorithm for SAT based on counting clauses (which is valid also for Unique-SAT and #SAT) is described, analyzed and tested. Although existing complete algorithms for SAT perform slower with clauses with many literals, that is an advantage for SARRIGUREN, because the more literals are in the clauses the bigger is the probability of overlapping among clauses, a property that makes the clause counting process more efficient. Actually, it provides a  $O(m^2 \times n/k)$  time complexity for random  $k$ -SAT instances of  $n$  variables and  $m$  relatively dense clauses, where that density level is relative to the number of variables  $n$ , that is, clauses are relatively dense when  $k \geq 7\sqrt{n}$ . Although theoretically there could be worst-cases with exponential complexity, the probability of those cases to happen in random  $k$ -SAT with relatively dense clauses is practically zero. The algorithm has been empirically tested and that polynomial time complexity maintains also for  $k$ -SAT instances with less dense clauses ( $k \geq 5\sqrt{n}$ ). That density could, for example, be of only 0.049 working with  $n = 20000$  variables and  $k = 989$  literals. In addition, they are presented two more complementary algorithms that provide the solutions to  $k$ -SAT instances and valuable information about number of solutions for each literal. Although this algorithm does not solve the NP=P problem (it is not a polynomial algorithm for 3-SAT), it broads the knowledge about that subject, because  $k$ -SAT with  $k > 3$  and dense clauses is not harder than 3-SAT. Moreover, the Python implementation of the algorithms, and all the input datasets and obtained results in the experiments are made available.

**Keywords**— polynomial time, SAT complete algorithm, K-SAT, dense clauses

# 1 Introduction

Boolean satisfiability problem (SAT) stands as an iconic challenge that has been extensively researched [1], since it was found to be the first NP-complete algorithm [3]. A reduced version of SAT is  $k$ -SAT where all clauses are represented in CNF (Conjunctive Normal Form) and contain exactly  $k$  literals. Although algorithms to run in polynomial-time have been defined for 2-SAT [11, 7],  $k$ -SAT remains to be NP-complete for  $k \geq 3$ . The complexity of current complete algorithms that solve  $k$ -SAT is exponential: in [4] it is described a deterministic local search algorithm for  $k$ -SAT that runs in  $O((2 - \frac{2}{k+1})^n)$  time, what means that it runs in  $O(1.5^n)$ ,  $O(1.6^n)$ , and  $O(1.6^n)$  time for values of  $k$  equal to 3, 4, and 5 respectively. In [2] a faster but still exponential  $O(1.473^n)$  algorithm for 3-SAT is presented. Moreover, in [9] they present an algorithm  $O(1.307^n)$  for Unique 3-SAT (a variation of 3-SAT that decides if there is only one satisfying assignment or not). It seems that the complexity of  $k$ -SAT grows for higher values of  $k$ , what sounds reasonable because known complete algorithms (Davis-Putnam' [6], Stalmarck's algorithm [15], DPLL [5] and others based on them) make use of existential quantifications, inference rules such as resolution and search that perform slower with higher values of  $k$ , that is, with more literals in the clauses. In fact, in [10] authors claim that the complexity of  $k$ -SAT increases with increasing  $k$ , but this is only under the assumption that  $k$ -SAT does not have subexponential algorithms for  $k \geq 3$ ; an assumption that has to be revisited, according to this work.

In this paper a new complete algorithm for SAT named SARRIGUREN<sup>1</sup> is presented. That algorithm does not make use of existential quantifications, inference rules nor search. It is based on counting unsatisfiable variations for clauses, and the fact of having many literals in the clauses (big values of  $k$ ) results in an advantage that will be shown in this paper. That algorithm provides an  $O(m^2 \times \frac{n}{k})$  polynomial-time complexity when applied to random  $k$ -SAT with dense clauses<sup>2</sup>, that is, to a set of  $m$  clauses of exactly  $k$  randomly-chosen literals of  $n$  different variables where  $k$  is relatively close to  $n$  ( $k \geq 7\sqrt{n}$  or even  $k \geq 5\sqrt{n}$ ). Moreover, this complete algorithm based on counting unsatisfiable variations of clauses is also a complete algorithm for variations of SAT explained in [1] such as the previously mentioned Unique-SAT or propositional model counting #SAT (a variation of SAT that calculates the total number of satisfying assignments).

There are also many incomplete methods that are very efficient in finding solutions to many instances of SAT, but that cannot guarantee unsatisfiability even if they do not find a solution. Among them we can find Survey Propagation that can solve random 3-SAT instances with one million variables and beyond in near-linear time [12]. Up to my knowledge no performance results have been reported for  $k$ -SAT instances with higher values of  $k$ .

In the following sections the algorithm is explained with an example, some definitions and concepts related with the algorithm are given and proven, the algorithm is presented, its corresponding analysis of complexity, experimental results of the algorithm, another algorithm to get the solutions or satisfying assignments is also explained, and finally, the conclusions are presented.

---

<sup>1</sup>SARRIGUREN is a Basque name of a town in Nafarroa/Navarra whose etymological meaning is “beautiful thicket”. Moreover, it is also my mother’s family name.

<sup>2</sup>As it will be discussed in section 5, the probability of exponential time complexity for some worst-cases in random  $k$ -SAT with dense clauses is practically zero.

## 2 Explanation of the algorithm with an example

Let us consider two sets of clauses:  $K_1 = \{k_1, k_2, k_3, k_4\}$  that is satisfiable and  $K_2 = \{k_1, k_2, k_3, k_4, k_5, k_6\}$  that is unsatisfiable, where the clauses are:

$$\begin{aligned} k_1 &= \bar{x}_1 \vee x_2 \\ k_2 &= x_2 \vee x_3 \\ k_3 &= \bar{x}_2 \vee \bar{x}_3 \\ k_4 &= x_1 \vee \bar{x}_2 \vee x_3 \\ k_5 &= x_1 \vee x_2 \\ k_6 &= \bar{x}_1 \vee \bar{x}_2 \end{aligned}$$

It is known that  $K_1$  is satisfiable if there is at least a variation<sup>3</sup> of Boolean values for the variables  $(x_1, x_2, x_3)$  such as  $K_1 = k_1 \wedge k_2 \wedge k_3 \wedge k_4 = 1$ . In Table 1 it can be seen that there are two variations (the satisfying assignments or solutions) that make the set of clauses  $K_1$  satisfiable:  $<0, 0, 1>$  and  $<1, 1, 0>$ . When the clauses  $k_5$  and  $k_6$  are added, then  $K_2$  is unsatisfiable because the combinations  $<0, 0, 1>$  and  $<1, 1, 0>$  make  $k_5$  and  $k_6$  equal to 0, respectively.

$x_1$	$x_2$	$x_3$	$k_1$	$k_2$	$k_3$	$k_4$	$K_1$	$\dots$	$k_5$	$k_6$	$K_2$
0	0	0	1	0	1	1	0	$\dots$	0	1	0
0	0	1	1	1	1	1	1	$\dots$	0	1	0
0	1	0	1	1	1	0	0	$\dots$	1	1	0
0	1	1	1	1	0	1	0	$\dots$	1	1	0
1	0	0	0	0	1	1	0	$\dots$	1	1	0
1	0	1	0	1	1	1	0	$\dots$	1	1	0
1	1	0	1	1	1	1	1	$\dots$	1	0	0
1	1	1	1	1	0	1	0	$\dots$	1	0	0

Table 1: Truth table for  $K_1$  and  $K_2$

It is not an efficient algorithm for solving SAT to build all the variations and check if each variation makes false at least one of the clauses because it has a complexity of  $O(2^n)$ , where  $n$  is the number of variables. However, to *count* the number of variations that make unsatisfiable at least one clause is much more efficient for many instances of SAT. If  $u$  is the number of unsatisfying variations and is equal to  $2^n$ , then the set of clauses is unsatisfiable. And, in other case, it is satisfiable and there are exactly  $(2^n - u)$  solutions. Following with the previous example, notice that  $K_2$  is unsatisfiable because the  $2^3 = 8$  variations do not satisfy at least one clause, and  $K_1$  is satisfiable because only 6 unsatisfiable variations have been found, what also means that there are  $(2^3 - 6) = 2$  solutions that satisfy all clauses.

Let us start by analyzing how can be counted the unsatisfying variations for each clause:

- $k_1 = \bar{x}_1 \vee x_2 \Rightarrow$  the set  $k_1^{full} = \{\bar{x}_1 \vee x_2 \vee \bar{x}_3, \bar{x}_1 \vee x_2 \vee x_3\}$  is equivalent to  $k_1 \Rightarrow \{<1, 0, 1>, <1, 0, 0>\}$  or  $\{x_1 \wedge \bar{x}_2 \wedge x_3, x_1 \wedge \bar{x}_2 \wedge \bar{x}_3\}$  is the set of unsatisfying variations or complementary conjunctive clauses that unsatisfy the clauses in

---

<sup>3</sup>This term variation refers a variation with repetition of 2 Boolean values taken  $n$  variables at a time.

$k_1^{full}$ . That set of variations follows the pattern  $P_1 = < 1, 0, - >$ . If  $v$  is the number of ‘-’ in the pattern or the number of variables that do not appear in the clause  $k_1$ , then there are exactly  $2^v = 2^1 = 2$  unsatisfying variations; and exactly the same number of clauses in  $k_1^{full}$ . Moreover, the complementaries of  $(k_1^{full})^C = \{x_1 \vee x_2 \vee x_3, x_1 \vee x_2 \vee \bar{x}_3, x_1 \vee \bar{x}_2 \vee x_3, x_1 \vee \bar{x}_2 \vee \bar{x}_3, \bar{x}_1 \vee x_2 \vee x_3, \bar{x}_1 \vee x_2 \vee \bar{x}_3\}$ , the set  $\{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3, \bar{x}_1 \wedge \bar{x}_2 \wedge x_3, \bar{x}_1 \wedge x_2 \wedge \bar{x}_3, \bar{x}_1 \wedge x_2 \wedge x_3, x_1 \wedge x_2 \wedge \bar{x}_3, x_1 \wedge x_2 \wedge x_3\}$  are the 8-2=6 solutions for  $K = \{k_1\}$  set.

- $k_2 = x_2 \vee x_3 \Rightarrow k_2^{full} = \{\bar{x}_1 \vee x_2 \vee x_3, x_1 \vee x_2 \vee x_3\} \Rightarrow$  set  $\{< 1, 0, 0 >, < 0, 0, 0 >\}$  of unsatisfying variations  $\Rightarrow$  unsatisfiability pattern  $P_2 = < -, 0, 0 > \Rightarrow 2$  unsatisfying variations (also 2 clauses in  $k_2^{full}$ ).

To calculate the cardinality of the union of two patterns (sets of variations)  $P_1$  and  $P_2$ , then the intersection of both sets is required:  $|P_1 \cup P_2| = |P_1| + |P_2| - |P_1 \cap P_2|$ . The intersection of two patterns is obtained by merging them. If there are complementary values for the same position, then the intersection is empty.  $\Rightarrow P_{12} = P_1 \cap P_2 = \text{intersect}(< 1, 0, - >, < -, 0, 0 >) = < 1, 0, 0 >$ , whose cardinality is  $2^0 = 1$ . It is easy to check that  $< 1, 0, 0 >$  is the only element appearing in both sets of unsatisfying variations of  $k_1$  and  $k_2$  (and also that only the clause  $\bar{x}_1 \vee x_2 \vee x_3$  appears in both sets  $k_1^{full}$  and  $k_2^{full}$ )

By the moment the number of unsatisfying variations for  $k_1$  and  $k_2$  is  $2+2-1=3$ , and the number of variations that satisfy both clauses is  $2^3 - 3 = 5$ , as can be checked in Table 2

$x_1$	$x_2$	$x_3$	$k_1$	$k_2$	$K = \{k_1, k_2\}$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	1	1

Table 2: Truth table for clauses  $k_1$  and  $k_2$

- $k_3 = \bar{x}_2 \vee \bar{x}_3 \Rightarrow k_3^{full} = \{\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, x_1 \vee \bar{x}_2 \vee \bar{x}_3\} \Rightarrow$  unsatisfiability pattern  $P_3 = < -, 1, 1 > \Rightarrow 2$  variations/clauses  
 $\Rightarrow P_{13} = P_1 \cap P_3 = \text{intersect}(< 1, 0, - >, < -, 1, 1 >) = \emptyset$ , because the values  $\underline{0}$  and  $\underline{1}$  appear in the position corresponding to the variable  $x_2$ .  
 $\Rightarrow P_{23} = P_2 \cap P_3 = \text{intersect}(< -, 0, 0 >, < -, 1, 1 >) = \emptyset$   
At this point  $P_1 \cap P_2 \cap P_3$  is also needed because  $|P_1 \cap P_2 \cap P_3|$  has to be added now (by following the formula presented in section 3.6). Fortunately  $P_1 \cap P_2 \cap P_3$  can be obtained by using the results of previously calculated intersections. In this case,  $P_{123} = P_1 \cap P_2 \cap P_3 = (P_1 \cap P_2) \cap P_3 = P_{12} \cap P_3$   
 $\Rightarrow P_{123} = P_{12} \cap P_3 = \text{intersect}(< 1, 0, 0 >, < -, 1, 1 >) = \emptyset$   
Unsatisfying variations now are  $3 + 2 - 0 - 0 + 0 = 5$
- $k_4 = x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow k_4^{full} = \{x_1 \vee \bar{x}_2 \vee x_3\} \Rightarrow$  pattern  $P_4 = < 0, 1, 0 > \Rightarrow 1$  variation/clause

$$\Rightarrow P_{14} = P_1 \cap P_4 = \text{intersect}(<1, 0, ->, <0, 1, 0>) = \emptyset$$

$$\Rightarrow P_{24} = P_2 \cap P_4 = \text{intersect}(<-, 0, 0>, <0, 1, 0>) = \emptyset$$

$$\Rightarrow P_{34} = P_3 \cap P_4 = \text{intersect}(<-, 1, 1>, <0, 1, 0>) = \emptyset$$

The only other intersection that needs to be calculated is  $P_{12} \cap P_4$  because all the other ones are empty, so the intersection with  $P_4$  is also empty.

$$P_{124} = P_{12} \cap P_4 = \text{intersect}(<1, 0, 0>, <0, 1, 0>) = \emptyset$$

Unsatisfying variations now are  $5 + 1 - 0 - 0 - 0 + 0 = 6$ , and as there are no more clauses in  $K_1$  then  $K_1$  is satisfiable and has 2 solutions. Those solutions are the complementaries of  $(K_1^{full})^C = \{x_1 \vee x_2 \vee \bar{x}_3, \bar{x}_1 \vee \bar{x}_2 \vee x_3\}$ . Although this algorithm does not provide any of those solutions, the algorithm presented in section 7 will be able to do it.

Let us continue with the rest of clauses of the unsatisfiable set  $K_2$

- $k_5 = x_1 \vee x_2 \Rightarrow k_5^{full} = \{x_1 \vee x_2 \vee \bar{x}_3, x_1 \vee x_2 \vee x_3\} \Rightarrow$  pattern  $P_5 = <0, 0, ->$   
 $\Rightarrow 2$  variations/clauses

$$P_{15} = P_1 \cap P_5 = \text{intersect}(<1, 0, ->, <0, 0, ->) = \emptyset$$

$\Rightarrow P_{25} = P_2 \cap P_5 = \text{intersect}(<-, 0, 0>, <0, 0, ->) = <0, 0, 0>$ , 1 variation to subtract

$$\Rightarrow P_{35} = P_3 \cap P_5 = \text{intersect}(<-, 1, 1>, <0, 0, ->) = \emptyset$$

$$\Rightarrow P_{45} = P_4 \cap P_5 = \text{intersect}(<0, 1, 0>, <0, 0, ->) = \emptyset$$

$$\Rightarrow P_{125} = P_{12} \cap P_5 = \text{intersect}(<1, 0, 0>, <0, 0, ->) = \emptyset$$

Unsatisfying variations now are  $6 + 2 - 0 - 1 - 0 - 0 + 0 = 7$

- $k_6 = \bar{x}_1 \vee \bar{x}_2 \Rightarrow k_6^{full} = \{\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, \bar{x}_1 \vee \bar{x}_2 \vee x_3\} \Rightarrow$  pattern  $P_6 = <1, 1, ->$   
 $\Rightarrow 2$  variations/clauses

$$P_{16} = P_1 \cap P_6 = \text{intersect}(<1, 0, ->, <1, 1, ->) = \emptyset$$

$$\Rightarrow P_{26} = P_2 \cap P_6 = \text{intersect}(<-, 0, 0>, <1, 1, ->) = \emptyset$$

$\Rightarrow P_{36} = P_3 \cap P_6 = \text{intersect}(<-, 1, 1>, <1, 1, ->) = <1, 1, 1>$ , 1 variation to subtract

$$\Rightarrow P_{46} = P_4 \cap P_6 = \text{intersect}(<0, 1, 0>, <1, 1, ->) = \emptyset$$

$$\Rightarrow P_{56} = P_5 \cap P_6 = \text{intersect}(<0, 1, 0>, <1, 1, ->) = \emptyset$$

$$\Rightarrow P_{126} = P_{12} \cap P_6 = \text{intersect}(<1, 0, 0>, <1, 1, ->) = \emptyset$$

$$\Rightarrow P_{256} = P_{25} \cap P_6 = \text{intersect}(<0, 0, 0>, <1, 1, ->) = \emptyset$$

Unsatisfying variations now are  $7 + 2 - 0 - 0 - 1 - 0 - 0 + 0 + 0 = 8$ . Therefore, with this clause the set of clauses is unsatisfiable

Although there are potentially many variations of patterns that require to be calculated, just when an intersection among some variations is empty, then adding more patterns to intersect with that is also empty, and they do not need to be calculated. In this case, from all the possible intersections among the patterns  $P_1, P_2, P_3, P_4, P_5$  and  $P_6$ , only 3 were not empty:  $P_{12}, P_{25}$  and  $P_{36}$ , that is 3 of 57 ( $\binom{6}{2} + \binom{6}{3} + \binom{6}{4} + \binom{6}{5} + \binom{6}{6} = 15 + 20 + 15 + 6 + 1 = 57$ ). The reason for this is that the clauses are relatively dense: the proportion among the number of literals and the number of variables is relatively high. In this case that proportion is 1 for clause  $k_4$  (3 literals, 3 variables) and 0.66 for the rest of clauses (2 literals, 3 variables). Moreover, when the intersection among patterns is not empty, the result pattern is even more dense:  $P_{12}, P_{25}$  and  $P_{36}$  have a density of 1.

This is exactly the advantage mentioned in section 1: having many literals in the clauses (big values of  $k$ ) increments the possibilities of finding complementary literals among the clauses what results in empty intersections. In this paper, we will see that this algorithm can be executed in polynomial time for all instances of  $k$ -SAT with relatively *dense* clauses; clauses where  $k$  is relatively close to  $n$  ( $k \geq 7\sqrt{n}$  or even  $k \geq 5\sqrt{n}$ ).

### 3 Previous concepts employed by the algorithm

Once the algorithm has been explained with a motivating example, the definitions and proofs of some concepts are going to be presented in this section.

#### 3.1 The sets $K^*$ , $K^{full}$ , $\bar{K}^{full}$ and $\bar{K}$

Let  $K = \{k_1 \dots k_m\}$  be a CNF formula of  $m$  disjunctive clauses with literals of  $n$  variables  $x_1 \dots x_n$ . The next sets can be defined:

- $K^*$  is the set of all possible disjunctive clauses that can be built with the *different* variations of literals of the  $n$  variables.

$$K^* = \{(x_1, \dots, x_n), (x_1, \dots, \bar{x}_n), \dots, (\bar{x}_1, \dots, \bar{x}_n)\}$$

- Each clause in  $K^*$  is a *full* clause that contains exactly one literal for each variable.
- $K^*$  contains  $2^n$  different full clauses:  $|K^*| = 2^n$

- $K^{full}$  contains all the full disjunctive clauses corresponding to the clauses  $k_i$  of  $K$ :

$$K^{full} = \bigcup_{i=1}^m (k_i^{full})$$

where  $k_i^{full}$  is the set of all *full* clauses obtained by adding to the clause  $k_i$  with  $j$  literals every variation of the  $x_{j+1}^i, \dots, x_n^i$  variables without literal in  $k_i$

$$k_i^{full} = \{k_i \vee x_{j+1}^i \vee \dots \vee x_n^i, k_i \vee x_{j+1}^i \vee \dots \vee \bar{x}_n^i, \dots, k_i \vee \bar{x}_{j+1}^i \vee \dots \vee \bar{x}_n^i\}$$

- $\bar{K}^{full}$  contains all the full conjunctive clauses that are the complementaries of the clauses in  $K^{full}$ :

$$\bar{K}^{full} = \{c | c = \bar{d} \wedge d \in K^{full}\}$$

For each clause in  $\bar{K}^{full}$  there is a clause in  $K^{full}$ , then  $|\bar{K}^{full}| = |K^{full}|$

- $\bar{K}$  contains all the conjunctive clauses that are the complementaries of the clauses  $k_i$  of  $K$ :

$$\bar{K} = \{c | c = \bar{d} \wedge d \in K\}$$

#### 3.2 $K^*$ is unsatisfiable

Let  $x_1 \dots x_n$  be a set of variables, then the conjunction of all the possible (disjunctive) clauses that can be formed with their literals is False, and therefore, unsatisfiable:  
 $(x_1 \vee \dots \vee x_n) \wedge (x_1 \vee \dots \vee \bar{x}_n) \wedge \dots \wedge (\bar{x}_1 \vee \dots \vee \bar{x}_n) = False$

##### Proof by induction

- It is true for  $n = 1$  because  $(x_1 \wedge \bar{x}_1) = False$

- We suppose it is true for  $n = k$   

$$(x_1 \vee \cdots \vee x_k) \wedge (x_1 \vee \cdots \vee \overline{x_k}) \wedge \cdots \wedge (\overline{x_1} \vee \cdots \vee \overline{x_k}) = False$$
- Is it also true for  $n = k + 1$ ?  

$$\begin{aligned} & (x_1 \vee \cdots \vee x_k \vee x_{k+1}) \wedge (x_1 \vee \cdots \vee x_k \vee \overline{x_{k+1}}) \wedge \cdots \wedge (\overline{x_1} \vee \cdots \vee \overline{x_k} \vee x_{k+1}) \wedge (\overline{x_1} \vee \\ & \cdots \vee \overline{x_k} \vee \overline{x_{k+1}}) = \\ & ((x_1 \vee \cdots \vee x_k) \wedge (x_1 \vee \cdots \vee \overline{x_k}) \wedge \cdots \wedge (\overline{x_1} \vee \cdots \vee \overline{x_k})) \vee x_{k+1}) \wedge \\ & ((x_1 \vee \cdots \vee x_k) \wedge (x_1 \vee \cdots \vee \overline{x_k}) \wedge \cdots \wedge (\overline{x_1} \vee \cdots \vee \overline{x_k})) \vee \overline{x_{k+1}}) = \\ & (False \vee x_{k+1}) \wedge (False \vee \overline{x_{k+1}}) = (x_{k+1}) \wedge (\overline{x_{k+1}}) = False \end{aligned}$$
- Yes, it is true for  $n = k + 1$ , and therefore the proposition is proven.

### 3.3 $K$ is equivalent to its expanded set of full clauses $K^{full}$

Now the following two properties can be proven:

- The set  $\{k\}$  with a disjunctive clause  $k$  is equivalent to  $k^{full}$   
As any clause  $k$  is equivalent to  $(k \vee False)$ , and the conjunction of all possible variation of clauses formed with a set of variables is  $False$  (see section 3.2)  

$$\begin{aligned} k &= (k \vee False) = \\ &= (k \vee ((x_{i+1} \vee \cdots \vee x_n) \wedge (x_{i+1} \vee \cdots \vee \overline{x_n}) \wedge \cdots \wedge (\overline{x_{i+1}} \vee \cdots \vee \overline{x_n})) = \\ &= ((k \vee x_{i+1} \vee \cdots \vee x_n) \wedge (k \vee x_{i+1} \vee \cdots \vee \overline{x_n}) \wedge \cdots \wedge (k \vee \overline{x_{i+1}} \vee \cdots \vee \overline{x_n})) \\ &\Rightarrow \{k\} \equiv k^{full} \end{aligned}$$
- $K$  is equivalent to  $K^{full}$   

$$K = \{k_1, \dots, k_m\} = \bigcup_{i=1}^m \{\{k_i\}\} = \bigcup_{i=1}^m (k_i^{full}) = K^{full}$$

### 3.4 $K$ is unsatisfiable when its $K^{full}$ contains $2^n$ clauses

$K^* = \{(x_1, \dots, x_n), (x_1, \dots, \overline{x_n}), \dots, (\overline{x_1}, \dots, \overline{x_n})\}$  is the set of all possible and *different* variations of clauses that can be formed with  $n$  variables where  $|K^*| = 2^n$  (according to 3.1). If the  $K^{full}$  set derived from a set  $K$  also contains  $2^n$  different clauses,  $|K^{full}| = 2^n$ , then they must be all the clauses in  $K^*$ :

$$\begin{aligned} &\Rightarrow K^{full} = K^* \\ &\Rightarrow K^{full} \text{ is unsatisfiable (according to 3.2)} \\ &\Rightarrow K \text{ is unsatisfiable (according to 3.3)} \end{aligned}$$

### 3.5 Complementaries of clauses in $(K^{full})^C$ satisfy $K$

Once known that  $K$  is unsatisfiable when  $K^{full} = K^*$ , we analyze now what happens when  $K^{full} \subset K^*$ . Let  $(K^{full})^C$  be the set of disjunctive full clauses that are in  $K^*$  but not in  $K^{full}$ , that is  $(K^{full})^C = K^* - K^{full}$ . Notice that  $K^{full} \cup (K^{full})^C = K^*$ , what means that  $K^{full}$  and  $(K^{full})^C$  are complementary sets and  $c \in (K^{full})^C \Leftrightarrow c \notin K^{full}$ . In this case, the complementaries of clauses of  $(K^{full})^C$  are solutions that satisfy  $K$ :

$$\begin{aligned} c &= (c_1, \dots, c_n) = (c_1 \vee \cdots \vee c_n) \in (K^{full})^C \rightarrow \\ &\rightarrow \bar{c} = (\overline{c_1}, \dots, \overline{c_n}) = (\overline{c_1} \wedge \cdots \wedge \overline{c_n}) \text{ is a solution of } K \end{aligned}$$

#### Proof by induction

- For  $n = 1$ , these are all the possibilities with elements of  $K_1^* = \{x_1, \bar{x}_1\}$ :

1.  $K = \{x_1\} \Rightarrow K^{full} = \{x_1\} \Rightarrow \bar{x}_1 \in (K^{full})^C \Rightarrow \bar{\bar{x}}_1 = x_1$  is a solution for  $K = \{x_1\}$
2.  $K = \{\bar{x}_1\} \Rightarrow K^{full} = \{\bar{x}_1\} \Rightarrow x_1 \in (K^{full})^C \Rightarrow \bar{x}_1$  is a solution for  $K = \{\bar{x}_1\}$
3.  $K = \{x_1, \bar{x}_1\} \Rightarrow K^{full} = \{x_1, \bar{x}_1\} \Rightarrow (K^{full})^C = \emptyset \Rightarrow \nexists c \in (K^{full})^C$

Therefore, in all cases for  $n = 1$   $\{c \in (K^{full})^C \rightarrow \bar{c}$  is a solution of  $K\}$  is true

- We suppose that it is true for  $n = q$   
 $c = (c_1, \dots, c_q) \in (K^{full})^C \rightarrow \bar{c} = (\bar{c}_1, \dots, \bar{c}_q)$  is a solution of  $K$  (Result 1)
- Is it also true for  $n = q + 1$ ?

#### Proof by reductio ad absurdum

Let us suppose that it is false:

$$\begin{aligned} ((c_1, \dots, c_{q+1}) \in (K^{full})^C \rightarrow (\bar{c}_1, \dots, \bar{c}_{q+1})) \text{ is a solution of } K &= False \\ \Rightarrow (c_1, \dots, c_{q+1}) \in (K^{full})^C \wedge (\bar{c}_1, \dots, \bar{c}_{q+1}) &\text{ is not a solution of } K \\ \Rightarrow (c_1, \dots, c_q, c_{q+1}) \in (K^{full})^C \wedge (\bar{c}_1, \dots, \bar{c}_q, \bar{c}_{q+1}) &\text{ is not a solution of } K \\ \Rightarrow (c_1 \vee \dots \vee c_q \vee c_{q+1}) \in (K^{full})^C &\text{ (Result 2)} \\ \wedge (\bar{c}_1, \dots, \bar{c}_q, \bar{c}_{q+1}) &\text{ is not a solution of } K \text{ (Result 3)} \end{aligned}$$

Let us evaluate the clauses of  $K^{full}$  with  $(\bar{c}_1, \dots, \bar{c}_q, \bar{c}_{q+1})$  then:

- The literal  $\bar{c}_{q+1}$  will satisfy all the clauses in  $K^{full}$  containing literal  $\bar{c}_{q+1}$ , because they are conjunctive clauses ( $k' \vee \bar{c}_{q+1} = k' \vee True = True$ ). Let  $K_{rest}^{full}$  be the rest of clauses to be satisfied.  
Notice that if the clause  $(c_1 \vee \dots \vee c_q \vee \bar{c}_{q+1})$  were in  $K^{full}$ , then  $(c_1 \vee \dots \vee c_q)$  would not be in  $K_{rest}^{full}$ .
- The literal  $\bar{c}_{q+1}$  will eliminate the literal  $c_{q+1}$  in the clauses of  $K_{rest}^{full}$  that contain it ( $k' \vee c_{q+1} = k' \vee False = k'$ ). As the previous result 2 says that  $(c_1 \vee \dots \vee c_q \vee c_{q+1}) \in (K^{full})^C$  then  $(c_1 \vee \dots \vee c_q \vee c_{q+1}) \notin K^{full}$ , and therefore  $(c_1 \vee \dots \vee c_q)$  will not be either in  $K_{rest}^{full}$ .
- At this point it is known that clause  $(c_1 \vee \dots \vee c_q) \notin K_{rest}^{full}$  and that  $K_{rest}^{full}$  is composed of  $q$  variables and not  $q + 1$  variables (because literals  $c_{q+1}$  and  $\bar{c}_{q+1}$  do not appear in clauses of  $K_{rest}^{full}$ ). As the proposition for  $n = q$  is true (Result 1), then  $(\bar{c}_1, \dots, \bar{c}_q)$  is a solution for  $K_{rest}^{full}$
- If the literal  $\bar{c}_{q+1}$  has satisfied part of  $K^{full}$  reducing it to  $K_{rest}^{full}$ , and the literals  $(\bar{c}_1, \dots, \bar{c}_q)$  have satisfied  $K_{rest}^{full}$ , then  $(\bar{c}_1, \dots, \bar{c}_q, \bar{c}_{q+1})$  is a solution for  $K$ , what it is a contradiction that demonstrates that the proposition for  $n = q + 1$  is also true.

Therefore the solutions that satisfy  $K$  are the complementaries of clauses in  $(K^{full})^C$ , or what it is the same, the conjunctive clauses that are in  $(\bar{K}^{full})^C$ .

### 3.6 Counting clauses in $K^{full}/\bar{K}^{full}$ directly from $K/\bar{K}$

To build the  $K^{full}$  set derived from  $K$  and find if there are solutions (complementaries of full clauses that are in  $K^*$  but not in  $K^{full}$ ) that make  $K$  satisfiable leads to an  $O(2^n)$  algorithm. It is obvious that if  $K$  is unsatisfiable then  $K^{full}$  would be  $K^*$ , that contains  $2^n$  full clauses. It is much more efficient to count the number of clauses that

would be in  $K^{full}$  directly from  $K$ , which have been shown to be equivalent to  $K^{full}$  in section 3.3. In that case, the number of solutions would be  $(2^n - |K^{full}|)$

In order to calculate the size of those sets of clauses, then common formula of combinatorics such as calculating the number of variations and the inclusion-exclusion principle [14] can be used.

- The number of full clauses corresponding to a clause  $k$  with  $j$  literals corresponding to  $n$  different variables is  $2^{n-j}$ , that is the number of variations that can be obtained with two literals  $x_p$  or  $\bar{x}_p$  for the  $(n-j)$  remaining variables, where  $p \in \{j+1, n\}$ . Therefore,  $|\{k\}| = 2^{n-j}$
- The number of full clauses in  $K = K^{full} = \bigcup_{i=1}^m (k_i^{full}) = \bigcup_{i=1}^m (\{k_i\})$  is determined by the principle of inclusion-exclusion:

$$|K| = |\bigcup_{i=1}^m (\{k_i\})| = \sum_{i=1}^m |\{k_i\}| - \sum_{j,k:1 \leq j < k \leq m} |\{k_j\} \cap \{k_k\}| + \\ + \sum_{j,k,l:1 \leq j < k < l \leq m} |\{k_j\} \cap \{k_k\} \cap \{k_l\}| + \cdots + (-1)^{m+1} |\{k_1\} \cap \cdots \cap \{k_m\}|$$

Due to the way in which  $\bar{K}^{full}$  set has been built (see section 3.1), then  $|K^{full}| = |\bar{K}^{full}|$  and, with a similar reasoning, the following can be concluded:

- $|\{\bar{k}\}| = 2^{n-j}$
- $|\bar{K}| = |\bigcup_{i=1}^m (\{\bar{k}_i\})| = \sum_{i=1}^m |\{\bar{k}_i\}| - \sum_{j,k:1 \leq j < k \leq m} |\{\bar{k}_j\} \cap \{\bar{k}_k\}| + \\ + \sum_{j,k,l:1 \leq j < k < l \leq m} |\{\bar{k}_j\} \cap \{\bar{k}_k\} \cap \{\bar{k}_l\}| + \cdots + (-1)^{m+1} |\{\bar{k}_1\} \cap \cdots \cap \{\bar{k}_m\}|$

### 3.7 Intersections of conjunctive clauses are efficient

The intersection of clauses  $c$  and  $d$  is the set of clauses built with the variations of literals of variables that satisfy  $c$  and that also satisfy  $d$ :  $\{c\} \cap \{d\} = \{c \wedge d\}$ . When the clauses are conjunctive ( $\bar{k}_i$  and  $\bar{k}_j$  are conjunctive) then the intersection is also a conjunctive clause that is much more efficient to calculate<sup>4</sup>

$$\begin{aligned} \bar{k}_i \wedge \bar{k}_j = \\ = (\bar{l}_{i1} \wedge \bar{l}_{i2} \cdots \wedge \bar{l}_{ip}) \wedge (\bar{l}_{j1} \wedge \bar{l}_{j2} \cdots \wedge \bar{l}_{jq}) = (\bar{l}_{i1} \wedge \bar{l}_{i2} \cdots \wedge \bar{l}_{ip} \wedge \bar{l}_{j1} \wedge \bar{l}_{j2} \cdots \wedge \bar{l}_{jq}) \end{aligned}$$

The good news here is that if there are complementary literals ( $\exists r, s : l_{jr} = \bar{l}_{is}$ ), then  $\bar{k}_i \wedge \bar{k}_j = \emptyset$ . Moreover, any other intersection with other clauses including  $\bar{k}_i$  and  $\bar{k}_j$  is also empty because  $k \wedge \emptyset = \emptyset$  for any  $k$ . As it is much more efficient to calculate intersections of conjunctive clauses, then it will be much better to calculate  $|\bar{K}|$  instead of  $|K|$

## 4 SARRIGUREN: a complete algorithm for SAT

The algorithm returns  $(2^n - |\bar{K}|)$ , the number of solutions or satisfying assignments of a set  $K$  of  $m$  disjunctive clauses  $\{k_1, \dots, k_m\}$  with  $n$  variables.  $K$  is unsatisfiable if there are zero solutions, and satisfiable if there at least one. The value  $|\bar{K}|$  is calculated by applying the formula  $\sum_{i=1}^m |\{\bar{k}_i\}| - \sum_{j,k} |\{\bar{k}_j\} \cap \{\bar{k}_k\}| + \cdots + (-1)^{m+1} |\{\bar{k}_1\} \cap \cdots \cap \{\bar{k}_m\}|$ . The cardinality of all the conjunctive clauses  $|\{\bar{k}_i\}|$  and intersecting clauses  $|\{\bar{k}_1\} \cap \cdots \cap \{\bar{k}_m\}|$  is calculated as  $2^{n-var\#}$ , and the intersection of conjunctive clauses

---

<sup>4</sup>If the clauses are disjunctive, the intersection clause is not disjunctive:  $k_i \wedge k_j = (l_{i1} \wedge l_{j1}) \vee (l_{i1} \wedge l_{j2}) \vee \cdots (l_{ip} \wedge l_{jq})$

is the result of merging the literals of the clauses (the intersection is  $\emptyset$  if clauses contain complementary literals). The cardinalities are calculated in this order:  $+|\{\bar{k}_1\}|$ ,  $+|\{\bar{k}_2\}|$ ,  $-|\{\bar{k}_1\} \cap \{\bar{k}_2\}|$ ,  $+|\{\bar{k}_3\}|$ ,  $-|\{\bar{k}_1\} \cap \{\bar{k}_3\}|$ ,  $-|\{\bar{k}_2\} \cap \{\bar{k}_3\}|$ ,  $+|\{\bar{k}_1\} \cap \{\bar{k}_2\} \cap \{\bar{k}_3\}|$ ,  $\dots$   $+/-|\{\bar{k}_1\} \cap \dots \cap \{\bar{k}_m\}|$ . Whenever an intersection of clauses is empty, then it is not further processed with other clauses. And, after the processing of a clause  $k_j$  (by calculating cardinalities with previous clauses  $k_1 \dots k_{j-1}$ , if  $|\bar{K}| = 2^n$ , then the algorithm ends by returning zero. Finally,  $k_i$  can be used instead of  $\bar{k}_i$ , because the counting of clauses and calculation of intersections gets the same results with both of them.

---

**Algorithm 1** SARRIGUREN algorithm

---

**Input:**  $K = k_1 \dots k_m$ , a set of disjunctive clauses sorted by variable number  
 $n$ : number of variables

**Output:** 0 if  $K$  is UNSAT or the number of solutions if it is SAT

**Precondition:** there are no complementary literals in any clause

$P \leftarrow []$   $\triangleright P$  contains the signed patterns processed until now  
 $u \leftarrow 0$   $\triangleright u$  number of unsatisfiable variations at the moment

```

for  $i$  in  $1..n$  do
     $ap \leftarrow \text{pattern}(k_i)$   $\triangleright ap$  is the pattern of the actual clause  $k_i$ 
     $N \leftarrow []$   $\triangleright N$ : signed patterns to process with clauses  $k_{i+1} \dots k_n$ 
     $u \leftarrow u + \text{cardinality}(ap,n)$   $\triangleright$  add to  $u$  the #unsat variations of  $k_i$ 
    add  $<^-, ap>$  to  $N$   $\triangleright$  adds the negative pattern of  $k_i$  in order to
         $\triangleright$  subtract repeated variations in  $k_{i+1} \dots k_n$ 

    for  $<sign, p>$  in  $P$  do  $\triangleright$  signed patterns processed in  $k_1 \dots k_{i-1}$ 
         $\triangleright p$  is the pattern to process with  $ap$ , the  $sign$  is '+' or '-'
         $ip \leftarrow \text{intersect}(ap,p)$   $\triangleright ip$  is the intersection pattern
        if  $ip$  is not empty then
             $u \leftarrow u sign \text{ cardinality}(ip,n)$   $\triangleright$  #unsat variations of the
                 $\triangleright$  intersection are added or subtracted
            add  $<\text{contrary}(sign), ip>$  to  $N$   $\triangleright$  adds intersection pattern
                 $\triangleright$  with the contrary sign to add/subtract repeated in  $k_{i+1} \dots k_n$ 
        end if
    end for
    append  $N$  to  $P$   $\triangleright$  the new signed patterns calculated by processing  $k_i$ 
         $\triangleright$  are added in order to be processed with  $k_{i+1} \dots k_n$ 
    if  $u = 2^n$  then
        return 0  $\triangleright$  UNSAT: all the variations are unsatisfiable
    end if
end for
return  $((2^n - u))$   $\triangleright$  SAT: there are  $(2^n - u)$  solutions

```

---

**pattern** ( $c$ ) returns  $c$  {And not  $\bar{c}$  because **cardinality** and **intersect** work equal}  
**cardinality** ( $c,n$ ) returns  $2^{n-nv}$ , where  $nv$  is the number of literals in  $c$   
**contrary** ( $s$ ) returns + if  $s$  is -, and - if  $s$  is +  
**intersect** ( $c,d$ ) returns sorted merge of  $c$  and  $d$  or  $\emptyset$  if complementary literals

---

Notice that, on the one hand, this algorithm is designed to know if a set is satisfiable, but not to know which are the solutions that satisfy that set. The solutions are the variations that are in  $(\bar{K}^{full})^C$ . However  $\bar{K}^{full}$  is not being built, only counted. Section 7 presents an algorithm that obtains solutions by using SARRIGUREN. On the other hand, SARRIGUREN is an algorithm for #SAT and for Unique-SAT because it counts the total number of satisfying assignments, and because it can be easily checked if there is only one or not.

## 5 Analysis of complexity of SARRIGUREN

In the algorithm there is a loop (**for**  $i$  in  $1..n$  **do**) that processes each clause  $k_i$  in this way: it adds  $|k_i|$  to a counter, it calculates the intersections between  $k_i$  and all the clauses  $p$  stored in  $P$  (**intersect**( $ap, p$ )), where  $P$  contains all the previous clauses  $k_1 \dots k_{i-1}$ , and all the non-empty or overlapping intersections among clauses  $k_1 \dots k_{i-1}$ . While those intersections are calculated then their sizes  $|k_i \cap k_x \cap \dots \cap k_y|$  are added or deleted from the counter, according to the formula of section 3.6. The new overlapping intersections found in each iteration are added to  $N$  (**add <contrary(sign),ip> to  $N$** ), and after the processing of clause  $k_i$ , added to  $P$  (**append  $N$  to  $P$** ) to be processed with clause  $k_{i+1}$  in the next iteration of the loop.

The key point in order to analyze the complexity of that algorithm is the number of overlapping intersections of clauses  $\overline{k_i}$  and  $\overline{k_j}$ . If that number of overlapping intersections is close to zero, then the complexity of the algorithm decreases substantially because no new intersection clauses are added to  $P$ .

In the following the probability of overlapping intersections clauses is first analyzed, then the number of new clauses that need to be processed. After that, the analysis of the complexity of SARRIGUREN algorithm (applied to  $k$ -SAT with dense clauses) for the average case will be discussed, and also some remarks for the best and worst cases.

### 5.1 Probability of overlapping among dense clauses

Let us find the probability of the intersection of clauses  $c$  and  $d$  to overlap (or not to be disjoint). That happens when there are no complementary literals in clauses  $c$  and  $d$ . If a clause  $c$  has  $k_c$  literals corresponding to  $n$  possible variables, the possibilities of clause  $d$  with  $k_d$  literals and also of  $n$  variables not to have a complementary literal with  $c$  are these ones:

- All the  $k_d$  literals of  $d$  are chosen from the  $(n - k_c)$  variables without literal of  $c$ :  $\binom{n-k_c}{k_d}$  combinations where each combination of  $k_d$  literals can variate with the 2 possibilities for each literal:  $2^{k_d}$ . There are in total  $\binom{n-k_c}{k_d} \times 2^{k_d}$  possibilities (or what it is the same:  $\binom{n-k_c}{k_d-0} \times (2^{k_d-0}) \times \binom{k_c}{0}$ )
- $(k_d - 1)$  literals of  $d$  are chosen from the  $(n - k_c)$  variables without literal of  $c$ :  $\binom{n-k_c}{k_d-1}$  combinations where each combination of  $(k_d - 1)$  literals can variate with the 2 possibilities for each literal:  $2^{k_d-1}$ . The last literal of  $d$  is chosen from the  $k_c$  variables with literal of  $c$ , but is the same literal (and does not overlap). There are in total  $\binom{n-k_c}{k_d-1} \times 2^{k_d-1} \times k_c$  possibilities (or what it is the same:  $\binom{n-k_c}{k_d-1} \times (2^{k_d-1}) \times \binom{k_c}{1}$ )
- $(k_d - 2)$  literals of  $d$  are chosen from the  $(n - k_c)$  variables without literal of  $c$ :  $\binom{n-k_c}{k_d-2}$  combinations where each combination of  $(k_d - 2)$  literals can variate with the 2 possibilities for each literal:  $2^{k_d-2}$ . The other 2 literals of  $d$  are chosen from the  $k_c$  variables with literal of  $c$ , but they are the same literal (and do not overlap). There are in total  $\binom{n-k_c}{k_d-2} \times (2^{k_d-2}) \times \binom{k_c}{2}$  possibilities (or what it is the same:  $\binom{n-k_c}{k_d-2} \times (2^{k_d-2}) \times \binom{k_c}{2}$ )
- The same can be done by choosing  $(k_d - i)$  literals of  $d$  until  $i = (k_d - 1)$ . There are in total  $\binom{n-k_c}{k_d-i} \times (2^{k_d-i}) \times \binom{k_c}{i}$  possibilities for each case.

- No literal of  $d$  is chosen from the  $(n - k_c)$  variables without literal of  $c$ . All the  $k_d$  literals of  $d$  are chosen from the  $k_c$  variables with literal of  $c$ , but they are the same literal (and do not overlap). There are  $\binom{k_c}{k_d}$  possibilities (or what it is the same:  $\binom{n-k_c}{k_d} \times (2^{k_d-k_c}) \times \binom{k_c}{k_d}$ )

In summary there are  $\sum_{i=0}^{k_d} (\binom{n-k_c}{k_d-i} \times (2^{k_d-i}) \times \binom{k_c}{i})$  possible  $d$  clauses that overlap with a given  $c$  clause, from a total of  $\binom{n}{k_d} \times 2^{k_d}$  possible  $d$  clauses. Therefore, the probability of  $c$  and  $d$  clauses to overlap is:

$$P_{overlap} = \frac{\sum_{i=0}^{k_d} (\binom{n-k_c}{k_d-i} \times (2^{k_d-i}) \times \binom{k_c}{i})}{\binom{n}{k_d} \times 2^{k_d}}$$

Let us see what happens in the  $k$ -SAT case with very dense clauses, that is, when  $k$  is very close to  $n$ , and  $k = k_c = k_d$ :

$$\begin{aligned} \lim_{k \rightarrow n} P_{overlap} &= \lim_{k \rightarrow n} \frac{\sum_{i=0}^k (\binom{n-k}{k-i} \times (2^{k-i}) \times \binom{k}{i})}{\binom{n}{k} \times 2^k} = \\ &= \frac{\sum_{i=0}^n (\binom{n-n}{n-i} \times (2^{n-i}) \times \binom{n}{i})}{\binom{n}{n} \times 2^n} = \frac{((n-n) \times (2^{n-n}) \times \binom{n}{n})}{\binom{n}{n} \times 2^n} = \frac{1}{2^n} \end{aligned}$$

The only feasible term in the summatory  $\sum_{i=0}^n$  is the corresponding to  $i = n$  because the others contain bad combinatorial numbers:  $\binom{0}{n} \dots \binom{0}{n-1}$ .

Therefore, when the  $k$ -SAT instance to solve has very *dense* clauses, the probability of overlapping intersections  $P_{overlap}$  is  $\frac{1}{2^n}$ , that is zero for big values of  $n$  ( $\lim_{n \rightarrow \infty} \frac{1}{2^n} = 0$ ), but also very close to zero for small values of  $n$  ( $n > 10$ , for example).

## 5.2 Number of new overlapping intersections to process

However, although the probability of overlapping clauses  $P_{overlap}$  is very close to zero for  $k$ -SAT instances with very *dense* clauses ( $k \rightarrow n$ ), the number of possible intersections among  $m$  clauses may not be zero:  $\binom{m}{2}$  intersections of 2 clauses  $\{\bar{k}_j\} \cap \{\bar{k}_k\}$ ,  $\binom{m}{3}$  intersections of 3 clauses  $\{\bar{k}_j\} \cap \{\bar{k}_k\} \cap \{\bar{k}_l\}$ , and so on.

Therefore, the expected number of new clauses added to  $P$  in the algorithm that need to be processed is  $\binom{m}{2} \times P_{overlap}$  intersections of 2 clauses  $\{\bar{k}_j\} \cap \{\bar{k}_k\}$ , that may be significative if  $P_{overlap}$  is not small enough. Those new non-overlapping intersection clauses will be intersected again with all  $\{\bar{k}_l\}$  clauses. Fortunately, the probability of overlapping for these intersections of 3 clauses  $\{\bar{k}_j\} \cap \{\bar{k}_k\} \cap \{\bar{k}_l\}$  will be smaller because the non-overlapping intersections of 2 clauses will be more dense because, in average, the intersections of 2 clauses will have a 50% more literals than the intersected clauses.

## 5.3 Complexity for the average case of $k$ -SAT dense

The complexity of SARRIGUREN algorithm for SAT, where there are clauses of any size is obviously exponential because there are  $\sum_{j=1}^m \binom{m}{j} = (2^m - 1)$  possible intersections among  $m$  clauses. In this section, the complexity of the algorithm is going to be analyzed but, only for the average case of random  $k$ -SAT with dense clauses. First, it will be discussed how dense have to be the clauses so that their intersections do not overlap, and then, the complexity of the algorithm for such dense clauses will be analyzed.

### 5.3.1 Dense clauses do not overlap for the average case

The formula  $P_{overlap}$  used to estimate the number of new overlapping intersections to process in the algorithm is valid for an average case with  $m$  arbitrary clauses formed by  $k$  literals of  $n$  variables. Moreover, it has been also shown that the probability  $P_{overlap}$  is zero for very dense clauses with many variables ( $k \rightarrow n$  and  $n \rightarrow \infty$ ). However, it is very interesting to know what happens when the density of the clauses is not 100%, because that can give an idea of the complexity of the algorithm for different values of  $n$  variables and  $k$  literals.

Table 3 shows the probability  $P_{overlap}$  of obtaining overlapping intersections among clauses with  $k$  literals of  $n$  variables and the number of overlapping intersections of 2 and 3 clauses (chosen from  $m = n \times 100$  clauses) that require to be processed in the rest of the algorithm:  $\binom{m}{2} \times P_{overlap}$  and  $\binom{m}{3} \times P_{overlap}$ . It can be seen that with densities over 0.5, the number of new overlapping clauses added is 0 almost in all cases where  $n$  is greater than 100. With densities of 0.25 it is also 0 when  $n$  is greater than 800. Even with densities of 0.1 and 0.5 that number of new clauses added is 0, but in those cases  $n$  has to be greater than 5000 and 20000, respectively.

Let us think if this is an intuitive result. If we have a clause  $c$  with  $k$  literals of  $n$  variables, and we build another one by taking variables from a box, and then deciding if the literal is positive or negative by throwing a coin, then there is a probability  $\frac{k}{n} \times \frac{1}{2} = \frac{k}{2n}$  of obtaining a complementary literal for the first literal of  $c$ . Although there are  $k$  possibilities to get a complementary one for any of the  $k$  literals of  $c$ , the probability of overlapping is not  $\frac{k^2}{2n}$  ( $k \times \frac{k}{2n}$ ) because the extractions of variables are not independent events: the probability  $\frac{k}{2n}$  of obtaining a complementary literal for the first literal changes for the following extractions that depend on the previous ones. In any case, in table 3 it can be noticed that when  $\frac{k^2}{2n} \geq 25$  (approximately when  $k \geq 7\sqrt{n}$ ) the probability of overlapping and the number of overlapping conjunctions is 0.000000000<sup>5</sup> (practically zero). So, the value  $\frac{k^2}{2n} \geq 25$  can serve as an easier way to decide if clauses are dense enough. Notice that the  $\frac{k^2}{2n}$  value for 3-SAT with 100 variables would be only 0.045, very far from 25, what means that 3-SAT clauses are not dense enough and that the complexity of the algorithm is exponential for 3-SAT.

### 5.3.2 Complexity of the algorithm with dense and disjoint clauses

Working with such dense clauses, the algorithm processes each clause and calculates its size. The intersections of each clause with all the other  $m - 1$  clauses have to be performed, although it is only to find that they are disjoint. To process all pairs of  $m$  clauses requires  $O(m^2)$  time. To calculate the size of a clause requires  $O(1)$  time, unless it has to be made by counting the number of literals that would require  $O(k)$  time. The intersection of clauses consists on a merge of 2 sorted clauses of  $k$  literals of  $n$  variables that can be processed in  $O(k)$  time because the literals are sorted by order of variable. Taking this into account, the complexity would be  $O(m^2 \times k)$ , but that is only for clauses with small values of  $k$ .

In fact, time complexity of the merging process is lower than  $O(k)$  for clauses with bigger values of  $k$ . The merging process will stop as soon as the complementary literals are found in the sorted clauses that are being merged. This can be approached with a geometric distribution [13] of a random variable that is the number of trials (number of literals to scan in the merging process) needed to get one success (to find

---

<sup>5</sup>The real values in table 3 are 4.90e-25, 4.93e-15, 1.79e-14, 4.97e-13, 3.83e-12 and 7.36e-12.

the first pair of disjoint literals), where the probability of success of each experiment is  $p = \frac{k}{2n}$ . For the geometric distribution it is known that  $1/p$  is the average or expected value, that is, the expected number of trials to find the first disjoint literal is  $\frac{2n}{k}$ . This means that for a density of 0.9 with 100 variables about 2 literals should be scanned ( $\frac{2 \times 100}{90} = 2.2$ ), and for a density of 0.1 with 100 variables, 20 literals should be scanned ( $\frac{2 \times 100}{10} = 20$ ). Therefore, the merging process can be processed in  $O(n/k)$  time<sup>6</sup>.

In summary, it can be stated that SARRIGUREN is a complete algorithm for SAT that offers, in the average case, a polynomial complexity  $O(m^2 \times n/k)$  for instances of  $k$ -SAT with  $m$  clauses that are relatively dense ( $k \geq 7\sqrt{n}$ ).

$n$	$k$	$density$ ( $k/n$ )	$\frac{k^2}{2n}$	$P_{overlap} = P_{ov}$ $k_c=k_d=k$	$\binom{m}{2}.P_{ov}$ $k_{c,d}=k$	$\binom{m}{3}.P_{ov}$ $k_c=k$ $k_d=1.5k$
10	9	0.9	4.0	0.0037109375	1853.6	2292.9
100	90	0.9	40.5	0.0000000000	0.0	0.0
100	70	0.7	24.5	0.0000000000	0.0	0.0
100	50	0.5	12.5	0.0000001348	6.7	0.0
200	100	0.5	25.0	0.0000000000	0.0	0.0
200	50	0.25	6.2	0.0008437706	1.69e5	9.49e5
400	100	0.25	12.5	0.0000007070	565.6	5.3
800	200	0.25	25.0	0.0000000000	0.0	0.0
1000	100	0.1	5.0	0.0052125370	2.61e7	4.53e9
5000	500	0.1	25.0	0.0000000000	0.5	0.0
1000	50	0.05	1.2	0.2776349687	1.39e9	1.28e13
20000	1000	0.05	25.0	0.0000000000	14.7	0.0

Table 3: Overlapping probabilities and expected number of new intersected clauses for different values of  $k$  and  $n$ , where  $m = n * 100$

#### 5.4 Best and worst-cases for the algorithm

The best-cases for SARRIGUREN executed over instances of  $k$ -SAT with relatively dense clauses are cases where all the clauses are disjoint among them, and the complementary literals are found during the merging as soon as possible, because those complementary literals correspond to the first variables. Some examples of best-cases are shown in table 4. In any case, these best cases should not be much more efficient than the average cases with densities satisfying  $k \geq 7\sqrt{n}$ , because for those cases almost all clauses should be disjoint among them. The execution of the  $O(n/k)$  merging processes could be slower because the complementary literals do not necessarily correspond to the first variables.

With respect to the worst-cases, the situation is very different. That happens when all the clauses overlap among them, that is, when there are no complementary literals and the majority of the clauses have repeated literals. As the clauses are dense,

---

<sup>6</sup>The number of literal scans does not exactly follow a geometric distribution, because  $p$  is not the same for all trials. That probability  $p$  grows with every fail:  $\frac{k}{2n}, \frac{k}{2(n-1)}, \frac{k}{2(n-2)}, \dots$  This means that the expected value is lower than  $1/p = \frac{2n}{k}$ . Anyway, it is also in  $O(n/k)$

$x_1$	$x_2$	$x_3$	$\dots$	$x_{k-1}$	$x_k$	$\dots$	$x_n$
0							<i>choose k - 1 literals with values 0 or 1</i>
1	0						<i>choose k - 2 literals with values 0 or 1</i>
1	1	0					<i>choose k - 3 literals with values 0 or 1</i>
1	1	1					$\dots$
1	1	1	$\dots$	0			<i>choose 1 literal with value 0 or 1</i>
1	1	1	$\dots$	1	1		

Table 4: Examples of best cases for the algorithm

$x_1$	$x_2$	$x_3$	$\dots$	$x_{k-1}$	$x_k$	$x_{k+1}$	$\dots$	$x_n$
1	1	1	1	1	1			
1	1	1	1	1		1		
1	1	1	1	1			$\dots$	
1	1	1	1	1				1

Table 5: Example of worst-case for the algorithm

means that there will be many repeated literals in those clauses. In table 5 it is shown an example of worst-case where all the possible intersection of 2 clauses among the  $n - k + 1$  clauses overlap. In that case the total number of possible intersections is in the order  $O(2^{n-k+1})$ , that even if clauses are 0.5 dense with  $n = 200$  and  $k = 100$  is a huge number.

At this point, it is important to say that the probability of finding  $m$  random dense clauses, such as all  $2^{m-1}$  possible intersections among those clauses are overlapping can be considered zero. It is obvious that if the probability of two dense clauses to overlap is 0.0000000000 (less than 1.0e-11) as shown in section 5.3.1, then the probability of these worst-cases for random  $k$ -SAT is exponentially much smaller, around  $(1.0e-11)^{2^{m-1}}$ .

Anyway, these pure worst-cases for SARRIGUREN working with relatively dense clauses and many repeated literals are trivial cases by using other methods. For example, in this case all the clauses are satisfied by assigning  $x_1 = 1$ . It is true that there could be other clauses with literal  $\bar{x}_1$ , and other clauses without that literal. In both cases, those clauses should still be dense clauses with repeated literals and they could be satisfied by using assignments for other variables. However, there could be many other dense clauses with non repeated literals, that would constitute bad cases for such assigning methods. Therefore, this should be studied further, but in any case, all these cases with overlapping clauses have a practically zero probability in random  $k$ -SAT with dense clauses.

## 6 Experimental Results

In this section some experimental results are presented. The algorithm has been programed in Python (see appendix A), a programming language that provides support for big integers, and it has been executed in a machine with Intel Core 3.60 GHz processor, 64GB RAM, Linux Mint 19.3 Trician OS and Python 2.7. The Python program and all the input sets and obtained results in the experiments are maade

available in [8].

SARRIGUREN has been tested with several sets of randomly generated  $k$ -SAT sets of  $N$  variables (20000, 5000, 800, 200 and 100),  $M$  number of clauses (100, 1000, 10000 and 100000) and the  $K$  values corresponding to different density types ( $0.9N$ ,  $7\text{Root}N$ ,  $6\text{Root}N$ ,  $5\text{Root}N$ ,  $4\text{Root}N$  and  $3\text{Root}N$ ). The density type  $0.9N$  is always 0.9, independently of the value of  $N$ . For example, the  $K$  of a  $0.9N$  density type with  $N=20000$  variables is  $\lfloor 0.9 \times 20000 \rfloor = 18000$  that corresponds to a  $18000/20000=0.9$  density. For the rest of density types  $x\text{Root}N$ , the value of  $K$  is  $\lfloor x \times \sqrt{N} \rfloor$ . The density ( $K/N$ ) depends on  $N$ , and it decreases for bigger values of  $N$ . For example, the  $K$  for a  $7\text{Root}N$  density type with  $N=5000$  variables is  $\lfloor 7 \times \sqrt{5000} \rfloor = 494$  that corresponds to a density of  $494/5000=0.01$ . Every experiment for a combination of  $N$ ,  $M$  and  $K$  has been repeated 20 times for almost all the cases with 100 and 1000 clauses, and for the other cases that number of executions has been reduced to 3, 2 or 1.

Figures 1 and 2 contain the graphics that show the average times employed by the algorithm for all the experiments, one graphic for each value of  $M$  (figure 1) and one graphic for the different density types (figure 2). In these graphics it can be seen that the efficiency is better for higher densities:  $0.9N$ , then  $7\text{Root}N$ ,  $6\text{Root}N$  and going down until  $3\text{Root}N$ . In fact for the less dense ones ( $4\text{Root}N$  and  $3\text{Root}N$ ) that do not appear in figure 1, some executions (with  $M=100000$ ) took so long that the processes were killed before finishing. It is also easy to see that experiments were more efficient for less number of variables. Moreover, the bigger is the number of variables, then the bigger are the differences among the density types. That is because the concrete density corresponding to a density type decreases when the number of variables grows. For example, for  $6\text{Root}N$  density type the density for 20000 variables is  $\lfloor 6 \times \sqrt{20000} \rfloor / 20000 = 848/20000 = 0.042$  while the density for 100 variables is  $\lfloor 6 \times \sqrt{100} \rfloor / 100 = 60/100 = 0.6$ . Notice that for the  $0.9N$  density type, there are no great time differences for the different number of variables, because in all cases the concrete density is the same: 0.9.

It is not surprising that the results for the  $3\text{Root}N$  and  $4\text{Root}N$  are very bad because the number of overlapping clauses increases a lot for those density types. In table 6 it can be seen that, in average, the number of overlapping clauses for the  $3\text{Root}N$  case grows 51 times more than the number of clauses  $M$ , and for the  $4\text{Root}N$  case it grows more than 4 times. For the  $5\text{Root}N$  and  $6\text{Root}N$  cases, there are some overlapping clauses, but the efficiency does not get significantly affected. Notice that there are no overlapping clauses for the cases  $7\text{Root}N$  and  $0.9N$ , as explained in section 5.3.1.

DT (density type)	OV#/CLAUSES#
0.9N	0
7RootN	0
6RootN	0.0002
5RootN	0.0664
4RootN	4.21
3RootN	51.4535

Table 6: Growth ratio of the number of clauses due to overlapping intersections.

At this point it is interesting to check if the polynomial time complexity  $O(m^2 \times n/k)$  analyzed in section 5.3.2, is compatible with the obtained results in these exper-

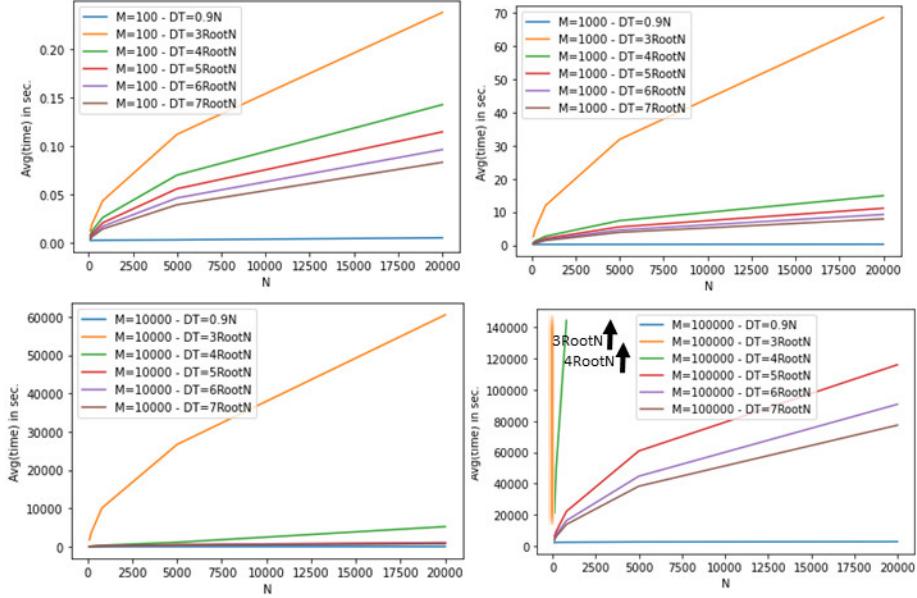


Figure 1: Graphics for each  $M$  showing times depending on  $N$  and density types

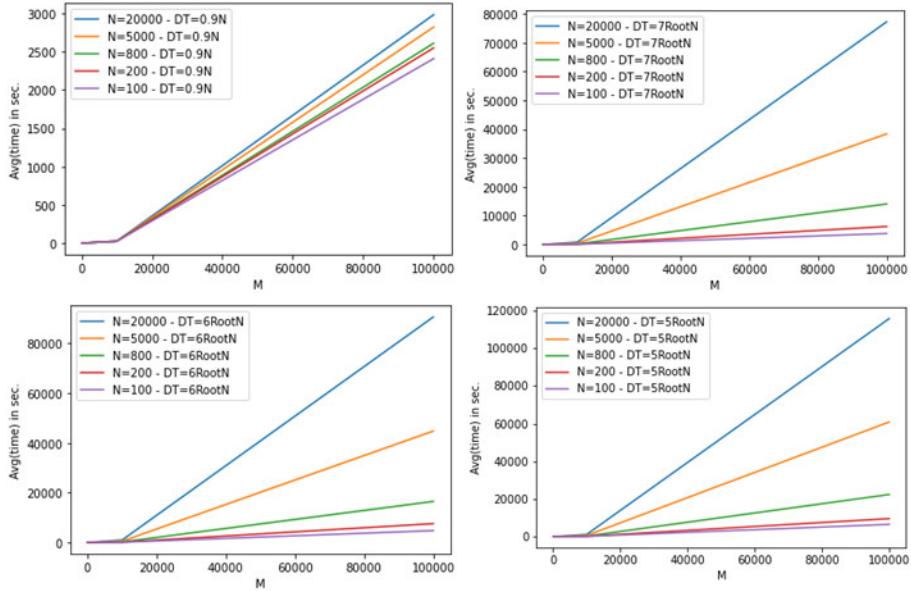


Figure 2: Graphics for each density type showing times depending on  $N$  and  $M$

iments. That polynomial complexity was only for instances of  $k$ -SAT with  $m$  clauses relatively dense ( $k \geq 7\sqrt{n}$ ), that is, it should be compatible at least with the experiments performed with 7RootN and 0.9N densities.

Let us start checking if the  $O(m^2)$  part of the  $O(m^2 \times n/k)$  complexity verifies or not. For similar values of  $n/k$ , when  $m$  is 10 times bigger, then the expected time should be  $10^2 = 100$  bigger. Table 7 shows, for different M values and density types<sup>7</sup> DT, if the average time for an M value (1000, 10000 and 100000) is around  $10^2 = 100$  times bigger than the average time for the M/10 value (100, 1000 and 10000). It can be considered compatible for the 0.9N and 7RootN density types, as expected. Moreover, it is also compatible for 6RootN and 5RootN, because the number of overlapping clauses is quite low. Notice that the Pearson correlation coefficient for these cases is 0.997495124. However, it cannot be considered compatible for 4RootN and 3RootN, also as expected due to the big number of overlapping clauses.

M	DT	Avg(time)	Expected value: time(M/10)*100	$O(M^2)$ compatible?
100	0.9N	0.00303		
1000	0.9N	0.24910	0.30381	Yes
10000	0.9N	23.84	24.91	Yes
100000	0.9N	2675.83	2384.17	Yes
100	7RootN	0.02925		
1000	7RootN	2.84	2.93	Yes
10000	7RootN	275.27	283.55	Yes
100000	7RootN	27934.50	27526.53	Yes
100	6RootN	0.03431		
1000	6RootN	3.34	3.43	Yes
10000	6RootN	325.10	333.92	Yes
100000	6RootN	32810.41	32509.71	Yes
100	5RootN	0.04122		
1000	5RootN	4.03	4.12	Yes
10000	5RootN	389.40	403.50	Yes
100000	5RootN	42989.82	38940.07	Yes
100	4RootN	0.05173		
1000	4RootN	5.41	5.17	Yes
10000	4RootN	1398.31	540.85	No
100	3RootN	0.08506		
1000	3RootN	23.93	8.51	No
10000	3RootN	20551.57	2392.50	No

Table 7: Is the average time the expected one for  $M^2$  and density type?

Now the goal is to check if the  $O(n/k)$  part of the  $O(m^2 \times n/k)$  complexity is compatible or not with the experiments. For that analysis it is going to be assumed that the results of the experiments for the different configurations of N and M for the 0.9N density type are not affected by the N/K term. In fact, they are the most efficient

---

<sup>7</sup>The  $n/k$  values for a density type are different, but average times of all of them are used.

ones and its N/K value is close to 1. Table 8 shows for the different number of variables N and density types DT (that determine the K values and therefore their N/K value) which are the average times of the experiments with all values of M and how many times bigger are these values compared to the values for the same configuration and density type 0.9N. If those results are about N/K times bigger than the corresponding result of 0.9N, then that would mean that results are compatible with the  $O(n/k)$  part of the  $O(m^2 \times n/k)$  complexity. And it can be considered that it is compatible for the 7RootN, 6RootN and 5RootN density types. Notice that the Pearson correlation coefficient for these cases is 0.994232519.

Therefore, it can be affirmed that the analyzed time complexity  $O(m^2 \times n/k)$  is compatible not only with the density types 0.9N and 7RootN, but also with 6RootN and 5RootN density types.

N	DT	Avg(time) for N/K and DT	N/K	Ratio w.r.t. 0.9N case with same N	$O(N/K)$ compatible?
20000	0.9N	752.33	~1		
5000	0.9N	711.68	~1		
800	0.9N	658.89	~1		
200	0.9N	643.58	~1		
100	0.9N	608.43	~1		
20000	7RootN	19520.14	20.22	25.95	Yes
5000	7RootN	9691.06	10.12	13.62	Yes
800	7RootN	3538.73	4.06	5.37	Yes
200	7RootN	1565.04	2.04	2.43	Yes
100	7RootN	950.81	1.43	1.56	Yes
20000	6RootN	22886.57	23.58	30.42	Yes
5000	6RootN	11300.90	11.79	15.88	Yes
800	6RootN	4151.87	4.73	6.30	Yes
200	6RootN	1896.80	2.38	2.95	Yes
100	6RootN	1187.45	1.67	1.95	Yes
20000	5RootN	22886.57	28.29	38.86	Yes
5000	5RootN	11300.90	14.16	21.57	Yes
800	5RootN	4151.87	5.67	8.54	Yes
200	5RootN	1896.80	2.86	3.72	Yes
100	5RootN	1187.45	2.00	2.66	Yes

Table 8: Is the average time N/K times bigger than the 0.9N case of same N?

## 7 Finding the solutions

SARRIGUREN algorithm calculates the number of solutions that satisfy a set  $K$  of clauses in CNF format. The algorithm is useful to know if  $K$  is unsatisfiable (when the number of solutions is 0) or satisfiable (when the number of solutions is greater than 0). It is not useful to know a concrete solution that makes  $K$  satisfiable, that is, a set of literals that satisfy all the clauses in  $K$ . In other words, it is a complete algorithm to

solve #SAT but not an algorithm that provides the solutions to a SAT problem. But that is something that can be easily achieved by using the SARRIGUREN algorithm.

SARRIGUREN-SOL is an algorithm that gets the solutions that make  $K$  satisfiable, once it is known that  $K$  is satisfiable. That algorithm proceeds in this way: it selects literals corresponding to the variables in the initial set of satisfiable clauses. For each chosen literal, it checks if there are solutions for the clauses that are not satisfied with that literal. If there are solutions, then the literal is added to the solution, and if not, the complement of the literal is added to the solution.

---

**Algorithm 2** SARRIGUREN-SOL

---

**Input:**  $K = k_1 \dots k_m$ , a set of disjunctive clauses sorted by variable number  
 $n$ : number of variables

**Output:** Set of literals that satisfy  $K$

**Precondition:** SARRIGUREN( $K, n$ ) > 0, that is,  $K$  is SAT

```

 $vars \leftarrow \langle x_1, \dots, x_n \rangle$ 
 $solution \leftarrow []$ 
while  $vars$  is not empty and  $K$  is not empty do
     $lit \leftarrow$  select literal ( $x_i$  or  $\bar{x}_i$ ) and remove  $x_i$  from  $vars$ 
     $K' \leftarrow$  clauses of  $K$  without  $lit$  or with  $\bar{lit}$  which is removed
    if  $\square$  is not in  $K'$  and SARRIGUREN( $K', \text{length}(vars)$ ) > 0 then
        add  $lit$  to  $solution$ 
    else
        add  $\bar{lit}$  to  $solution$ 
         $K' \leftarrow$  clauses of  $K$  without  $\bar{lit}$  or with  $lit$  which is removed
    end if
     $K \leftarrow K'$ 
end while
return  $solution$ 

```

---

The instruction “select literal ( $x_i$  or  $\bar{x}_i$ ) and remove  $x_i$  from  $vars$ ” has been left quite general, because different possibilities can be implemented. For example, a priority list of desired literals to be part of the solution could be provided.

Moreover, there is another algorithm that provides additional information about the solutions. The algorithm SARRIGUREN-SOL-LITS calculates the number of solutions for each literal.

---

**Algorithm 3** SARRIGUREN-SOL-LITS

---

**Input:**  $K = k_1 \dots k_m$ , a set of disjunctive clauses sorted by variable number  
 $n$ : number of variables

**Output:** Set of literals that satisfy  $K$

**Precondition:** SARRIGUREN( $K, n$ ) > 0, that is,  $K$  is SAT

```
for each lit in < $\overline{x}_1, x_1, \dots, \overline{x}_n, x_n$ > do
     $K' \leftarrow$  clauses of  $K$  without  $lit$  or with  $\overline{lit}$  which is removed
    numSolutions  $\leftarrow$  SARRIGUREN( $K', n - 1$ )
    print  $lit$  "has" numSolutions "solutions"
end for
```

---

## 8 Conclusions

In this paper a new complete algorithm for SAT (and also for SAT variations such as Unique-SAT or propositional model counting #SAT) called SARRIGUREN has been described, analyzed and tested. The Python implementation and all the input datasets and obtained results in the experiments are made available.

That algorithm has an  $O(m^2 \times n/k)$  time complexity for random  $k$ -SAT instances of  $n$  variables and  $m$  dense clauses, where dense means that  $k \geq 7\sqrt{n}$ . With that density the number of overlapping clauses found in the algorithm can be considered zero. Notice that, under these conditions, a clause with 20000 variables and 989 literals (and therefore a density of  $989/20000=0.049$ ) is considered dense enough so that there are no overlapping clauses. Moreover, even for less dense clauses  $k \geq 5\sqrt{n}$ , that  $O(m^2 \times n/k)$  complexity is also true because the number of overlapping clauses remains to be very small, what means that for example a clause with 20000 variables and 707 literals (0.035 density) is also considered dense. Although theoretically there could be worst-cases with exponential complexity, the probability of those cases to happen in random  $k$ -SAT with dense clauses can also be considered zero (even smaller than the probability of having overlapping clauses, that is zero).

One disadvantage of this complete algorithm compared to others based on inference rules is that, no understandable explanation can be offered when the set of clauses is detected as unsatisfiable; only that the  $2^n$  unsatisfiable clauses have been counted.

Two complementary algorithms, SARRIGUREN-SOL and SARRIGUREN-SOL-LITS, have also been presented that provide the solutions to  $k$ -SAT instances and valuable information about number of solutions for each literal.

Although SARRIGUREN does not solve the NP=P problem because it is not a polynomial algorithm for 3-SAT, it broads the knowledge about that subject. The assumption that  $k$ -SAT does not have subexponential algorithms for  $k \geq 3$  has to be revisited, according to this work.

Finally, it has not escaped my knowledge that SARRIGUREN can be modified in order to obtain new heuristic SAT algorithms and parallel SAT algorithms.

On the one hand, for the case of heuristic SAT algorithms, it is quite obvious that if  $k \geq 5\sqrt{n}$ , the probability of finding overlapping clauses is very close to zero (practically zero if  $k \geq 7\sqrt{n}$ ). In that case, the part of the algorithm where intersections are calculated can be omitted, and the complexity would drastically drop to a linear  $O(m)$ . SARRIGUREN-SOL algorithm would provide solutions that could be easily tested, also with  $O(m \times n)$  complexity.

On the other hand, for the case of parallel algorithms, it is easy to see that the number of clauses can be calculated by levels. In a first level,  $\sum_{i=1}^m |\{\bar{k}_i\}|$  can be calculated by processors that compute  $|\{\bar{k}_i\}|$  in parallel and report the results to a node that performs the sum. Once that level is finished, the same can be done to compute the negative cardinalities  $-\sum_{j,k: 1 \leq j < k \leq m} |\{\bar{k}_j\} \cap \{\bar{k}_k\}|$ . And after that, the level that calculates again positive cardinalities:  $+\sum_{j,k,l: 1 \leq j < k < l \leq m} |\{\bar{k}_j\} \cap \{\bar{k}_k\} \cap \{\bar{k}_l\}|$ , and so on. Notice that, if after the completion of a level that reports negative cardinalities the value  $2^n$  is reached, then the parallel computation can end and return UNSAT.

## A Python code of SARRIGUREN

```

def pattern(c):
    return c

def cardinality(c,n):
    return 2**n-len(c)

def contrary(sign):
    return '+' if sign=='-' else '-'

def intersect(c,d):
    result = []
    i = j = 0
    while i < len(c) and j < len(d):
        if c[i] == -d[j]:
            return []
        elif c[i]==d[j]:
            result.append(c[i])
            i += 1
            j += 1
        elif abs(c[i])<abs(d[j]):
            result.append(c[i])
            i += 1
        else:
            result.append(d[j])
            j += 1
    result.extend(c[i:])
    result.extend(d[j:])
    return result

def filterClauses(K,l):
    Kprime=[]
    for c in K:
        if -l in c:
            d=[]
            for lit in c:
                if lit!=l: d.append(lit)
            if d==[]: return False # K UNSAT
            Kprime.append(d)
        elif l not in c:
            Kprime.append(c)
    return Kprime

def sarriguren(K,n,getNumOverlapping=False):
    # Precondition: K disjunctive clauses
    # sorted by variable number
    P=[]
    u=0
    m=len(K)
    for i in range(0,m):
        ap=pattern(K[i])
        N=[]
        u+=cardinality(ap,n)
        N.append([i,ap])
        for sp in P:
            ip=intersect(ap,sp[1])
            if ip!=[]:
                if sp[0]=='+': u+=cardinality(ip,n)
                else: u-=cardinality(ip,n)
                N.append([contrary(sp[0]),ip])
        P=P+N
        if u==2**n:
            ifgetNumOverlapping:
                return [0,len(P)-m]
            return 0
        if getNumOverlapping:
            return [2**n-u,len(P)-m]
    return 2**n-u

def sarriguren_sol(K,n,vars=[]):
    # Precondition: sarriguren(K,n)>0
    # vars is priority list of literals to assign
    # (it must have exactly the n literals)
    solution=[]
    if vars==[]: # random literals tried
        for i in range(1,n+1):
            if random.randint(0,1)==0: vars.append(i)
            else: vars.append(-i)
    while vars!=[] and K!=[]:
        l=vars[0]
        vars.remove(l)
        Kprime=filterClauses(K,l)
        if Kprime and sarriguren(Kprime,len(vars))>0:
            solution.append(l)
        else:
            solution.append(-l)
            Kprime=filterClauses(K,-l)
            K=Kprime
    return solution

def sarriguren_sol_lits(K,n):
    # Precondition: sarriguren(K,n)>0
    Sols=[]
    for lit in range(1,n+1):
        for sign in [-1,1]:
            Kprime=filterClauses(K,lit*sign)
            if Kprime==False: numSolutions=0
            else: numSolutions=sarriguren(Kprime,n-1)
            Sols.append([lit*sign,numSolutions])
    return Sols

```

## References

- [1] BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. *Handbook of Satisfiability* (2009), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press.
- [2] BRUEGEMANN, T., AND KERN, W. An improved deterministic local search algorithm for 3-sat. *Theor. Comput. Sci.* 329, 1–3 (dec 2004), 303–313.
- [3] COOK, S. A. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium* (New York, 1971), ACM, pp. 151–158.
- [4] DANTSIN, E., GOERDT, A., HIRSCH, E. A., KANNAN, R., KLEINBERG, J., PAPADIMITRIOU, C., RAGHAVAN, P., AND SCHÖNING, U. A deterministic  $(2/(k+1))n$  algorithm for k-sat based on local search. *Theoretical Computer Science* 289, 1 (2002), 69–83.
- [5] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A machine program for theorem-proving. *Commun. ACM* 5, 7 (jul 1962), 394–397.
- [6] DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *J. ACM* 7, 3 (jul 1960), 201–215.
- [7] EVEN, S., ITAI, A., AND SHAMIR, A. On the complexity of timetable and multi-commodity flow problems. *SIAM Journal on Computing* 5, 4 (1976), 691–703.
- [8] GoÑI, A. Sarriguren algorithms, datasets and results. <https://goo.su/zV3Pt6E>. Accessed: 2024-01-12.
- [9] HANSEN, T. D., KAPLAN, H., ZAMIR, O., AND ZWICK, U. Faster k-sat algorithms using biased-ppsz. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (New York, NY, USA, 2019), STOC 2019, Association for Computing Machinery, p. 578–589.
- [10] IMPAGLIAZZO, R., AND PATURI, R. On the complexity of k-sat. *Journal of Computer and System Sciences* 62, 2 (2001), 367–375.
- [11] KROM, M. R. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly* 13 (1967), 15–20.
- [12] MÉZARD, M., PARISI, G., AND ZECCHINA, R. Analytic and algorithmic solution of random satisfiability problems. *Science* 297, 5582 (2002), 812–815.
- [13] RAIKAR, S. P. “geometric distribution”. encyclopedia britannica. <https://www.britannica.com/topic/geometric-distribution>. Accessed: 2024-01-08.
- [14] SANE, S. S. *The inclusion-exclusion principle*. Hindustan Book Agency, Gurgaon, 2013, pp. 57–79.
- [15] STÅLMARCK, G., AND SÄFLUND, M. Modeling and verifying systems and software in propositional logic. *IFAC Proceedings Volumes* 23, 6 (1990), 31–36. IFAC/EWICS/SARS Symposium on Safety of Computer Control Systems 1990 (SAFECOMP’90), Gatwick, UK, 30 October–November 2, 1990.