SAT Instance Analysis

Joseba Celaya

Montserrat Hermo, Advisor

A thesis submitted
in order to obtain the
Bachelor Degree of
Computer Science

University of Basque Country
September 5, 2021

eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática
Computación

Trabajo de Fin de Grado

SAT Instance Analysis

Autor
Joseba Celaya

Directora
Montserrat Hermo

2021

In memory of my father

Abstract

During last 20 years, the advances on SAT solving techniques has lead us to overcome theoretical complexity worst case scenarios for big varieties of families and distributions of SAT instances. SAT solvers based on conflict-driven clause learning (CDCL), for example, are shown to be effective on large industrial benchmarks with millions of variables each. But, the reasons why techniques like this are successful with certain classes of instances remains unknown, and under research. In order to understand the great performance of state of the art SAT solving techniques, there's been a big effort on characterising the structure of such SAT formulas.

In this work, we introduce a tool for studying formulas with a given structure, and extracting some of the features discussed in recent bibliography.

Acknowledgements

I would like to thank to my project advisor Montse Hermo, for accepting to direct my thesis and for helping and guiding my academic decisions in, what it has been by far, the toughest year of my life.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
1.1 Brief Algorithm History 3
1.2 Development 4
1.3 Characterising SAT formulas 4

2 Statistical Approach **7**
2.1 Graph Transformations 9
2.2 Community Structure 11
2.3 Scale-Free Property 12
2.4 Self-Similar Structure 13

3 The Tool **15**
3.1 Tool Description 15
3.2 Motivation 16
3.3 Objectives 17
3.3.1 Main Objectives 17
3.3.2 Secondary Objectives 18
3.4 Research and Development Workflow 19
3.4.1 Research 19
3.4.2 Code Development 22
3.5 Project Outline 24
3.6 Functionalities 25
3.6.1 backdoors 25
3.6.2 cnf.py 26
3.6.3 feat.py 27
3.6.4 gen.py 27
3.6.5 io.py 28

3.7 Future Work 29
3.7.1 Do One Thing and Do It Well 30
3.7.2 Pure C 30
3.7.3 Share 30

4 Conclusions **31**

Appendix A kanban2csv.py **33**

Appendix B sia.bat and setup.sh **39**

Chapter 1

Introduction

”To speak is to fall into tautology.”

Jorge Luis Borges, *The Library Of Babel*

A *propositional logic*, or *propositional calculus* formula, is a sentence belonging to the *formal language of propositional logic (PL)*. Here is a propositional logic formula:

$$(p \implies q) \land p$$

Formulas can be transformed applying inference rules:

$$(p \implies q) \land p$$

$$(\neg p \lor q) \land p$$

$$q$$

Propositional logic formulas, can be transformed into equivalent formulas called *Conjunctive Normal Form (CNF)*, which are formulas composed by *conjunctions* of *disjunction* of literals:

$$\phi = \bigwedge_{i=1}^{m} C_i, \quad C_i = \bigvee_{j=1}^{k} x_j$$

The fact that an arbitrary formula can be translated into a CNF is very powerful. In advance, we will refer to *propositional logic formulas in CNF* just as *CNF Formulas*, or *SAT Instances*. CNF Formulas can also be defined in term of sets, so a formula is a set, and each clause another set containing literals:

$$\Gamma = \{\{x_1, x_2, x_3\}, \{\bar{x_1}\}, \{\bar{x_2}\}\}$$

[1]

The Boolean Satisfiability Problem, or SAT for short, is the problem of deciding whether a given *propositional logic* formula has a variable assignment that makes the formula true. There is a similar problem definition that is applied on CNF formulas, with clauses of fixed size k. This problem is called k-SAT.

It was the first known $\mathcal{NP}$-Complete program, meaning that every other problem contained in $\mathcal{NP}$ can be reduced to it. This has made the problem one of the central problems of research in *complexity theory* and computer science.

Lots of problems arising in other areas such as combinatorial optimisation can be studied as SAT problems. Anyway, algorithms for specific domains usually perform better, than solving the translation into a SAT instance, and then solving it.

[1] This notation is going to be used further. It can also use positive and negative integers to denote variables and negated variables.

1.1 Brief Algorithm History

The first algorithm introduced is from 1960. Is the *Davis-Putnam (DP)* algorithm. It's an algorithm that makes use of *resolution* for checking the validity of a given formula. It can be alternatively defined as a *proof system*. Resolution's space complexity grows exponentially with respect to the length of the formula; so this make using the algorithm in big scale unpractical.

In 1962, an improved algorithm called *Davis-Putnam-Logemann-Loveland (DPLL)* appeared [13], which introduced the use of two *heuristics*, the *unit propagation*, and the *pure literal rule*. Also, it based on building a solution from travelling a tree like space search. The running time of this procedure is in $\mathcal{O}(2^n)$. But in practice, the average running times are bounded polynomially. This algorithm scheme is the basis of most modern SAT solvers.

In 1996, *Conflict Drive Clause Learning* algorithms appeared [30]. The core idea of these procedures, is to whenever a conflict is found searching in a branch, build an *implication graph*, find the cut that lead to a conflict, obtain a new clause which is the negation of the assignment that lead to a conflict, get back in the search tree, and add the created clause to the original formula, in order to prune that part of the search space, and not entering again during the search.

Theses are the most used algorithms in the context of *SAT Solving*, but there are a lot of powerful algorithms based on other schemes. For instance, we have algorithms based on *greedy local search* like *GSAT* [20]. These types of algorithms are incomplete.

There are also *SAT Solvers* based on *parallel computation* like *GPU4SAT* [14] and *matrix based solvers* like *MatSat* [27] that are inspired by *Neural Network* systems, which can also be differentiable!

1.2 Development

Industrial SAT instances are the ones which solubility has an industrial or practical application; for example, *cryptography*, *hardware verification*, *software verification*, etc. SAT solvers are good solving industrial SAT instances. Their improvement in efficiency is motivated by the introduction of *lazy data-structures*, *learning mechanisms* and *activity based heuristics*.

1.3 Characterising SAT formulas

Characterising what makes a SAT instance difficult to solve is a tough task. Analysing a SAT formula yields to trying to characterise its ***structure and hardness***, and how does this relate to the efficiency of specific SAT solving methods. In the literature, two approaches have been followed.

The first one, is to use tools from *proof complexity*, for analysing specific SAT solving methods [12]. In this way, we can choose, for example, *resolution* as a *proof system* and produce a refutation statement about a formula from a specific family of SAT formulas. These formulas can be defined syntactically. An example of this are *Horn formulas*, which are formulas containing clauses of no more than one positive literal. This family of formulas, are known to be polynomially solvable since 1978, as stated in *Schaefer's dichotomy theorem* [2] [28] and *linearly* solvable since 1984 [15]. Therefore, identifying whether a formula has any known *syntactic structure* can be useful for designing a better solving strategy.

[2]Special cases of this theorem states that *2CNF*, *Horn*, *Dual-Horn*, *XOR-SAT* and formulas where setting all variables to true or all variables to false satisfies all clauses, are in $\mathcal{P}$.

The second method, is based on extracting statistical features from formulas, that *are believed* to determine the performance of a given solving method over those family of formulas, due to *empirical observation*. An example of this, is the *infamous clause to variable ratio* feature of a *random k-SAT* formula, and the following conjecture:

Definition 1 (**The Threshold Conjecture**)**.** For each k, there is some c' such that for each fixed value of $c < c'$, *random k-SAT* with n variables and cn clauses is satisfiable with probability tending to 1 as $n \to \infty$; and when $c > c'$, unsatisfiable with probability tending to 1.

For $k = 3$, this threshold, if exist, is bounded by $3.003 < c' < 4.598$ [11]. It is observed that for randomly generated *k-SAT* formulas, can be a good measure of hardness for solving methods like *DPLL* [29] procedures, but not that effective for *local search*-like approaches. We will refer to this approach as the ***statistical approach***

We will talk about some different ways to study SAT formulas under the statistical approach.

Chapter 2

Statistical Approach

"Nothing is invented, for it's written in nature first."

Antoni Gaudí

First attempts on trying to identify features for characterising the hardness of a SAT formula were made by *Nudelman, Devkar, Shoham and Leyton-Brown* in 2004 [24] and followed by *Biere and Sinz* [7] in 2006 and *Ansótegui, Bonet, Levy and Manyà* [4] in 2008. These attempts, are usually made for analysing *industrial SAT instances* and *random SAT instances*. First ones, are those that we find in "nature", and therefore, they can be more difficult to understand; since we don't know precisely how they've been generated. This is the reason why a statistical analysis approach can be useful for dealing with them. By other hand, we have *random instances*. These serve as a bottom reference for comparing them with industrial ones, because we understand better their behaviour, as we mentioned in first chapter.

These techniques were applied in their SAT solver *SATzilla* [23], which takes advantage of a wide variety of *statistical features* in order to classify SAT formulas, apply *regression* to predict different algorithm's running times on those formulas, and select the most adequate algorithm according to the classification. Such SAT solving approaches are called *portfolio SAT solving methods*. Our tool will help us to compute some of those features.

Some of these features, are *graph features* computed over the *graph form* of a SAT formula.

Following this direction, the study of *complex networks* has influenced the SAT research community. The *scale-free property* [3], the *community structure* [5] and the *self-similar structure* [1] are some examples.

Let's introduce some definitions in order to continue:

Definition 2 (**Graph**)**.** A graph is a pair $G = (V, E)$ of sets such that $E \subseteq V^2$; thus, the elements graph of E are 2-element subsets of V. The elements of V are called vertices or nodes, and the elements of E edges.

Definition 3 (**Vertex Degree** (k_u))**.** The number of edges incident to a given node u, denoted by $deg(u)$ or k_u.

Definition 4 (**r-partite Graph**)**.** Let $r \geqslant 2$ be an integer. A graph $G = (V, E)$ is called *r-partite* if V admits a partition into r classes such that every edge has its ends in different classes: vertices in the same partition class must not be adjacent.

Definition 5 (**Bipartite Graph**)**.** *r-partite* graph for an r value of 2.

Definition 6 (**Undirected Weighted Graph**[1])**.** Generalisation of a graph. Is a pair (V, w) where V is a set vertices and $w : V \times V \to \mathbb{R}^+$ satisfies $w(x, y) = w(y, x)$. Notice how for a regular graph $w(x, y) = 1$ if $(x, y) \in E$ and 0 otherwise. Vertex degree is defined as $deg(x) = \sum_{y \in V} w(x, y)$. Bipartite graphs are tuples (V_1, V_2, w) where $V_1 \cap V_2 = \emptyset$ and $w : V_1 \times V_2 \to \mathbb{R}^+$.

Definition 7 (**Neighbourhood** (N_i))**.** The graph formed from the set of adjacent vertices of the i, and all of the edges connecting pairs of vertices in that set.

[1]This generalisation is proposed for making further notation presented more fluent

2.1 Graph Transformations

The efforts made on trying to represent CNF formulas in a way that is suitable for visualisation have bring us the necessity of using *graph theoretic* representations of then. In some cases, these representations are *not complete*, in the sense that you can't rebuild the formulas given their graph form. We will use a formula example for the ones we used in the project: *VIG* and *CVIG*.

Let's say we have the following SAT Instance in set notation:

$$\Gamma = \{\{1,2,3\}_1, \{1,2,4\}_2, \{-3,5\}_3, \{4,-5\}_4, \{5,-6\}_5, \{6,7,9\}_6, \{6,-8,9\}_7\}$$

Definition 8 (**Variable Incidence Graph (VIG)**[2]). Given a SAT instance Γ [3] over the set of variables V, its *variable incidence graph* is a graph $G = (V, E)$ were $E = \{(u,v) \in V \times V \mid u, v \in C \text{ and } C \in \Gamma\}$ Its undirected weighted graph form consists on a tuple (V, w), with a weight function:

$$w(x,y) = \sum_{\substack{c \in \Gamma \\ x,y \in c}} \frac{1}{\binom{|c|}{2}}$$

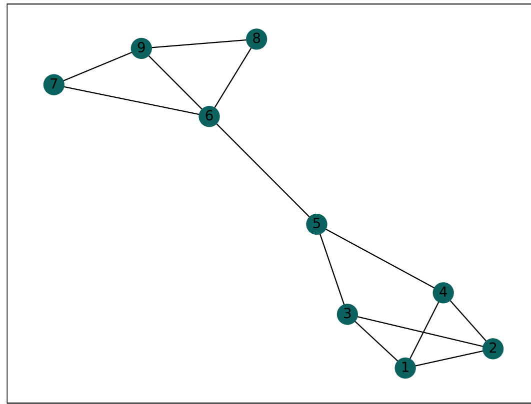


Figure 2.1: VIG form of Γ

[2] *Variable Interaction Graph* [31] term is also used for referring *VIG*.
[3] SAT instances

Definition 9 (**Clause-Variable Incidence Graph (CVIG)**[4]). Given a SAT instance Γ over the set of variables V, its *clause-variable incidence graph* is a bipartite graph G with sets V and $\{C \mid C \in \Gamma\}$, were each variable node is connected to the clause it is contained in. Its undirected weighted graph form consists on a tuple $(V, \{C \mid C \in \Gamma\}, w)$, with a weight function:

$$w(v, C) = \begin{cases} 1/|C| & \text{if } x \in C \\ 0 & \text{otherwise} \end{cases}$$

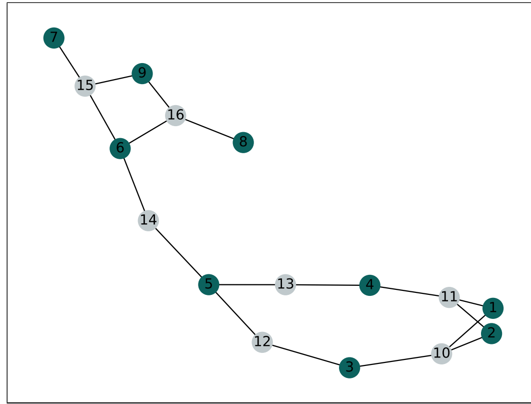


Figure 2.2: CVIG form of Γ

Definition 10 (**Clause Incidence Graph (CIG)**). Given a SAT instance Γ over the set of variables V, its *clause incidence graph* is an undirected graph, where there is a vertex for each $C \in \Gamma$, and an edge between two clauses that share a negated literal.

Definition 11 (**Resolution Graph** [31]). In the resolution graph $G_R = (S, E)$ an (undirected) edge is drawn between two clauses C_1 and C_2 if and only if there is a variable $x \in X$ such that $x \in C_1$ and $\bar{x} \in C_2$. Clauses C_1 and C_2 which are adjacent in G_R can thus be resolved. (Note that we also allow tautological resolvents here.)

[4] *Factor Graph* [31] term is also used for referring *CVIG*.

2.2 Community Structure

Modularity is a measure proposed by *Newman and Girvan* [21] for studying the so called community structure of a network. Community structure is presented in networks that can be easily partitioned into communities, such that most edges connect nodes of the same community. Modularity is defined as the number of edges falling within groups minus the expected number in a equivalent network with edges placed at random. This measure has been proposed by *Ansótegui, Giráldez-Crú and Levy* [5] for studying the underlying structure of industrial SAT instances. Modularity is obtained from the graph representation of a SAT instance, with the expression below [10]. The *VIG* form of a graph is commonly used for this task. Adj_{vw} value is 1 if vertex v and w are connected, and 0 otherwie. c_v denotes the community of a vertex v. $\delta(i,j)$ value is 1 if $i = j$, 0 otherwise. m is the number of edges in the graph.

$$Q = \frac{1}{2m} \sum_{v,w \in V} [Adj_{vw} - \frac{k_v k_w}{2m}] \delta(c_v, c_w)$$

In our context, we will compute modularity in the way below[2]. G is a graph, and C a partition of communities :

$$Q(G,C) = \sum_{C_i \in C} \frac{\sum_{x,y \in C_i} w(x,y)}{\sum_{x,y \in V} w(x,y)} - \left(\frac{\sum_{x \in C_i} deg(x)}{\sum_{x \in V} deg(x)} \right)^2$$

Whenever we talk about the modularity of a graph, we refer to the maximum obtainable modularity value for some partition. The decision problem version of optimising the modularity of a partition is $\mathcal{NP}$-complete [9]. There are several approximation algorithms, some are *greedy algorithms*, some are based on *label propagation* and some based on *graph folding*, like the *Louvain method*[8], which runs at $\mathcal{O}(n \log n)$

Most industrial SAT instances have a high modularity value Q that is not present in *random instances* [5]. It is shown that the number of communities and the modularity value of the graph of a real-world SAT instance is better for predicting the running time of a *CDCL* solver than the *number of variables and clauses* feature [22].

Prepossessing instances has no major impact on the structure of *industrial instances*. For random instances, modularity is only significant for very low clause-variable ratio; and there is no remarkable change observed in phase transition region. *Clause learning* procedures reduces of the modularity of instances at running time.

2.3 Scale-Free Property

Whenever we talk about *scale-free property*, we refer to the *scale free distribution*, or the *power law* distribution. It's density function is the following, where c is a constant, and α our statistical feature of interest:

$$p(x) \sim cx^{\alpha}$$

It's been observed that many large networks modelling real live constructions such as the World Wide Web present the scale-free property [6]. A *scale free network* is a network with *scale-free* degree distribution, that is, nodes' degree values are distributed following a *power-law* distribution. Scale free networks present the so called *small world property*, where nodes are *highly clustered* and *path length between them is small* [34]. The *path lenght* is the number of edges in the shortest path between two nodes. *The characteristic path length* L, is the average path length between all pairs of nodes. *The local clustering coefficient* C_u, is a measure of how close is the subgraph formed by the neighbourhood N_u of a node u from a clique. A clique containing those nodes has at most $k_u(k_u - 1)/2$ edges. *Clustering coefficient* C is the average clustering over all nodes in the graph. For an undirected graph, the *local clustering coefficient* value is defined as:

$$C_u = \frac{2|E|}{k_i(k_i - 1)}$$

The α value can be computed via *linear regression*, or the *maximum likelihood method.*

It's been noticed, that *industrial SAT instances* do present a *scale-free* structure. α value ranges in $[2, 3]$ in most of the cases [3]. Also, *clause learning* based methods seems to not modify dramatically this structure. This fact allows us to better understand the internal structure of these instances.

2.4 Self-Similar Structure

Self similarity is, to put in plainly, the property of an object to be exactly or approximately similar to a part of itself. The idea of self similarity arised from the empirical observation that geographical curves (the ones we find out in coastlines for example), present shapes that follow this principle [19]. This idea can also be applied to graphs. Let's introduce some concepts:

Definition 12 (*Box*)**.** Given a graph G, a box is subset of nodes such that the distance between any pair of nodes is smaller than a fixed value l.

Definition 13 ($N(l)$)**.** The minimum number of boxes of size l required to cover a given graph.

Now we can define *self-similarity* on a graph:

Definition 14 (*Self-Similarity Graph Property*)**.** It is said about a graph G, when for its $N(l)$ function, its value decreases polynomially for some value d in the following way: $N(l) \sim l^{-d}$

Chapter 3

The Tool

"You are an engineer"

Brian Eno and Peter Schmidt, *Oblique Strategies*

In this chapter, we are going to cover what our tools is, and what is not; and what can it do. We are going to talk about the workflow followed, and how it evolved. And at the end, we are going to discuss the future plans for the project.

3.1 Tool Description

The tool's name is *sia*, acronym for *SAT Instance Analysis* [1]. It is presented as a **Python Library**. It is meant to be used in **Windows and Linux** Python environments. Contains a **C/C++ Extension Module** for providing fast computations of some statistical features. C/C++ Extension Module needs to be build and installed. The necessary building tools, instructions and recommendations are listed within the *README* file. IO operations, formula manipulation, and other functionalities are built on plain Python.

[1] Sia is also a woman name, and the name of a pop artist `https://es.wikipedia.org/wiki/Sia`

3.2 Motivation

The idea of the tool that we are presenting arises not only from the personal necessity of computing some of the mentioned statistics, as from the necessity of manipulating *CNF formulas* in an easy and handy way.

Our computer language of choice for this task is going to be *Python* [33], not only because at dealing with statistical data it is one the most popular languages, but because of how easy is to share a tool built on it; specially when the project size is small/intermediate. Loading, and manipulating SAT instances feels intuitive in such a language. Also, there is a good ecosystem of *Python* libraries and tools for dealing with *satisfiability* and *SMT* problems, which makes this language a valid option.*CNFgen* [17] and *PySAT* [16] are some of the most interesting in this context. *CNFgen* is designed for creating SAT benchmark formulas, coming from *Proof Complexity* research area, such as matching or combinatorial problems; while *PySAT* provides a practical low level interface with *state-of-the-art* SAT solvers.

So, there is a tool in *Python* that creates **benchmark formulas**, and a tool that **solves formulas** with *state-of-the-art SAT* solvers. It seems natural that a new tool should appear that **analyses formulas**.

3.3 Objectives

The main objectives of the project were to provide a handy way for computing the statistical features mentioned in the chapter 2. The interest on this approach of dealing with *SAT instances* in a statistical/graph theoretic way was motivated by my recent interest on *computational complexity* and *graph theory*. Also, this [18] talk about the topic was decisive for my to finally choose it. The aspect of manipulating *CNF* formulas was something decided to cover at the programming state of the project.

3.3.1 Main Objectives

The main objectives were to understand and implement the computation of the already *infamous* statistical features:

- *Modularity*
- *Fractal dimension*
- *Scale-free structure*

Under the following conditions:

- The tool needs to be *easy to install*
- The tool needs to be *easy to use*
- The tool needs to be *easy to modify*

These objectives were fulfilled. But if we talk about modifying the *C/C++ Extension Module* things change; because this is, in fact, **not trivial to modify and adapt**. It requires, of course, some C/C++ knowledge, which is *usually* far from what you would expect from the average Python programmer. The guided building and installation process is intended to work for a Linux user using the **GNU g++** compiler, and a Windows user using the **mingw32** compiler. Using the **clang** compiler alternatively in *macOS*, or the **Microsoft Visual C++ (MSVC)** in Windows (which is commonly suggested for building Python C/C++ Extension Modules there) should work **fine**.

3.3.2 Secondary Objectives

The secondary objective, was related to building utilities for *CNF formula manipulation*.

Although, this task was loosing more and more sense as I was getting more in depth in the world of SAT related software. Already mentioned software like *CNFgen*[17] and PySAT [16] have good capabilities for *CNF formula* manipulations. Also, it already exists a package in Python called *cnftools*[2] and another one by the same name written in *C++*, with the same main author of *CNFgen* [17][3] that provide *CNF formula* manipulation. And if we talk about general logic formula manipulation and parsing, the list can be just endless.[4]

What I didn't found was a library that provided just common SAT solving step functions for transforming formulas, such as *variable propagation*, or the use of heuristics. Such functionalities can be good for understanding some of the algorithms behind combinatorial search, and SAT solving. This has been the main objective from the set of secondary objectives.

Some other discussed idea with my project advisor was the idea of visualising the execution trace of various *SAT solving algorithms* like *DPLL*, but this was discarded because:

- Building an efficient graphical application in Python is a hard task, even with the appropriate base tools and libraries. Is something you shouldn't make in Python.
- It already exists a tool like *DPVIS*[32], that does the same in *Java*, a tool like *3dVis*, from the same author, that does the same in *C++*[5]; and another Python tool called *iSAT*[25] that interfaces with SAT solvers and provides graph construction and visualisation and some graph statistical feature computation over space search graphs at each steps.

Anyway, a lot of time an effort was invested in this idea. The possibility of using the Python project *vispy*[6] made this idea somehow viable at the beginning. But this illusion soon faded out. Fair enough, it was used for rendering some of the images shown in this project.

[2] *cnftools*, Python: `https://github.com/easyas314159/cnftools`

[3] *cnftools*, C++: `https://github.com/MassimoLauria/cnftools`

[4] GitHub proposes 119 results for the search "logic manipulation" and 469 for "logic parsing" in July 2021.

[5] DPVIS is from 2005, and 3dVis from 2006.

[6] The *vispy* project: `https://vispy.org/`

3.4 Research and Development Workflow

3.4.1 Research

The research and reading process has been managed using a diary system of *Markdown* plain notes. Managed by the *Obsidian*[7] software. With this, system you can build any kind of workflow you are able to design in a very minimalist and easy to maintain way.

In my case, the *Zettelkasten*[8] method was used for note taking. Obsidian allows you to create notes that reference other notes, and build a network of knowledge.

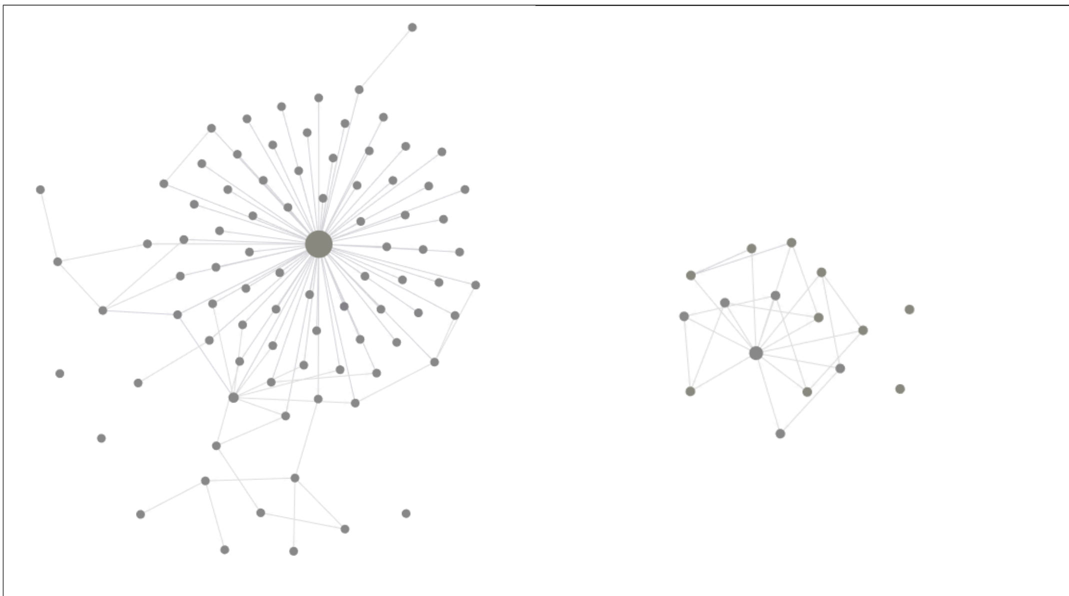


Figure 3.1: Obsidian graphview of the notes containing the tags ”sat” and ”sia”

Each node is an unique note. Each node contains a code at the beginning of its title of the form yyyymmddhhmm, so time can be easily tracked. Notes can reference to other notes, and tags for making them easy to classify. Tags were used for displaying the figure 3.1

[7] `https://obsidian.md/`

[8] `https://en.wikipedia.org/wiki/Zettelkasten`

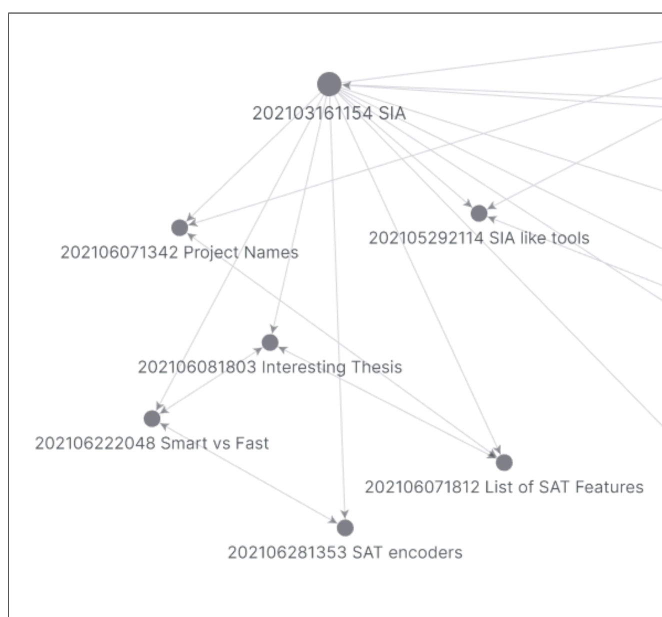


Figure 3.2: Example of some notes

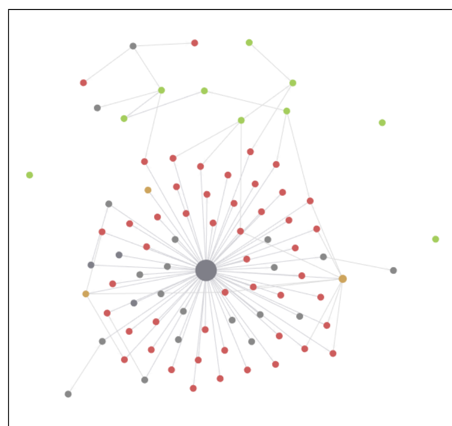


Figure 3.3: Obsidian graphview containing the intersection of sat and sia notes

The core node was serving as an index. The red dots are *papers*, the green ones are intersecting *sia* related notes, and the yellow ones are online/recorded talks. The grey ones belongs to notes on some other SAT tools, web pages of interest, and references to some researches of the SAT community. Each notes contains reviews of each paper, talk or concept made during the research, in order to make information easy to access in the future. Some of this papers, and ideas, are not mentioned in this report, because they correspond to preliminary work ideas that haven't been developed.

This system was adopted at the beginning to keep track of the agenda, and all the meetings with my project advisor. *Obsidian* has an addon that allows you to create notes representing days, and visualise each day in a calendar view. This was very handy at the beginning.

It was also used for taking notes of what have I done during specific amounts of times. Here is an example of how a rendered *Markdown* file of day looks like with a *Gantt* diagram:

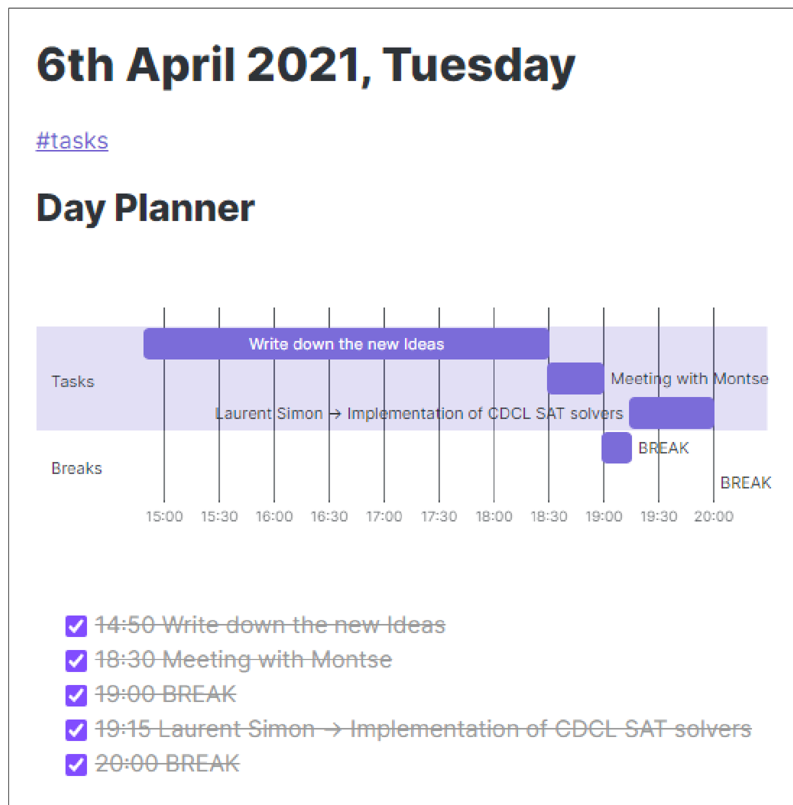


Figure 3.4: Obsidian day-planner task list and Gantt diagram

I really liked this system at the beginning, but when I felt the need of being more accurate with the time I spent, and to know how well invested this was, that is to say, how efficient I was with my time, I ended up switching to other time measurement system.

That system is going to be introduced in the following section.

3.4.2 Code Development

Here are going to be listed some key points of how code was develop:

- The main developing platform is been ***Windows 10***, along with the ***Windows Subsystem for Linux 2 (WSL2)***. This has allowed me to test and build the tool in Windows and Linux simultaneously with less effort.

- The Python interpreter used in Windows was the one provided in ***miniconda*** distribution, and the ***default Python interpreter*** provided for Ubuntu in Linux. The Python version used and tested were the 3.8.5, and the 3.8.10 respectively. 3.8 version should be just appropriate.

- The compilers I used and tested where the ***mingw32*** and the **GNU g++** for building the C Extension Package. Now that I have obtained some experience on this task, **I wouldn't recommend** to develop this kind of software with ***mingw32*** in Windows. It can bring you some headaches.

- No *Integrated Development Environments (IDE)* were used, as the most observant one will already inferred considering the last point. The ***Sublime Text 3*** and ***Vim*** text editors were used, in Windows and Linux respectively. The unpleasant default Windows Terminal, and the new upgraded one were used for launching the application.

- ***Make*** was used as the build system for Windows (the implementation provided by **mingw32**) and Linux. A single *Makefile* manages the differences between compiling for Windows, and Linux.

- For building C Extension Package, the ***setuptools*** library was preferred; but ***distutils*** is also supported.

- For setting up the Windows and Linux environments, I automated some tasks using Windows *bat* files, and Linux *shell* scripts. This files were called *sia.bat* and *setup.sh* respectively.

- The code was linted, formatted and tested using ***pylint***, ***autopep8*** and ***pytest***. The use of these tools was automated with the *Makefile*.

- In the implementation stage, ***Kanban*** methodology was used. Small tasks were listed, tagged, and time-measu red. For making this process easier, I used a service called *Kanbanflow*[9], which allows you to create kanban boards, and measure the time spent on each task.

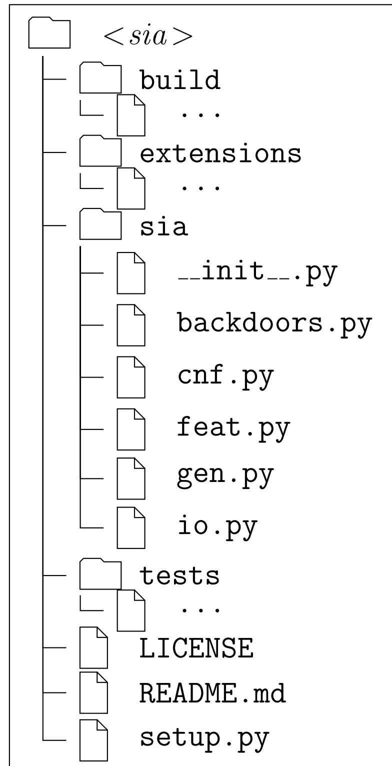
- *Kanbanflow's* time measurement system is the ***Pomodoro*** technique, a system were you work for a fixed amount of time on a task for 25 minutes, with no distraction, and then you rest for 5 minutes. 25 and 5 are the recommended *Pomodoro* times, but they can be modified. In reality, or at least in my case, you tend to make the pomodoros (that's how each individual working period is named) last longer when you feel comfortable/inspired. *Kanbanflow* allows you to track the extra time spent easily too. My longest pomodoro, to date, has lasted 2h and 15m, and it was related to obtaining and inserting images in this exact report text.

- The free version of *Kanbanflown* doesn't allow you to keep the log of the time spent for more than 14 days back in time. So I wrote a Python script called *kanban2csv.py* for parsing a raw copy of the data displayed on the web into a *csv* file.

[9] `https://kanbanflow.com/`

3.5 Project Outline

The project is organised as it follows, using a common *Python project* code structure.



3.6 Functionalities

The package is currently composed of 5 modules:

- backdoors.py
- cnf.py
- feat.py
- gen.py
- io.py

3.6.1 backdoors

This was part of the secondary objectives. The secondary objective was implementing functions for computing *CNF formula* manipulations. Some of the utilities needed in this context, where the ones for checking if a given formula belonged to a specific ***syntactic family***. These utilities were going to be part of *cnf.py*. But, after reading about *backdoors* [35], I decided to create a separate module, containing these. Detecting whether if a formula contains a backdoor, lies in what someone would expect as a topic of *"analysis"*, so looking to the future this looked like a good idea. Recognising polynomially solvable *syntactic classes* is part of the backdoor search process.

Another possibility was to create a third module containing *syntactic classes* related utilities, but at the moment, it doesn't feel appropriate.

Current contents:

- `class Synclass`: Class static methods for recognising *syntactic classes*:
  - `def is_k_sat(clauses, k)`
  - `def is_horn(clauses)`
  - `def is_dual_horn(clauses)`
- `def all_subsets(n, size=None)`: Returns a generator providing all possible variable assignments of n variables and length $size$.

3.6.2 cnf.py

This was part of the secondary objectives. This is the module intended for formula manipulation. It contains:

- `def empty_clauses(clauses):` Check if there exist an empty clause.
- `def unit_clauses(clauses):` Check if there exists an unit clauses.
- `def propagate_literal(clauses, literal)` Propagates a given literal.
- `def propagate(clauses, assumptions):` Propagates a given list of assumptions.
- `def unit_propagation(clauses):` Propagates unit literals of a given clause list.
- `def reduce_clauses(clauses):` Applies unit propagation until it isn't possible.
- `def pure_literal(n, clauses):` Returns an array of size number of vars n, indicating variable appearance, if positive pure literal, if negative pure literal, or none of the previous options.
- `def get_pure_literal(n, clauses):` Returns an assumption list from a given pure literal check list.
- `def to_3_sat(n, clauses):` Converts a CNF instance into a 3-CNF instance.

3.6.3 feat.py

Part of the main objectives. It computes the statistics mentioned in chapter 2. It serves as an interface for calling the *C Extension* module **featsat**; which is an interface for C/C++ lower level code, were heavy computations are performed. Here is were the implementation of the procedures described in chapter 2 [3][5][1] are contained:

- `def modularity(file_name, mode='vig'):` Computes de modularity of a CNF formula from a given file. It has VIG and CVIG mode.
- `def self_similar(file_name, mode='vig'):` Computes de fractal dimension of a CNF formula from a given file. It has VIG and CVIG mode.
- `def scale_free(file_name, mode='var'):` Computes de scale free exponent of a CNF formula from a given file. It has var and clause mode.

3.6.4 gen.py

This module was created before I knew about the existence of *CNFgen*[17]. It contains some common *random formula families* generation functions. It was decided to be included in the package because of practicality:

- `def random_formula_check(k, n, m, seed):` Check restrictions on random formulas.
- `def random_k_cnf(k, n, m, seed=None):` Random k-CNF formulas with no repeating literals per clause.
- `def random_horn(k, n, m, seed=None):` Random Horn formulas with clauses of size k.
- `def pigeon_hole(n):` Pigeonhole Principle[10] encoding for n pigeons and n-1 holes.

[10] https://en.wikipedia.org/wiki/Pigeonhole_principle

3.6.5 io.py

This was the first module to be created for obvious reasons. It contains functions for reading and loading *CNF DIMACS* files, which is the standard file format for encoding *CNF* formulas:

- `def parse_dimacs(file: Iterable[str]):` Parses iterable object of strings describing a DIMACS cnf file.
- `def from_file(in_file: Path):` Parse dimacs file.
- `def get_header(in_file: Path):` Parse dimacs file header.
- `def read_tar(file_path: Path):` Reads a tar file containing dimacs files.

3.7 Future Work

The main objectives of this project were fulfilled, and the phase of developing the secondary ones is been started. But it isn't enough. During development stage, Python project building skills and knowledge about how to write good Python code is been acquired. And now, *open sourcing* and sharing the project in the current state seems awkward (Anyway, the code is been uploaded to **GitHub** in order to share it with the tribunal) [11]. The reasons for not sharing it yet are:

- The functionalities covered by the package seems to be thematically displaced, even if all belong to the context of satisfiability. Also, only the ones from the main objectives cover a necessity in an efficient way; even if it is a *niche* necessity. The solution for this, is to prune the variety of functionalities, and **do better** the ones that remain. Knowing about the *UNIX Design Philosophy* [26] could have helped in the beginning of the project, to build a better tool.

- Sharing a project in *the good* way implies making an *exhaustive* code documentation; and even if this project contains straight forward implementations of already well know algorithms, currently the use instructions are contained only in the *README* file. By the way, this is something usual in a lot of *state-of-the-art* SAT solvers source code repositories[12]. But the fact that this is usually done in this way, doesn't mean it is right.

- Lots of unit tests are usually used in order to guarantee that code is working properly on each release. The initial tests written in the project were focused on testing the *IO* functions. Open sourcing the project would imply to build better suites of tests.

This reasons arise from the *software engineering* mindset, and the truths is, that this project is more related to researching purposes. Anyway, we have to keep them into consideration.

Talking about design and performance, the possibility of building the whole package in C/C++ is been around my head during the last project month. Reusing efficient C code from well-known SAT solvers in order to implement more efficient data structures, and create an interface in Python not only seems like an interesting challenge, but as an opportunity to create something that eventually someone might need (At least, this is something I would have liked to have 2 years ago).

[11] `https://github.com/blcksy/sia`
[12] *Kissat* is an example of this: `https://github.com/arminbiere/kissat`

Considering all said, these are my future plans for this project:

3.7.1 Do One Thing and Do It Well

The project is going to be redesigned. All the tools that are already available in the SAT community are going to be taken into consideration, and new opportunities are going to be identified. If the final decision consist on creating an efficient Python interface for an already existing tools, this must be done in a way that becomes truly useful.

3.7.2 Pure C

The functionalities covered in the secondary objectives set are going to be separated from the current project, and they are going to be reintroduced as another Python package. And this time, it is going to be written as a complete *C Extension Module*, and properly extended.

3.7.3 Share

The main project is going to be redesigned and presented as a feature extracting module interface for Python, and published on GitHub. It will contain all the necessary instructions for its correct installation, and use.

Chapter 4

Conclusions

The main objectives were accomplished. Lots of secondary objectives and ideas were studied, but not developed. No strict completion dates were imposed besides the *University Project Delivery Dates*, and there was plenty of time for reading and discovering about SAT solving, and SAT related software. This was also why the average work load began to increase while we were reaching the end, and why the necessity to track the time invested was emerging. It is convenient to start since the very beginning tracking time, in order to know what takes you more time to accomplished. Adopting a more rigorous way to track time, helped in the second half of the project. As a note, approximately 45 hours were used in the creation of this document, including the process of learning how to properly use LaTeX, formatting layouts, obtaining images, solving errors, etc. besides the actual process of writing; and approximately 34 hours only for learning how to build and test properly a *C Extension Module*. These two tasks were thoroughly measured because of the suspicion that they were going to take forever. Anyway, when counting the actual time spend I felt surprised. This was useful too, to know which were my working *"bottlenecks"*. It turns out that the most time consuming tasks, but with less impact int the work, have been the ones related to taking notes and reading papers, that finally haven't been mentioned. This fact is been specially frustrating. Nevertheless, all that process turns into experience acquired.

Realising at every step I made that there was already a software providing the ideas I had felt frustrating. But it helped me to get more in depth into the subject.

Since the project was proposed by me, I felt really helpless in the sense that I had no colleagues to share and talk about the project, and my advisor wasn't specialised on the topic. Also, because of personal reasons, I've been working from home, and this increased the feeling of being lost. Nevertheless, my advisor was really supportive with me, and guided me in the best way possible, considering the context. The meetings were held weekly by video conference on Tuesdays at 18:30, unless specified via email otherwise.

However, there are some things I've learnt during the completion of the project, which are truly positive:

- Time management in long-term projects.
- Scientific paper reading skills.
- A better understanding of Python and C/C++.

Appendix A

kanban2csv.py

```
'''
Tool for parsing time log from https://kanbanflow.com/
into a csv file

Kanban flow format:
    # <Date, number of pomodoros>
    <task title>
    <pomodoro status>
    <time spent>
    <time range>

Example:

    # Monday, 2 August 1h 16m 3 Pomodoros
    Create a package
    Successful Pomodoro
    25m
    14:26 - 14:51


TODO:

    - Improve how the year is introduced
'''

import csv
from datetime import datetime
```

```python
from pathlib import Path

KANBAN_PATH = 'sia/KANBAN.md'
LOG_CSV_PATH = 'sia/log.csv'


def get_log(file_in):
    '''
    Parses kanban flow data into a list of lists of lists of str

    Format:
        [
        [['<date:day>'], ['<task>', '<status>', '<time>', 'hours']],
        ]
        ...
    '''

    path = Path(file_in)
    log_list = []
    count = 0

    with open(path, 'r') as file:

        for line in file:

            if line.startswith('#'):
                log_list.append([[line.strip()[2:]]])
                continue

            if not line.strip():
                continue

            if count == 0:
                log_list[-1].append([])

            log_list[-1][-1].append(line.strip())
            count += 1

            if count == 4:
                count = 0

    # for l in log_list:
    #     for b in l: print(b)

    return log_list
```

```
def create_csv(log):
    '''
    Creates csv file with the given rows and '\t' delimeter

    Example:

        day         hour            time tasks              status
        2021/08/02  14:26 - 14:51   25m  Create a package   Successful Pomodoro
    '''

    new_log = []

    day = ''
    hours = ''
    time = ''
    task = ''
    status = ''

    for day_list in log:

        if len(day_list) == 1:
            # Empty day list
            pass

        for task_list in day_list:

            if len(task_list) == 1:
                if task_list[0] == 'missing':
                    continue

                day = task_list[0]
                day = ' '.join(day.split()[0:3])
                date = datetime.strptime(day, '%A, %d %B')
                # TODO: Improve how the year is introduced
                date = date.replace(year=datetime.now().year)
                continue

            hours = task_list[3]
            time = task_list[2]
            task = task_list[0]
            status = task_list[1]

            new_task = [date.strftime('%Y/%m/%d'), hours, time, task, status]
```

```
            new_log.append(new_task)

    # Save log entries in LOG_CSV_PATH
    with open(LOG_CSV_PATH, 'w', newline='') as file:
        writer = csv.writer(file, delimiter='\t')
        writer.writerow(['day', 'hour', 'time', 'tasks', 'status'])
        writer.writerows(new_log)


def count_time(log):
    '''
    Returns str containing total pomodoros and time
    '''

    pomodoros = 0
    hours = 0
    minutes = 0
    seconds = 0

    for day_list in log:
        if len(day_list) == 1:
            continue

        for task_list in day_list:
            if len(task_list) == 1:
                # Print each day info
                print(task_list)
                if task_list[0] == 'missing':
                    continue
                pomodoros += int(task_list[0].split()[-2])
                continue

            time = task_list[2].split()

            for unit in time:
                if unit[-1] == 'h':
                    hours += int(unit.replace('h', ''))
                    continue
                if unit[-1] == 'm':
                    minutes += int(unit.replace('m', ''))
                    continue
                if unit[-1] == 's':
                    seconds += int(unit.replace('s', ''))
                    continue
```

```
            '''
            if task_list[2][-1] == 'm':
                minutes += int(task_list[2].replace('m', ''))
                continue
            if task_list[2][-1] == 's':
                seconds += int(task_list[2].replace('s', ''))
                continue
            '''

    hours = hours + minutes//60
    minutes = (minutes % 60) + seconds//60
    seconds = seconds % 60
    time = f'pomodoros: {pomodoros}, hours: {hours}, '\
           f'minutes: {minutes}, seconds: {seconds}'
    return time


if __name__ == '__main__':

    log_entries = get_log(KANBAN_PATH)
    total_time = count_time(log_entries)
    print(total_time)
    create_csv(log_entries)
```

Appendix B

sia.bat and setup.sh

```
:: Script for starting sia conda env
@title sia
@cd Documents\Sia\sia
:: ENV variables
@set FITNESS_TOOL=C:\Programas\miniconda3\Scripts\pylint.exe
@set FORMATTING_TOOL=C:\Programas\miniconda3\Scripts\autopep8.exe
@set TEST_TOOL=C:\Programas\miniconda3\Scripts\pytest.exe
:: Commandline aliases
@doskey ll=ls -la
@doskey pt=%TEST_TOOL%
@doskey t=tree
@doskey tf=tree /F
@conda activate sia
```

```
#!/bin/bash
source Sia/sia-venv/bin/activate
export FITNESS_TOOL=pylint
export FORMATTING_TOOL=autopep8
export TEST_TOOL=pytest
alias t='tree /mnt/c/Users/o/Documents/Sia/sia/ -I "sia-venv|tools"'
cd /mnt/c/Users/{User}/Documents/Sia/sia
```

Bibliography

[1] Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., and Levy, J. The fractal dimension of sat formulas. In *International Joint Conference on Automated Reasoning* (2014), Springer, pp. 107–121.

[2] Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., and Levy, J. Structure features for sat instances classification. *Journal of Applied Logic 23* (2017), 27–39.

[3] Ansótegui, C., Bonet, M. L., and Levy, J. On the structure of industrial sat instances. In *International Conference on Principles and Practice of Constraint Programming* (2009), Springer, pp. 127–141.

[4] Ansótegui, C., Bonet, M. L., Levy, J., and Manya, F. Measuring the hardness of sat instances. In *AAAI* (2008), vol. 8, pp. 222–228.

[5] Ansótegui, C., Giráldez-Cru, J., and Levy, J. The community structure of sat formulas. In *International Conference on Theory and Applications of Satisfiability Testing* (2012), Springer, pp. 410–423.

[6] Barabási, A.-L., and Albert, R. Emergence of scaling in random networks. *science 286*, 5439 (1999), 509–512.

[7] Biere, A., and Sinz, C. Decomposing sat problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation 2*, 1-4 (2006), 201–208.

[8] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment 2008*, 10 (2008), P10008.

[9] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. On modularity-np-completeness and beyond. *ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), Tech. Rep 19* (2006), 2006.

[10] Clauset, A., Newman, M. E., and Moore, C. Finding community structure in very large networks. *Physical review E 70*, 6 (2004), 066111.

[11] Cook, S. A., and Mitchell, D. G. Finding hard instances of the satisfiability problem: A survey. *Satisfiability Problem: Theory and Applications 35* (1996), 1–17.

[12] Cook, S. A., and Reckhow, R. A. The relative efficiency of propositional proof systems. *The journal of symbolic logic 44*, 1 (1979), 36–50.

[13] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[14] Deleau, H., Jaillet, C., and Krajecki, M. Gpu4sat: solving the sat problem on gpu. In *PARA 2008 9th International Workshop on State–of–the–Art in Scientific and Parallel Computing, Trondheim, Norway* (2008), Citeseer.

[15] Dowling, W. F., and Gallier, J. H. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming 1*, 3 (1984), 267–284.

[16] Ignatiev, A., Morgado, A., and Marques-Silva, J. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT* (2018), pp. 428–437.

[17] Lauria, M., Elffers, J., Nordström, J., and Vinyals, M. Cnfgen: A generator of crafted benchmarks. In *International Conference on Theory and Applications of Satisfiability Testing* (2017), Springer, pp. 464–473.

[18] Levy, J. On the formal characterization of industrial sat instances, February 2021. `https://www.youtube.com/watch?v=KQvCd4oqJNg`.

[19] Mandelbrot, B. How long is the coast of britain? statistical self-similarity and fractional dimension. *science 156*, 3775 (1967), 636–638.

[20] Mitchell, D., Selman, B., and Leveque, H. A new method for solving hard satisfiability problems. In *Proceedings of the tenth national conference on artificial intelligence (AAAI-92)* (1992), pp. 440–446.

[21] Newman, M. E., and Girvan, M. Finding and evaluating community structure in networks. *Physical review E 69*, 2 (2004), 026113.

[22] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., and Simon, L. Impact of community structure on sat solver performance. In *International Conference on Theory and Applications of Satisfiability Testing* (2014), Springer, pp. 252–268.

[23] Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., and Hoos, H. Satzilla: An algorithm portfolio for sat. *Solver description, SAT competition 2004* (2004).

[24] Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A., and Shoham, Y. Understanding random sat: Beyond the clauses-to-variables ratio. In *International Conference on Principles and Practice of Constraint Programming* (2004), Springer, pp. 438–452.

[25] Orbe, E., Areces, C., and Infante-López, G. isat: structure visualization for sat problems. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2012), Springer, pp. 335–342.

[26] Pike, R., and Kernighan, B. Program design in the unix environment. *AT&T Bell Laboratories Technical Journal 63*, 8 (1984), 1595–1605.

[27] Sato, T., and Kojima, R. Matsat: a matrix-based differentiable sat solver. *arXiv preprint arXiv:2108.06481* (2021).

[28] Schaefer, T. J. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing* (1978), pp. 216–226.

[29] Selman, B., Mitchell, D. G., and Levesque, H. J. Generating hard satisfiability problems. *Artificial intelligence 81*, 1-2 (1996), 17–29.

[30] Silva, J. P. M., and Sakallah, K. A. Grasp—a new search algorithm for satisfiability. In *The Best of ICCAD*. Springer, 2003, pp. 73–89.

[31] Sinz, C. Visualizing sat instances and runs of the dpll algorithm. *Journal of Automated Reasoning 39*, 2 (2007), 219–243.

[32] Sinz, C., and Dieringer, E.-M. Dpvis–a tool to visualize the structure of sat instances. In *International Conference on Theory and Applications of Satisfiability Testing* (2005), Springer, pp. 257–268.

[33] van Rossum, G. Python tutorial. Tech. Rep. CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[34] Walsh, T., et al. Search in a small world. In *Ijcai* (1999), vol. 99, Citeseer, pp. 1172–1177.

[35] Williams, R., Gomes, C. P., and Selman, B. Backdoors to typical case complexity. In *IJCAI* (2003), vol. 3, pp. 1173–1178.