Empirical Analysis of the RC2 MaxSAT Algorithm

Stepan Kochemazov, Victor Kondratiev, Irina Gribanova
ISDCT SB RAS, Irkutsk, Russia
Emails: veinamond@gmail.com, vikseko@gmail.com, the42dimension@gmail.com

***Abstract*—The Boolean satisfiability problem (SAT) and maximum satisfiability problem (MaxSAT) are among the most well-known combinatorial problems in today's computer science. The algorithms for their solving also go hand-in-hand, in that most MaxSAT solvers employ SAT solvers as the so-called oracles. In the present paper we perform a computational study of the RC2 algorithm, which is among the best state-of-the-art algorithms for MaxSAT solving. We view it from the SAT oracle viewpoint and consider how the SAT oracle's runtime is distributed among RC2 procedures and heuristics, and how this statistics differs depending on the SAT solver employed as an oracle. In addition to that we consider the two baseline MSE'18 configurations of RC2, analyze their performance and experiment with blending them together.**

***Keywords*—MaxSAT, SAT, rc2**

I. Introduction

Nowadays, the industrial design poses many challenges that require the use of highly efficient computational apparatus for solving combinatorial problems of large dimension. In particular, many problems can be expressed in a relatively simple language, involving the constraints over the variables that take integer values or even only the values from the set $\{0,1\}$. They arise in economics, planning, computer-aided design and many other areas. Among the most well known scientific approaches for solving such problems one can point out the ability to express them in form of instances of two classical problems from Computer Science: the Integer Linear Programming problem (ILP) [1] and the Boolean satisfiability problem (SAT) [2].

ILP consists in maximizing the value of a linear function defined over a set of integer variables while satisfying a number of linear inequalities. In SAT the problem is specified by an arbitrary Boolean formula, and the goal is to determine whether there exists an assignment of all variables of a formula on which it evaluates to $True$. While both problems are NP-complete, in practical reality, ILP solving algorithms fit well with large-scale optimization tasks planning, scheduling, logistics, etc. Meanwhile, SAT solving algorithms are usually employed to construct formal proofs or find counterexamples under certain assumptions, and find a lot of uses in hardware and software verification, certain areas of planning, etc.

What makes SAT unique is that the algorithms for its solving are often used as the so-called *oracles* in the algorithms designed for tackling combinatorial problems with the hardness beyond NP. The term "SAT oracle" is inspired by the concept of oracles widely used in computational complexity theory, where they represent some entities capable of solving certain (hard) problems. Two well-known examples of the problems, the algorithms for solving which employ SAT oracles, are Satisfiability Modulo Theories (SMT) [3] – the approach that blends the rich expressiveness of languages of mathematical theories different from propositional logic with the effectiveness of modern SAT solvers, and Maximum Satisfiability [4] – the optimization variant of SAT, that makes it possible to naturally consider instances related to optimization. After all, in day-to-day practice, it is usually more convenient to solve *optimization* instances: find the shortest path from A to B, establish the shipping schedule to minimize losses while maximizing profits, etc., instead of *decision* instances: prove or disprove that there is a path from A to B shorter than k, or find a schedule under which the profits are at least P while losses are at most L or prove that it does not exist.

The latter observation, together with the fast progress and wide availability of fast SAT solving algorithms (compared to the ILP area where commercial solvers are in a league of their own performance-wise) led to a large popularity of MaxSAT and the fast development of algorithms for its solving. In the present paper we study one of the algorithms for solving MaxSAT, called `RC2` [5]. It was proposed in 2018 and took the first place in both weighted and unweighted categories of the MaxSAT Evaluations held in 2018 and 2019[1].

In our study we look at `RC2` from the perspective of the employed SAT oracle in order to better understand the possible venues for improving the interconnection between the higher-order MaxSAT logic and the ground-level SAT oracle. For this purpose we analyze the runtime distributions of the main `RC2` procedures, especially the ones that employ SAT oracles.

Let us give a brief overview of the paper: in the next chapter we introduce the necessary notation regarding SAT and MaxSAT, provide the basic information about the architecture of modern SAT and MaxSAT solvers. In Section III we describe the `RC2` algorithm, the way it uses SAT solvers as oracles, and the related heuristics. After this we present the details about the experimental setting, describe our computational experiments and analyze the obtained data.

II. Preliminaries

A Boolean variable is a variable that takes the values from the set $\{False, True\}$ which in practice usually converts into $\{0,1\}$, where 0 represents $False$ and 1 represents $True$. A Boolean formula is comprised of Boolean variables connected

[1]The MaxSAT Evaluations are annual competitive events for MaxSAT solvers that promote and chronicle the progress in this area

using braces and the so-called *logical connectives*. The latter are the well-known logical operations that usually form a complete basis, e.g. $\{\wedge, \neg\}$, $\{\wedge, \vee, \neg\}$, etc. In both SAT solving and MaxSAT solving, Boolean formulas are considered in the Conjunctive Normal Form (CNF), so for simplicity let us only introduce this form. A formula in CNF is defined over the basis $\{\wedge, \vee, \neg\}$. CNF is a conjunction ($\wedge$) of clauses, where a *clause* is a disjunction ($\vee$) of literals, and a *literal* is either a Boolean variable or its negation ($\neg$). Literals x and $\neg x$ are called *complementary*, because for any value of x one of them will take the value 1. For convenience, the clauses in CNF are not allowed to contain complementary or duplicate literals. An *assignment* of variables is the set of values from $\{0, 1\}$, assigned to each variable. An assignment is called *satisfying* if it turns formula into 1 in accordance with the standard semantics of logical connectives. In the case of CNF it is especially easy to check that an assignment is satisfying: it is sufficient to ensure that every clause if satisfied. A formula is called *satisfiable* if it has at least one satisfying assignment, otherwise it is called *unsatisfiable*.

A. SAT and SAT Solvers

The Boolean satisfiability problem (SAT) consists in the following: for an arbitrary CNF to answer whether it is satisfiable or not (decision problem). The more practical search variant is to find a satisfying assignment if CNF is satisfiable.

The practical algorithms for solving SAT are called *SAT solvers*. They have a (relatively) long history and are considered one of the success stories in today's Computer Science. The SAT solving algorithms achieved a tremendous progress during the recent 30 years, going from barely being able to tackle formulas with hundreds of variables and thousands of clauses to effectively solving practical SAT instances with hundreds of thousands of variables and millions of clauses. This fact, together with the applications of SAT solvers in formal verification of circuits [6] and software [7] made them into an effective tool widely employed today in many other areas, such as bioinformatics [8] or cryptography [9].

The result of the many years of progress in SAT solving and the currently predominant algorithm for this purpose is called *Conflict-Driven Clause Learning* (CDCL). The CDCL SAT solvers enhance the depth-first search with various methods, such as *clause learning* and backjumping [10], variable selection heuristics [11], restarts, heuristics for manipulating the clauses of a formula mid-search, etc., see [2] for a comprehensive overview. An important feature of some of the state-of-the-art SAT solvers is the ability to work in the so-called *incremental* mode [12], which makes it possible to use a single SAT solver to tackle a sequence of SAT instances, in which each following instance is formed by extending the previous instance by adding to it new clauses and possible new variables. Thanks to this one can reuse the information accumulated by the solver throughout the process to improve its performance.

B. Maximum Satisfiability

The *Maximum satisfiability problem* (MaxSAT) is an optimization variant of SAT. For a Boolean formula in CNF the MaxSAT problem is to maximize the number of satisfied clauses. Due to a number of practical considerations, in practice MaxSAT is usually considered in a slightly different formulation called *Weighted Partial MaxSAT* (WPMS). The *Weighted* part refers to the fact that clauses are allowed to have weights. The *Partial* part instead is aimed at splitting formula in two: the part that must be satisfied (*hard* clauses) and the part that may be (partially) falsified (*soft* clauses). One can view hard clauses in the context of the weighted model as the clauses the individual weight of which exceeds the total weight of all soft clauses.

There are two main classes of MaxSAT solving algorithms: complete and incomplete. Complete algorithms make it possible to construct provably optimal solutions to the optimization problem at hand and it is the complete algorithms that employ SAT solvers. Incomplete algorithms, on the other hand, are designed in such a way as to quickly find a solution which is quite good, and then use the allotted time to improve it (if possible). One notable example of incomplete algorithms for MaxSAT which is also employed in the context of SAT solving are the Stochastic Local Search algorithms. Generally, they do not employ any inference rules like SAT solvers, instead, they explore the space of possible assignments of variables of a formula, but they do it with insane speed, checking up to billions of assignments per second. The recent advancements in MaxSAT solving actually focus mostly on incomplete algorithms and also on ways to efficiently combine fast incomplete methods for finding approximations of a solution with the ability of complete methods to find an optimum.

So, a MaxSAT instance is a weighted CNF: a CNF in which some clauses are hard, e.g. they must be satisfied, and the remaining clauses are soft and are assigned some weights. The goal is to find an assignment of variables that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses.

The state-of-the-art MaxSAT solvers rely on two techniques to employ SAT oracles: selector literals and cardinality constraints [13]. An individual *selector literal* is added to each soft clause. For example, if there is an original clause $a \vee b \vee c$, it is modified to $a \vee b \vee c \vee s_1$. By assigning $s_1 = 1$ the algorithm can specify that this clause is already satisfied and can be excluded from consideration (e.g. when invoked on a CNF with this clause, the SAT oracle will search for a solution that does not have to satisfy it). Vice versa, by fixing s_1 to 0 the SAT oracle will have to satisfy this clause. The cardinality constraints technique allows to represent in SAT form the [in]equalities of the kind $x_1 + x_2 + \ldots + x_n [\leq, \geq, =] K$.

It is easy to use the two described techniques to construct the following straightforward SAT-based algorithm for MaxSAT. First, introduce selector literals to all soft clauses in an input weighted CNF. Next, set the value of auxiliary variable k to 1. Encode the cardinality constraint over selector variables so

that it enforces that the sum of weights of all satisfied soft clauses is at least k. Then, invoke a SAT solver on the CNF constructed by appending the encoded cardinality constraint to CNF with selectors, if the answer is ”Satisfiable” then increment k, otherwise, k is the maximum weight that we needed to find.

It is clear, that if we exclude SAT oracle calls, then the algorithm is polynomial and works for at most S steps. Thus, the use of SAT oracle constitutes the major part of the complexity of this MaxSAT solving algorithm. Due to the fact that the size of cardinality encodings quickly grows with the increase of the number of variables involved, the presented approach is not as good as more sophisticated algorithms for MaxSAT. For an excellent tutorial on MaxSAT we refer to online resource [2].

III. RC2 ALGORITHM FOR MAXSAT

The `RC2` algorithm [5] is one of the state-of-the-art complete algorithms for MaxSAT. It is implemented as a part of the PySAT package [14]. Let us below briefly describe its main features and the ways it interacts with a SAT oracle.

The `RC2` abbreviation means *Relaxable Cardinality Constraints*. It is a *core-guided* MaxSAT solver, meaning that its functioning revolves around the so-called unsatisfiable cores. Such a core is formed by a subset of clauses that are responsible for a SAT instance being unsatisfiable. In particular, `RC2` employs the selector variables in a manner similar to that of the algorithm described above, but constructs cardinality constraints only for selector variables from unsatisfiable cores.

`RC2` heavily relies on the incremental capabilities of an employed `MiniSat`-like [15] SAT oracles: the extraction of an unsatisfiable core is performed via the *assumptions* mechanism (see [12]), the manipulation of cardinality constraints is implemented via the use of iterative cardinality encodings, such as incremental Totalizer encoding [16], [17], etc. Since we do not aim to explain `RC2` in detail, for particulars regarding the inner workings of the algorithm refer to [5]. As it was already mentioned, the goal of the present paper is to study how the `RC2` handles the SAT oracles. Let us consider it in detail in the following sections.

IV. ANALYSIS OF RC2 INTERACTION WITH SAT ORACLE

`RC2` performs SAT oracle calls in 4 procedures [3]:

- `compute_` – the main procedure that seeks to find the solution to the optimization problem
- `trim_core` – the procedure used to reduce the size of a recently extracted unsatisfiable core.
- `exhaust_core` – the procedure used to *exhaust* an unsatisfiable core by attempting to increase its cost
- `minimize_core` – the procedure used to invoke a SAT oracle with a tight conflict budget in order to reduce an extracted unsatisfiable core via a deletion-based algorithm.

[2]https://ecai20-maxsat-tutorial.github.io/

[3]https://github.com/pysathq/pysat/blob/master/examples/rc2.py

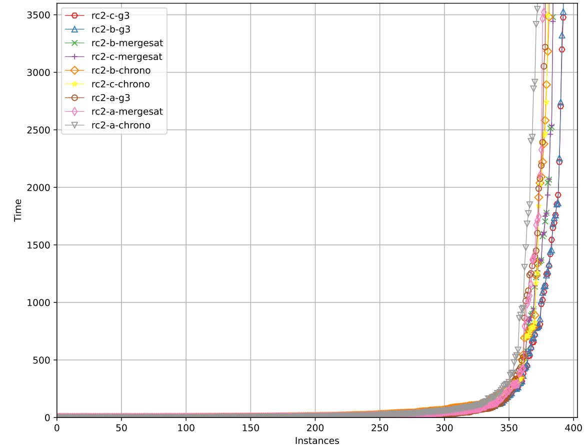

Fig. 1. Cactus plot for considered solvers on unweighted benchmarks

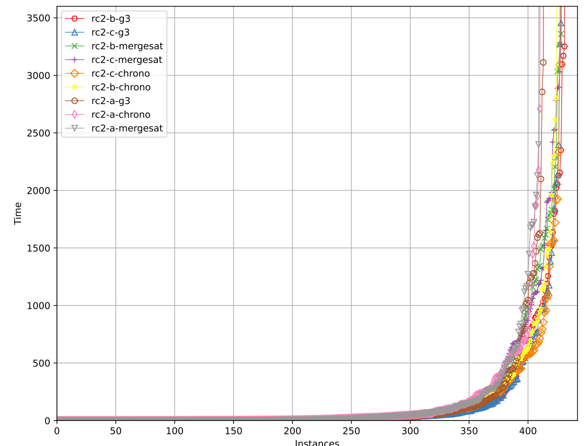

Fig. 2. Cactus plot for considered solvers on weighted benchmarks

Apart from the mentioned methods, one of the major heuristics in the algorithm is the detection of intrinsic AtMost1 constraints, implemented in the `adapt_am1` method.

Note, that there are two baseline versions of `RC2`: `RC2-A` and `RC2-B`, which differ in how they reduce the unsatisfiable cores. Usually, the `RC2-B` version performs slightly better than the other.

A. Experimental Setup

All experiments were run on a PC with AMD Ryzen 5900x CPU (12 cores) and 80 GB RAM, working under the Ubuntu 22.04 OS. In all cases 12 threads were running simultaneously. In the role of benchmarks we initially picked the suite of instances from the weighted and unweighted categories of

TABLE I
STATISTICS ON THE RUNTIME OF PROCEDURES IN RC2 WITH DIFFERENT SAT ORACLES ON **UNWEIGHTED** BENCHMARKS

	Parsing		Compute		Adapt AM1		Minimize		Exhaust		Trim	
	abs	rel	abs	rel	abs	rel	abs	rel	abs	rel	abs	rel
						`RC2-A-Glucose3`						
mean	2.412	0.308	42.815	0.268	1.636	0.145	0.000	0.000	45.771	0.238	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	124.644	1.000	3146.778	1.000	69.196	0.998	0.001	0.003	1994.739	0.991	0.002	0.004
median	0.111	0.139	0.099	0.107	0.009	0.016	0.000	0.000	0.083	0.099	0.000	0.000
						`RC2-B-Glucose3`						
mean	3.093	0.276	47.783	0.219	2.016	0.120	29.067	0.230	35.919	0.132	0.000	0.000
min	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	105.170	1.000	2732.359	0.999	70.783	0.998	3298.355	0.996	2967.639	0.990	0.004	0.003
median	0.122	0.104	0.102	0.048	0.009	0.011	0.165	0.082	0.052	0.030	0.000	0.000
						`RC2-C-Glucose3`						
mean	3.066	0.285	47.417	0.217	2.019	0.117	28.459	0.227	35.361	0.131	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	104.207	1.000	2696.451	0.999	71.387	0.998	3255.592	0.995	2852.446	0.989	0.004	0.004
median	0.123	0.108	0.096	0.046	0.009	0.009	0.163	0.081	0.048	0.029	0.000	0.000
						`RC2-A-Maplechrono`						
mean	1.204	0.222	38.834	0.313	1.583	0.113	0.000	0.000	56.555	0.328	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	21.962	1.000	2391.076	0.998	70.024	0.998	0.001	0.002	3175.744	0.996	0.001	0.003
median	0.117	0.033	0.746	0.128	0.008	0.002	0.000	0.000	0.481	0.174	0.000	0.000
						`RC2-B-Maplechrono`						
mean	1.453	0.202	39.783	0.273	1.807	0.102	22.033	0.158	39.587	0.247	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	64.954	1.000	2957.176	0.994	69.389	0.998	3261.089	0.989	2835.435	0.994	0.004	0.003
median	0.118	0.034	0.534	0.061	0.009	0.003	0.210	0.040	0.341	0.065	0.000	0.000
						`RC2-C-Maplechrono`						
mean	1.448	0.203	30.737	0.270	1.830	0.103	21.636	0.160	38.625	0.247	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	67.389	1.000	2283.725	0.993	69.466	0.998	3274.200	0.988	2685.708	0.994	0.004	0.003
median	0.117	0.035	0.475	0.061	0.009	0.003	0.223	0.042	0.304	0.060	0.000	0.000
						`RC2-A-Mergesat3`						
mean	2.316	0.273	37.163	0.307	1.650	0.126	0.000	0.000	45.214	0.266	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	131.545	1.000	3075.583	1.000	68.049	0.998	0.001	0.003	2886.466	0.996	0.002	0.004
median	0.122	0.092	0.209	0.138	0.009	0.007	0.000	0.000	0.159	0.130	0.000	0.000
						`RC2-B-Mergesat3`						
mean	1.975	0.237	48.584	0.248	2.068	0.112	23.784	0.221	29.989	0.163	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	81.939	1.000	2362.645	0.994	72.766	0.998	1445.034	0.995	1685.072	0.989	0.005	0.004
median	0.123	0.061	0.225	0.063	0.009	0.006	0.203	0.075	0.103	0.039	0.000	0.000
						`RC2-C-Mergesat3`						
mean	1.972	0.237	48.436	0.247	2.009	0.111	23.959	0.222	30.120	0.163	0.000	0.000
min	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	81.227	1.000	2378.219	0.995	71.547	0.998	1445.101	0.995	1729.190	0.989	0.005	0.003
median	0.122	0.066	0.236	0.062	0.010	0.007	0.195	0.074	0.097	0.040	0.000	0.000

MaxSAT Evaluation 2021 [4]. Then we ran `RC2-B` with Glucose 3 as SAT oracle (which is standard) on these instances with the time limit of 3600 seconds and collected only the benchmarks on which it terminated during this time (either by successfully solving them or due to some unexpected behavior).

B. Evaluating the Contribution of `RC2` Procedures to Runtime

Our main goal is to assess how much each of the procedures contributes to runtime. Of particular interest are the time required for parsing and for extracting intrinsic AtMost 1 constraints, since both procedures are actually implemented in Python, and Python is well known for not being as efficient as e.g. C or C++ when it comes to resource-intensive operations. Apart from that, it is interesting to see how the SAT oracle use, runtime-wise, is split between the main (`compute_`) procedure and the heuristics.

To study the considered details we launched `RC2` in three configurations: standard two: `RC2-A` and `RC2-B` and also `RC2-C` which enables heuristics employed in both configurations. We also decided to see whether the behavior differs depending on what solver is employed as a SAT oracle. Thus, in the role of the latter we considered three solvers supported by PySAT:

- Glucose 3 – the baseline `RC2` solver
- Maplechrono – the MapleLCMDistChronoBT solver that won in SAT Competition 2018
- Mergesat 3 – the MergeSAT solver [18] which is a maintained project aiming at preserving the MiniSat/Glucose legacy.

[4] https://maxsat-evaluations.github.io/2021/

TABLE II
STATISTICS ON THE RUNTIME OF PROCEDURES IN `RC2` WITH DIFFERENT SAT ORACLES ON **WEIGHTED** BENCHMARKS

	Parsing		Compute		Adapt AM1		Minimize		Exhaust		Trim	
	abs	rel	abs	rel	abs	rel	abs	rel	abs	rel	abs	rel
`RC2-A-Glucose3`												
mean	6.707	0.428	36.792	0.173	4.179	0.145	0.000	0.000	49.731	0.228	0.365	0.008
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	400.178	1.000	2894.375	0.982	427.359	0.998	0.007	0.001	2562.246	0.985	42.600	0.157
median	0.148	0.339	0.028	0.061	0.008	0.010	0.000	0.000	0.028	0.051	0.000	0.001
`RC2-B-Glucose3`												
mean	5.146	0.349	40.586	0.129	3.934	0.121	61.825	0.315	25.773	0.076	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	365.097	1.000	3024.803	0.990	429.336	0.998	3026.639	0.976	1491.174	0.912	0.010	0.001
median	0.134	0.148	0.024	0.026	0.008	0.008	0.192	0.166	0.013	0.023	0.000	0.000
`RC2-C-Glucose3`												
mean	5.075	0.354	28.677	0.124	4.003	0.121	58.463	0.317	22.642	0.071	0.257	0.004
min	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	417.002	1.000	3028.002	0.961	430.930	0.998	3132.125	0.983	1446.559	0.851	28.443	0.058
median	0.144	0.152	0.022	0.026	0.008	0.007	0.178	0.154	0.011	0.024	0.001	0.001
`RC2-A-Maplechrono`												
mean	5.857	0.333	36.605	0.205	4.284	0.123	0.000	0.000	42.201	0.318	1.017	0.009
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	390.325	1.000	1776.640	0.992	451.236	0.998	0.009	0.001	1779.656	0.985	105.256	0.186
median	0.135	0.120	0.169	0.070	0.008	0.007	0.000	0.000	0.238	0.150	0.001	0.001
`RC2-B-Maplechrono`												
mean	5.163	0.294	21.111	0.163	3.939	0.106	51.758	0.265	27.412	0.166	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	368.242	1.000	758.911	0.986	437.650	0.998	2236.522	0.952	1541.358	0.985	0.019	0.001
median	0.139	0.086	0.099	0.030	0.008	0.005	0.318	0.086	0.089	0.052	0.000	0.000
`RC2-C-Maplechrono`												
mean	5.818	0.293	18.685	0.158	4.098	0.105	49.057	0.260	26.567	0.171	0.455	0.006
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	439.520	1.000	1013.682	0.987	435.952	0.998	2436.954	0.948	1229.961	0.956	39.269	0.154
median	0.144	0.091	0.094	0.031	0.008	0.005	0.337	0.089	0.100	0.049	0.001	0.001
`RC2-A-Mergesat3`												
mean	6.021	0.374	36.492	0.206	4.295	0.133	0.000	0.000	55.229	0.254	1.548	0.018
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	377.585	1.000	1602.950	0.976	452.126	0.998	0.010	0.001	1941.928	0.986	83.650	0.355
median	0.140	0.197	0.066	0.099	0.009	0.010	0.000	0.000	0.052	0.110	0.003	0.003
`RC2-B-Mergesat3`												
mean	5.304	0.309	41.213	0.154	3.971	0.112	75.073	0.313	42.450	0.104	0.000	0.000
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	372.215	1.000	2816.565	0.983	444.320	0.998	2693.492	0.976	3079.620	0.955	0.028	0.001
median	0.132	0.107	0.057	0.034	0.009	0.006	0.343	0.146	0.032	0.032	0.000	0.000
`RC2-C-Mergesat3`												
mean	6.051	0.312	36.214	0.155	4.063	0.110	80.054	0.298	40.218	0.106	2.291	0.010
min	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
max	435.943	1.000	2626.575	0.975	452.325	0.998	2553.352	0.957	2854.320	0.965	263.884	0.323
median	0.136	0.104	0.077	0.032	0.009	0.006	0.300	0.129	0.032	0.031	0.005	0.003

They represent a progression of sorts, since Maplechrono can be viewed as the solver incorporating all Glucose 3 heuristics, and Mergesat3 as the one incorporating all heuristics from both Glucose3 and Maplechrono.

The results of the experiments are summarized in Tables I, II, and also in Figure 1, 2 split in accordance with the class of benchmarks, unweighted for Table I and Figure 1 and weighted for Table II and Figure 2. The columns of the tables contain the measurements of runtime in seconds in *abs* column (absolute) and relative to the total runtime of the solver in table *rel*. The statistics are presented only for the benchmarks that have been successfully solved by each solver variant. As it will be seen later, they solved similar portion of benchmarks overall, thus making the comparison fair.

In fact, the variance in the absolute values in the parsing column is mainly due to slight runtime variations due to the use of different CPU cores, since the parser implementation is independent from the SAT oracle. However, we decided that it will be better to leave it in the Tables to better see the overall picture.

1) Cactus plots: Figures 1 and 2 contain the so-called cactus plots. To draw it, the runtimes of each algorithm over solved benchmarks are ordered in an increasing order (independently from other algorithms) and plotted as a line. It means, that the further to the right the line goes — the more benchmarks have been solved and the closer it is to the horizontal axis — the smaller is the average runtime. From the plots it is clear that indeed, the `RC2-A` configurations perform the worst, while configurations `RC2-B` and `RC2-C` are quite similar. Another interesting fact is that Glucose3 as a SAT

oracle outperforms both Mergesat and Maplechrono, however, not by much, since for the vast majority of benchmarks (about 350 unweighted and 400 weighted) all variants perform more or less the same.

2) General observations: From the median values in both tables it is clear, that typically, the SAT oracle calls in `RC2` are very short, but their number is significant. It means that instead of focusing on heuristics that help solving large and hard SAT instances (as the SAT competition solvers do), the SAT oracles employed by `RC2` are better off with heuristics targeting easy instances, possibly, taking into account the peculiar features of cardinality encodings employed by the algorithm.

3) Parsing: Let us look at the absolute and relative values of the time required to parse the problems. It is quite surprising to see that with the (absolute) mean time 2.4 seconds for unweighted benchmarks and about 5 seconds for weighted benchmarks, in relative terms parsing takes from 20 to 35 percent of overall runtime. One of the reasons for this behavior is that the average runtime of `RC2` on the considered benchmarks is quite small. Nevertheless, it is highly likely that implementing `RC2` in a language that will allow to make a better parser can easily yield a large average runtime reduction.

4) Adapt AM1: It is also quite curious to see how much the adaptation of intrinsic Atmost1 constraints can take: up to 400 seconds, and it takes from 10 to 15 percent on average. Here we can only repeat what was already mentioned with regards to parsing: that the Python implementation is likely hindering the performance of the algorithm in this particular procedure.

5) Compute, Exhaust and Minimize: The most surprising revelation for us was that the main procedure, that aims to solve the problem, actually takes only up to a third of total runtime on average. In fact, the Exhaust procedure in almost all cases appears to take more time than compute. The same goes for the Minimize procedure, which, surprisingly, takes more time on weighted benchmarks and less time on unweighted ones. Overall, treating the Compute procedure as the main one is mostly a misconception, since in the case of `RC2` all three procedures take more or less the same amount of time.

6) Trim: The Trim procedure is not enabled in `RC2-B`, it works only in the other two configurations. Nevertheless, from the tables is is clear that it takes the least amount of time, usually negligible, however, it potentially can improve the performance of the solver in general.

7) Variations depending on the solver used as SAT oracle: From the vast amount of information in the Tables it is hard to quickly grasp the discrepancies that are produced by using a different SAT solver as an oracle, but they are there. For example, while the mean time spent on Compute and Exhaust by Maplechrono is smaller than that by the other two solvers, the median time is actually several times larger, meaning that more often than not the corresponding SAT oracle calls take more time. Apparently, this is the main reason why Maplechrono variants lack in performance as it can be seen from the cactus plots. Overall, the median times for Glucose 3 are the smallest, with Mergesat3 lagging slightly behind. Understanding the reasons behind the better behavior of Glucose 3 and behind the bad one by Maplechrono in the main procedures of `RC2` may well be the key for finding further improvements to the algorithm.

V. Conclusions and Future Work

In this paper we studied the `RC2` algorithm for solving MaxSAT from the SAT oracle perspective. The gathered statistics shed a lot of light on the interconnection between the algorithm and the employed SAT oracles and made it possible to outline a number of possible venues that may enable improving the `RC2` performance on average. These venues include implementing RC2 in C/C++ or at least taking measures towards drastically decreasing the time required to read WCNF; implementing Adapt AM1 heuristic in a SAT solver to reduce its runtime; tuning SAT solvers for good performance on easy instances instead of focus on hard instances characteristic for SAT competition solvers.

Acknowledgments

The research was supported by the Russian Science Foundation, project 22-21-00834.

References

[1] L. A. Wolsey, *Integer programming*. John Wiley & Sons, 2020.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*, ser. FAIA. IOS Press, 2021, vol. 336.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Satisfiability - Second Edition*, ser. FAIA. IOS Press, 2021, vol. 336, pp. 1267–1329.

[4] F. Bacchus, M. Järvisalo, and R. Martins, “Maximum satisfiabiliy,” in *Handbook of Satisfiability - Second Edition*, ser. FAIA. IOS Press, 2021, vol. 336, pp. 929–991.

[5] A. Ignatiev, A. Morgado, and J. Marques-Silva, “RC2: an efficient maxsat solver,” *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 53–64, 2019.

[6] R. Drechsler, T. A. Junttila, and I. Niemelä, “Non-clausal SAT and ATPG,” in *Handbook of Satisfiability - Second Edition*, ser. FAIA. IOS Press, 2021, vol. 336, pp. 1047–1086.

[7] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *TACAS*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[8] I. Lynce and J. Marques-Silva, “Haplotype inference with boolean satisfiability,” *Int. J. Artif. Intell. Tools*, vol. 17, no. 2, pp. 355–387, 2008.

[9] G. V. Bard, *Algebraic Cryptanalysis*. Springer US, 2009.

[10] J. a. P. Marques-Silva and K. A. Sakallah, “Grasp: A search algorithm for propositional satisfiability,” *IEEE Trans. Comput.*, vol. 48, no. 5, p. 506–521, may 1999.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *DAC*, 2001, p. 530–535.

[12] N. Eén and N. Sörensson, “Temporal induction by incremental sat solving,” vol. 89, no. 4, pp. 543–560, 2003.

[13] O. Roussel and V. M. Manquinho, “Pseudo-boolean and cardinality constraints,” in *Handbook of Satisfiability - Second Edition*, ser. FAIA. IOS Press, 2021, vol. 336, pp. 1087–1129.

[14] A. Ignatiev, A. Morgado, and J. Marques-Silva, “PySAT: A Python toolkit for prototyping with SAT oracles,” in *SAT*, 2018, pp. 428–437.

[15] N. Eén and N. Sörensson, “An extensible sat-solver,” in *SAT*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.

[16] O. Bailleux and Y. Boufkhad, “Efficient cnf encoding of boolean cardinality constraints,” in *CP*, ser. LNCS, vol. 2833, 2003, pp. 108–122.

[17] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, “Incremental cardinality constraints for maxsat,” in *CP*, ser. LNCS, vol. 8656, 2014, pp. 531–548.

[18] N. Manthey, “The mergesat solver,” in *SAT*, ser. LNCS, vol. 12831. Springer, 2021, pp. 387–398.