Project report

IN4336 Combinatorial Algorithms
Project 3: Graph Coloring: applications, approximations and solvers

David Hoepelman
1521969

Luis Garcia Rosario
4062949

ABSTRACT

In this paper we perform a comparison of solvers for the Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and Integer Linear Programming (ILP) problems by translating instances of the Graph Coloring (GC) and the Resource Constrained Project Scheduling Problem (RCPSP) problems into encoding which these solvers can handle.

Because we used the optimization versions of GC and RCPSP we could not directly use the decision SAT and SMT solvers. We formulated two approaches to iterative over the candidate optimal solutions and compared them against each other.

GC was successfully encoded for all solvers, but the SAT and SMT version suffered from having quadratic complexity in the value of the optimal solution. We were not able to successfully encode and test RCPSP to SAT, because the only known translation has exponential complexity.

In our GC tests ILP was able to solve more instances, but SAT and SMT were several magnitudes fasted on small problem instances.

1. INTRODUCTION

Solvers for NP-Hard problems can be used to solve instances of other NP-Hard problems by polynomially transforming the other problem instance. This is called an encoding. This possibility makes it feasible to make highly-optimized solving algorithms called solvers for certain relatively general NP-Hard problems.

For most popular problems multiple solvers exists, and benchmarks are available or contests are held to compare them. However comparisons between problems are rarer and in this project we will do such a comparison. We want to see if some problems are more suitable to encode other problems in or if the choice of problem to encode to makes little impact. Therefore we formulated the following research question for this project:

RQ Is there a speed difference between different target problems when solving an identical translated problem instance?

In answering our research questions we will focus on solvers for *Integer Linear Programming* (ILP), *Satisfiability* (SAT) and *Satisfiability Modulo Theories* (SMT).

As problems to encode we will focus on the *Resource Constrained Project Scheduling Problem* (RCPSP) and *Graph Coloring* (GC).

2. TARGET PROBLEMS

Satisfiability (SAT)

The satisfiability problem is informally defined as given a boolean formula in conjunctive normal form (CNF), find an assignment that makes the formula true. A formula is in CNF if it is a conjunction of clauses $(C_1 \wedge C_2 \wedge \ldots \wedge C_n)$ where each clause is a disjunction of literals $(l_1 \vee l_2 \vee \ldots \vee l_n)$ where each literal is a variable or the negation of a variable (p or $\neg p$). All propositional formulas can be converted to CNF. A SAT solver will find either a valid assignment for the given formula, or will report that it is unsatisfiable.

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories is a generalization of satisfiability. A SMT problem instance is still a given formula, and the problem is to find out if it is satisfiable. However, a variable can now be a predicate, a function which evaluates to a boolean, over non-boolean variables and operators. Commonly SMT solvers support integer, real, integer array and bitvector variables and linear or non-linear arithmatic, but other extensions are possible.

The concrete advantage of this is that certain constraints which are hard to specifiy in a pure boolean formula can now be specified in a more natural language.

Integer Linear Programming (ILP)

An Integer Linear Program is a linear program but which in which additional constraints can be added so variables can only be integers. While solving linear programs is in P, solving integer linear programs is NP-Hard. An ILP consists of a number of variables $x_0, \ldots x_n$, a linear function f to maximize or minimize and constraints which all follow the form of $a_0x_0 + \ldots + a_nx_n \leq b$.

A specialized form of the ILP is the binary integer linear program, where every variable is a binary variable.

3. SOLVING OPTIMIZATION PROBLEMS WITH DECIDABILITY SOLVERS

A decidability solver finds a feasible solution to a problem instance. In practice one does often not only which to find *a* feasible solution, but one wants to find *the best* feasible solution out of all feasible solutions. This is called an optimization problem.

ILP is an optimization problem and can thus naturally be used as a target for optimization problems. To use decidability solvers for an optimization problem one can formulate the optimization problem as a sequence of decision problems P with cost $c \in \mathbb{N}$. In each step the solver looks for a solution to P_c.

We independently though of two ways to go through this sequence, but they are identical too the approaches described by Wille et. al [24]. We will use their terminology.

The first approach called the *iterative approach* starts at the lower bound of the possible costs $c = LB$, usually $LB = 0$. While the solver can find no solution for P_c, c is incremented. In pseudocode:

```
$c \leftarrow UB$
while not $solvable$ do
    $P \leftarrow encode(instance, c)$
    $solvable \leftarrow solve(P)$
    if not $solvable$ then
        $c++$
    end if
end while
$OPT \leftarrow c$
```

This approach has the obvious disadvantage that $OPT - LB$ iterations are needed which is especially problematic if OPT is large

An alternative approach is too determine increasingly better lower and upper bounds. If P_c is determined feasible, one can look for a solution to $P_{c'}$ with $c' < c$. Likewise if P_c is unfeasible, one can look for a solution to $P_{c'}$ with $c' > c$. The minimal cost OPT is found when P_{OPT-1} is unfeasible, and P_{OPT} is feasible. This is called to approximation approach. In pseudocode:

```
while $LB \neq UB - 1$ do
    $c = LB + \lceil \frac{UB-LB}{2} \rceil$
    $P \leftarrow encode(instance, c)$
    $solvable \leftarrow solve(P)$
    if $solvable$ then
        $UB \leftarrow c$
    else
        $LB \leftarrow c$
    end if
end while
$OPT \leftarrow UB$
```

By performing a binary search over all possible c, we set the maximum number of iterations to $\lceil log_2(UB - LB) \rceil$.

During our experiments we encountered the problem that the first problem $P_{\frac{1}{2}UB}$ took a long time to encode, even though it usually was satisfiable. To get around this better upper bounds were needed, for which we tested both theoretical upper bounds and approximation algorithms.

4. GRAPH COLORING

The graph coloring problem is one of the Karp's 21 original NP-Complete problems [13]. Graph coloring exists in several variants, but they are all reducible to the vertex coloring problem. We chose the optimization variant of the vertex coloring problem which is informally defined as given an undirected graph $G = (V, E)$, find the minimal number of colors k needed to color each vertex so that no adjacent vertex shares a color. More formally:

$$\begin{aligned}
&G = (V, E) && E \subseteq V^2 \\
&map : V \to [1, k] && k \in \mathbb{N} \\
&\text{with } map(v) \neq map(w) && (v, w) \in E \\
&VC = min\{k \mid map : V \to [1, k]\}
\end{aligned}$$

4.1 Bounds

The possible values of k have clearly defined bounds. All graphs except the empty graph will need at least 1 color, and all graphs with an edge will need at least 2 colors. We stuck with an lower bound of 0 for the approximation approach and 1 for the iteration approach.

The trivial upper bound for a graph is $|V|$ colors, as every vertex can have its own unique color in that case. Another useful upper bound is $max(degree(v)) + 1$ for $v \in V$, the degree being the number of neighbors or edges of a node. This is an upper bound because a greedy algorithm can easily color a graph with this many colors by coloring every node using a color which none of its neighbors have. As every node is guaranteed to have at most $max(degree(v))$ neighbours at most $max(degree(v)) + 1$ colors are needed to color the whole graph using a greedy algorithm.

4.2 SAT encoding

As satisfiability is a decision problem k needs to be a parameter of the transformation, see section 3 for how we determined k.

We adopted the encoding from [22]. We define a variable for every node and color combination $p_{vi}(v \in V, i \in [1, k])$. If node v has color i then p_{vi}, otherwise $\neg p_{vi}$. The following clauses ensure that the assignment will be valid for the GC instance:

$$\bigvee_{1 \leq i \leq k} p_{vi} \qquad v \in V \tag{2a}$$

$$\neg p_{vi} \vee \neg p_{vj} \qquad v \in V, 1 \leq i < j \leq k \tag{2b}$$

$$\neg p_{vi} \vee \neg p_{wi} \qquad v \in V, 1 \leq i \leq k \tag{2c}$$

Clauses of type 2a ensures that every node has one or more colors. Clauses of type 2b ensures every node has at most one color. Clauses of type 2c ensure that nodes that share an edge do not have the same color. The encoding uses $O(|V|k)$ variables and $O(|E| + |V| \cdot k^2)$ clauses.

There is another possible encoding which uses $O(|V| \cdot \lceil log_2 k \rceil)$ variables and $O(2^{\lceil log_2 k \rceil} \cdot |E| + (2^{\lceil log_2 k \rceil} - k) \cdot |V|)$ clauses which shows promising results in [24]. Unfortunately we did not have time to implement and test this encoding and refer to the related and future work section 9.

4.3 SMT encoding

Our SMT encoding is identical to our SAT encoding. There are several candidate encodings which do use this additional expressive power, which are detailed in the future work section.

4.4 ILP encoding

We adopted the binary ILP encoding for GC from Faigle et al. [11]. We define the following variables: x_{ik} which is 1 iff node i has color k and y_k which is 1 iff color k is used. As ILP is an optimization problem and we have an upper bound we can create variables for $k \leq |V|$. The objective function $\min \sum_{k=1}^{n} y_k$ ensures that the minimal number of colors will be used. We then define the following constraints:

$$\sum_{k=1}^{n} x_{ik} = 1 \qquad i = 1, ..., n \tag{3a}$$

$$x_{ik} - y_k \leq 0 \qquad i,k = 1, ..., n \tag{3b}$$

$$x_{ik} + x_{jk} \leq 1 \qquad (i,j) \in E, k = 1, ..., n \tag{3c}$$

$$0 \leq x_{ik}, y_k \leq 1 \tag{3d}$$

$$x_{ik}, y_k \in \mathbb{Z} \tag{3e}$$

3a ensures that every node only has one color. 3b ensures that whenever a color is assigned to a node, that color is counted as used. 3c ensures that nodes which share an edge do not have the same color. 3d and 3e ensure that all variables are binary.

These variables and constraint together ensure that a valid solution to the GC instance can be constructed from the x_{ik} variables.

5. RCPSP

The Resource Constraint Project Scheduling Problem (RCPSP) is informally defined as given a project with resource limits, jobs with dependencies and resource consumption of jobs, find an optimal schedule so that at no time too much resources are consumed and the project is completed at the earliest time possible.

5.1 Formal definition and notation

Given a project P with:

- A set of jobs $V = \{1, \ldots, N\}$
- A set of resources $\mathcal{R} = \{1, \ldots, M\}$
- Precedence relations $(i,j) \in E \subset V \times V$ in which (i,j) means job j is dependent upon job i
- Durations $d_i \in \mathbb{N}$ for each job $i \in V$
- Resource limits R_j for each $j \in \mathcal{R}$
- Job resource consumptions r_i, j for each job $i \in V$ and resource $j \in \mathcal{R}$

Define:

- The start time $s_i \in \mathbb{N}$ of job i in the schedule
- The finish time $f_i = s_i + d_i$ of job i
- The in-progress status $u_{i,t} \in \{0,1\}$ of job i at time t in the schedule

Find the minimal make-span $T = \max_{i \in V}\{f_i\}$ subject to:

- Dependencies:
  $f_i \leq s_j$ for every $(i,j) \in prec$
- Resource constraints:
  $\sum_{i \in V} u_{i,t} \cdot r_{i,j} \leq R_j$ for every $j \in R, t \in [0,T]$

In addition we define job 0 and job $N+1$, which are a super-source and super-sink. Both have duration 0 and use no resources. Alls jobs which have no other dependency depend on job 0, job $N+1$ depends on all jobs on which no other job depends. Thus $s_0 = f_0 = 0$ and $s_{N+1} = f_{N+1} = T$

5.2 Bounds

All encodings need to use lower and upper bounds for all s_i and f_i. An upper bound for $T = f_{N+1}$ is given by a sequential schedule, in which no two jobs run concurrently. This means that $T \leq \sum_{i \in V} d_i$. If such a schedule is not possible then no schedule is viable as there is a single job which consumes too much resources so this is a safe upper bound. $s_0 = 0$, because otherwise the makespan could be improved by moving each job s_0 time steps earlier.

Using this information we can compute the bounds for each s_i and f_i by calculating the critical path bounds which are based on ignoring the resource constraints and looking at the graph of job dependencies [14].

We can calculate for job i the earliest start es_i and earliest finish $ef_i = es_i + d_i$ by realizing that a job can never start before all its dependencies have finished. Because we know $es_0 = ef_0 = 0$ we can now calculate every earliest start:

$$es_j = max\{ef_i | (i,j) \in E\}$$

Similarly we can calculate for job i the latest start ls_i and latest finish $lf_i = ls_i + d_i$ by realizing that a job must finish before all its dependents can start. Because we know $ls_{N+1} = lf_{N+1} = \sum_{i \in V} d_i$ we can now calculate every latest finish:

$$lf_i = min\{ls_j | (i,j) \in E\}$$

These bounds can be calculated in $O(n^2 \log n)$ time, or $O(n^3)$ time using the Floyd-Warshall algorithm [12].

5.3 SAT

Our encoding is based on Horbach [12]. To encode RCPSP into SAT we define two types of variables.

Firstly we define a variable for every start time and job combination:

$$s_{it} \text{ for } i \in V, t \in [es_i, ls_i]$$

Secondly we define a variable for every time in which a project can be active:

$$u_{it} \text{ for } i \in V, t \in [es_i, lf_i]$$

We then need some consistency clauses:

$$\bigvee_{t \in [es_i, ls_i]} s_{it} \qquad i \in V \tag{4a}$$

$$s_{it} \implies u_{il} \qquad i \in V, t \in [es_i, ls_i], l \in [t, t+d_i) \tag{4b}$$

$$\neg s_{jt} \quad \bigvee_{l \in [es_i, es_j - d_i]} s_{il} \qquad (i,j) \in E, t \in [es_j, ls_j] \tag{4c}$$

4a ensures that every job starts. 4b ensures that if job i starts at time t, that it then is in progress during $[t, t + d_i]$. 4c ensures that if job j depends on job i, that job j will not start before all of its dependencies have had time to complete.

For the last set of clauses we need to define a *cover*. $C \subset V$ is called a cover for any resource $j \in \mathcal{R}$ $\sum_{i \in C} r_{ij} > R_j$ or in other words C is a cover if its jobs exceed a resource limit. A cover C is minimal if you cannot remove any job while remaining a cover.

The following clauses ensure that the jobs in progress at one time will never be a cover

$$\bigvee_{i \in C} \neg u_{it} \; t \in [0, T], C \text{ is a minimal cover}$$

Unfortunately the number of minimal covers can be exponential, so this encoding is exponential without further tricks. We will come back to this in the experiments section. Currently no polynomial SAT encoding of RCPSP is known [3].

5.4 SMT

The SMT encoding or RCPSP is identical to the SAT encoding, but instead of a clause for every minimal cover the following constraint is added:

$$\sum_{i \in V, es_i \leq t \leq lf_i} u_{it} \cdot r_{ij} \leq R_j \text{ with } t \in [0, T], j \in \mathcal{R}$$

This ensures that a no point the resource usage of all in progress jobs exceeds the limit for one of the resources. Because these a polynomial number of clauses with polynomial size, the SMT encoding is polynomial.

5.5 ILP

For our RCPSP in ILP encoding we adopted the binary ILP models from Horbach [12] and Mingozzi et al [18]. For each job $i \in V$ and for each period $t \in \{es_i, ..., ls_i\}$ a binary variable s_{it} indicates whether an activity i starts at period t. Additionally we defined in process variables u_{it} which are 1 iff job i is in process at time t.

Our objective function minimizes the makespan:

$$\min(\sum_{t = es_{n+1}}^{ls_{n+1}} t \cdot s_{n+1,t})$$

.

Because only 1 $s_{n+1,\_}$ can be 1 in a feasible solution this is equivalent to $\min T$

We then define the following constraints:

$$\sum_{t \in [es_i, ls_i]} s_{it} = 1 \qquad i \in V \tag{5a}$$

$$u_{il} - s_{it} \geq 0 \qquad i \in V, t \in [es_i, ls_i], l \in [t, t + d_i) \tag{5b}$$

$$\sum_{l \in [es_j, t - d_j]} s_{jl} \geq s_{it} \qquad (j, i) \in E, t \in [es_i, ls_i] \tag{5c}$$

$$\sum_{i \in V, es_i \leq t \leq lf_i} r_{ik} u_{it} \leq R_k \qquad t \in [0, T], k \in R \tag{5d}$$

$$s_{00} = 1 \tag{5e}$$

$$s_{it} \in 0, 1 \tag{5f}$$

$$u_{it}, x_{it} \in \mathbb{Z} \tag{5g}$$

5a ensure that all projects start. 5b ensures that projects are in process for their duration after they are started. 5c ensures that all precedence relations are respected. 5d ensures that at no moment too much resources are used. 5e, 5f and 5g are some bookkeeping.

6. EXPERIMENTAL SETUP

We performed our experiments on a VMware VM running Ubuntu Linux server 14.04LTS x86_64 running under a Windows 7 x64 installation. One core of an Intel Core i5-2500K 3.3Ghz, 4 GB DDR3-1600Mhz and 50GB of a SSD were available to the VM.

6.1 Solvers

For our SAT solver we used a solver called Lingeling [8] because it performed well in a wide number of categories in the SAT Competition 2014 [7] and is freely available for research. It accepts the common DIMARCS CNF format [9] as an encoding for SAT instances in which we stored our translated instances.

As a SMT solver we chose Z3 [10] because it is freely available and supports a wide range of operators. Z3 accepts a variant of the SMT-LIB version 2 [6] language which we used to store our translated instances.

As an ILP solver we used the commercial Gurobi optimizer [2] because it accepts a wide variety of LP-related models and input formats, because it is freely available for academic purposes and because it is one of the best-performing ILP solvers available at the moment [17]. Gurobi accepts a wide variety of input formats. We used the proprietary but human-readable Gurobi LP format to store our translated instances.

6.2 Graph Coloring

For graph coloring we used the instances from COLOR 02/03/04 [1]. These instances have several different and contain both generated and real-world instances of graph coloring problem. We only used the 66 instances for which the optimal solution was known, because we did not expect to be able to solve instances which were previously unsolved and we wanted to be able to determine if our encodings were faulty.

We wrote python scripts to translate the graph coloring instances from their source format into the target format

Solver	Upper bound	Approach
SAT	Highest degree	Approximation
SAT	Greedy	Approximation
SAT	-	Iterative
SMT	Greedy	Approximation
ILP	Highest degree	-
ILP	Greedy	-

Table 1: Combinations tested for graph coloring

Approach	Total solve time
SAT Degree	632s
SAT Greedy	674s
SAT Iterative	726s
SMT Greedy	570s
ILP Degree	2240s
ILP Greedy	3068s

Table 2: Total solve time of instances solved on all solvers

using the encodings detailed in section 4. While python is not a very fast language, we did not expect to translations to take a significant amount of time thus justifying the use of a higher level language. We then created a benchmark wrapper around the solvers, which would call the translation script and the solver for each instance while measuring time taken for each action and stopping the solver at a specified timeout. For out experiments we set this timeout per instance at 1800s. This benchmark wrapper also took care of the iterations needed to solve optimization problem with decidability solvers as detailed in section 3. All scripts and instances are available on our GitHub source repository.

The solver, bound and approach combinations tested can be found in Table 2. We did not test the SMT-highest degree upper bouund because the greedy approximation algorithm gives a strictly better bound. We did not test the SMT-iterative approach because the experiments could not be completed in time.

6.3 RCPSP

For RCPSP we were planning to use the instances from PSPLIB [15]. We wrote python scripts to translate the graph coloring instances from their source format into the target format using the encodings detailed in section 5.

While testing our SAT translation we found that even with small instances with 30 jobs our script quickly ran out of memory while generating the minimal set covers. While we already knew that the SAT translation was exponential, the authors of the translation reported success with very small problem instances. We did not manage to replicate this success. The authors of the encoding circumvented this problem by modifying a SAT solver and dynamically adding clauses when necessary. We deemed such modifications outside of the scope of this project.

We decided to focus the rest of our experiments on graph coloring and did not pursue RCPSP any further. We refer to the related and future work section 9

7. RESULTS

Our raw result data is listed in Table 3 in the appendix. We managed to result find optimal solutions for **44 out of 67** instances tested. Our first result is that the all SAT approaches and the SMT solver performed approximately the same. SAT iterative and SAT greedy both managed to solve 26 instances, one more than both SAT degree and SMT. 6 instances were solved with SAT which were not solved with ILP.

We also see that all SAT approaches perform broadly the same and that only SAT greedy outperforms SAT degree because of the better upper bound which means possibly less iterations. The reason for this is that k's not close to the optimum are checked very quickly because they are either over-constrained when k is too small or under-constrained when k is too large. As such the approximation approach can quickly converge its bounds. In our experiments we saw that usually withing one or a few minutes the bounds were narrowed down to $[OPT-1, OPT+1]$ and that most of the time was spend on determining whether OPT was satisfiable. There were a few instances though where most of the time was spent on the $OPT-1$ instance. We saw that in general the SAT solvers could more quickly determine if an instance was satisfiable rather than determine that it was unsatisfiable.

ILP degree and ILP greedy performed very comparably, both managed to solve exactly the same instances. ILP solved a larger amount of instances than SAT and SMT with 34 instances solved, of which 14 were not solved by SAT and SMT. A counterintuitive result for us was that ILP degree, which has a worse upper bound, performed better than ILP greedy which had a better upper bound and thus smaller problem file size. ILP greedy needed 8459 seconds to solve all instances solved, while ILP degree needed 7066 seconds to solve the same instances. ILP greedy performed between 5% better to 200% worse than ILP degree on individual instances.

The most interesting instances are those that could not be solved by either ILP or SAT/SMT. The instances that SAT/SMT could solve but ILP could not are from two types. The first one is a random graph and does not seem to be that special. We did not find an explanation as to why the SAT family could solve it in under a second but ILP could not. The other instances are all Leighton graphs, which are graphs with a certain type of asymmetry. It is well known that symmetry in the instance is a problem for both SAT and ILP solvers [19]. It could be that the SAT and SMT solver greatly benefit from this kind of asymmetry, while ILP does not. The instances solved by ILP but not by SAT all share the property have the highest OPT of all instances solved. As the translation for SAT was quadratic in k, while the one for ILP was linear in k this is probably the reason why ILP could solve these instances and SAT and SMT couldn't. Another interesting fact is that all problems ILP could solve but SAT couldn't, the greedy algorithm found either an optimal solution or a very close one. For one type of instances it always found an optimal coloring, and for the other type of instances it was less than 10% above the optimal.

Table 2 shows the total time needed to solve the instances which could be solved with all approaches. It should be noted that while SMT looks faster than SAT, all SAT approaches are faster if instances which could not be solved on ILP are taken into account. Figure 1 shows the same solving time, but split up per instance. It is interesting to see that while ILP solved more instances, it is generally much slower

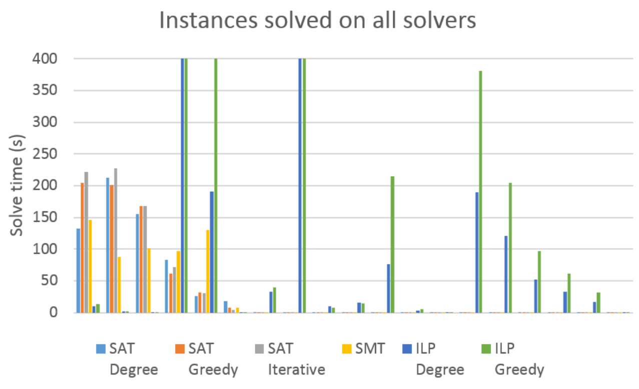

Figure 1: Solve time per instance for instances solved with all approaches

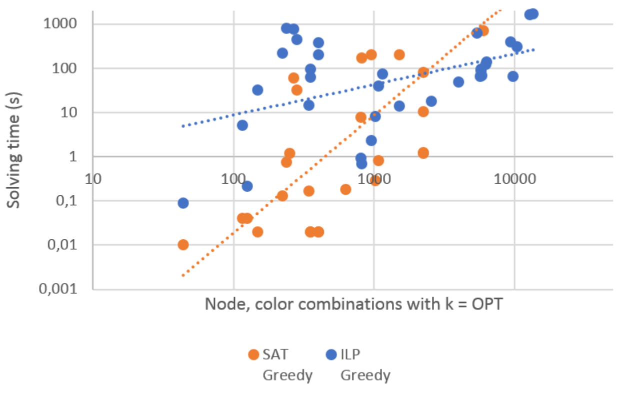

Figure 2: Solve time plotted against possible node, color combinations $|N| * k$ with $k = OPT$

than SAT and SMT.

We then looked for a possible correlation between properties of the instance. The solving time does not show any correlation with N, M, NM, $\frac{M}{N*(N-1)}$ (edge density) or OPT. The best correlation we found can be seen in figure 2 and is the number of color combinations possible with optimal colors $N * OPT$. The correlation is still weak at best, but for both SAT and ILP a general increase in solving time can be seen as it increases, which was not the case for the other metrics. It can be seen that the solving time is much more influenced by it for SAT than for ILP, which is a possible explanation as to why ILP was able to solve a few of the larger instances.

8. CONCLUSION

Our testing has not been as extensive as we initially had planned, primarily we can base our conclusion on only four distinct encodings while lots more are possible.

Our first result is that not all solver can be used to solve all problems. While we know from theory that there is a polynomial transformation possible from every NP problem to every NP-Hard problem, that transformation is apparently not always known to the scientific community. As such we can safely answer our research question with a no: not all solvers types can be used for all problems.

Our Graphs Coloring SAT, SMT and ILP comparison gives further insight into this question. We shown that the specific approach makes relatively little difference and that the choise of solver type is much more important. Specifically we've seen that for graph coloring ILP is able to solve a broader set of graph coloring instances and is less susceptible to increased problem size, but does perform worse on smaller instances than SAT and SMT do.

9. RELATED AND FUTURE WORK

In the literature there are some interesting previous work on solving graph coloring and resource constraint project scheduling problem. Here, we will discuss few of them and their methods that may have been relevant to our research. Our work was based on earlier work done simmilar to, among others, [19], where they optimally solve hard combinatorial problems (graph coloring as case study) by the reducing these to generic problems such as SAT and 0-1 ILP. They used a technique called symmetry breaking in an attempt to improve the performance of reduction-based methods, which are not competitive with problem-specific methods. This technique may be used together with the academic 0-1 solver PBS II to obtain bether performance results in future work. Besides focusing on solving the graph coloring problem, we would have liked to expand our research on the RCPSP problem encoding to compare the performance of the ILP and SMT solvers. For the RCPSP problem there are a few papers that we found interesting. As in [12], they adapted a satisfiability solver for there specific domain of the (RCPSP) problem. They were able to close several medium and large size benchmark instances from the PSLIB of the RCPSP that have never been closed before by providing tighter lower bounds and by finding bether feasible solutions. They cover the problem of dealing with linear inequalities over Boolean variable that can cause the number of clauses needed for the represention of many combinatorial problems to grow exponentially. We find this methods interesting enough to exploit further. In [14] they used methonds and techniques for reducing problem data of RCPSP which can be exploited with the *destructive improvement technique* that restricts a problem by setting a maximal objective function value F and try to contradict the feasibility of this reduced problem. This too might be interesting to evaluate the performance improvement. We implemented an ILP encoding of the RCPSP problem to compare the solving time with the SAT encoding, but we didn't have the SAT solving times to compare with. It can be usefull to explore the SMT encoding of RCPSP, as it can work with among others integer, integer array and bitvector variables, and use its solving times and compare it with the ILP encoding.

10. REFERENCES

[1] Color02/03/04: Graph coloring and its generalizations.

[2] Gurobi optimizer.

[3] Mohammad Abdolshah. A review of resource-constrained project scheduling problems (rcpsp) approaches and solutions. 2014.

[4] Ignasi Abío and Peter J Stuckey. Encoding linear constraints into sat. In *Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[5] Rui Alves, Filipe Alvelos, and Sérgio Dinis Sousa. Resource constrained project scheduling with general precedence relations optimized with sat. In *Progress in Artificial Intelligence*, pages 199–210. Springer, 2013.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[7] Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. Sat competition 2014. 2014.

[8] Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. *SAT COMPETITION 2014*, page 39, 2014.

[9] DIMACS Challenge. Satisfiability: Suggested format. *DIMACS Challenge. DIMACS*, 1993.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] Ulrich Faigle, W. Kern, and Georg Still. *Algorithmic Principles of Mathematical Programming*. Springer, 2002.

[12] Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 181(1):89–107, 2010.

[13] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[14] Robert Klein and Armin Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research*, 112(2):322–346, 1999.

[15] Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997.

[16] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[17] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, 2012.

[18] Aristide Mingozzi, Vittorio Maniezzo, Salvatore Ricciardelli, and Lucio Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, 44(5):714–729, 1998.

[19] Arathi Ramani, Fadi A Aloul, Igor L Markov, and Karem A Sakallah. Breaking instance-independent symmetries in exact graph coloring. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 324–329. IEEE, 2004.

[20] Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011.

[21] Prakash C Sharma and Narendra S Chaudhari. Polynomial 3-sat encoding for k-colorability of graph. *Special Issue of International Journal of Computer Application on Evolution in Networks and Computer Communications (1)*, pages 19–24, 2011.

[22] T̃exas University CS395T. Reducing graph coloring to sat. `https://www.cs.utexas.edu/users/vl/teaching/lbai/coloring.pdf`.

[23] Miroslav N Velev and Ping Gao. Exploiting hierarchical encodings of equality to design independent strategies in parallel smt decision procedures for a logic of equality. In *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pages 8–13. IEEE, 2009.

[24] Robert Wille, Daniel Große, Mathias Soeken, and Rolf Drechsler. Using higher levels of abstraction for solving optimization problems by boolean satisfiability. In *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*, pages 411–416. IEEE, 2008.

[25] Hao Wu, Rosemary Monahan, and James F Power. Exploiting attributed type graphs to generate metamodel instances using an smt solver. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 175–182. IEEE, 2013.

APPENDIX

Table 4: Graph coloring instance types

Type	Description
CAR	Based on MYC graphs with added nodes with identical density
DSJ	Random graphs made by Johnson et al. [13].
GOM	Latin squares
LEI	Leighton Graphs, these have specic kinds of asymmetry
MIZ	No information was available about this type
MYC	Graphs made from Mycielski transformations, which ensures graphs without triangles
REG	Graphs made from register allocation instances from specic programs
SGB-B	Graphs based on character interactions within books
SGB-G	Graphs based on footballs teams playing against each other
SGB-M	Geometric graphs of cities connected if they are close enough
SGB-Q	Queen graphs, giving the number of queens that can be placed on an NxN graph without them attacking each other

Table 3: Solve time in seconds for all GC instances

Instance	Type	N	M	OPT	Deg + 1	Greedy	SAT Deg.	SAT Greedy	SAT It.	SMT	ILP Deg.	ILP Greedy
1-Insertions_4	CAR	67	232	4	23	5	83,72	61,05	71,57	96,56	751,44	751,44
2-Insertions_3	CAR	37	72	4	10	4	0,05	0,02	0,03	0,03	16,34	31,47
2-Insertions_4	CAR	149	541	4	38	5						
3-Insertions_3	CAR	56	110	4	12	4	0,18	0,13	0,15	0,11	76,67	215,06
3-Insertions_4	CAR	281	1046		57	5						
4-Insertions_3	CAR	79	156	3	14	4	0,77	0,76	0,88	0,45	732,79	789,4
4-Insertions_4	CAR	475	1795		80	5						
DSJC125.1	DSJ	125	736	5	24	8	0,23	0,18	0,18	0,15		
DSJC125.5	DSJ	125	3891		76	26						
DSJC125.9	DSJ	125	6961		121	56						
DSJR500.1	DSJ	500	3555	12	26	15		716,01	951,31			
qg.order30	GOM	900	26100	30	59	32						
qg.order40	GOM	1600	62400	40	79	64						
qg.order60	GOM	3600	212400	60	119	64						
qg.order100	GOM	10000	990000	100	199	128						
le450_5a	LEI	450	5714	5	43	14	9,98	10,49	0,56	25,69		
le450_5b	LEI	450	5734	5	43	13	29,54	78,76	1,29	270,02		
le450_5c	LEI	450	9803	5	67	17	3,74	1,18	0,82	4,75		
le450_5d	LEI	450	9757	5	69	18	9	1,22	0,79	5,02		
le450_15a	LEI	450	8168	15	100	22						
le450_15b	LEI	450	8169	15	95	22						
le450_15c	LEI	450	16680	15	140	30						
le450_15d	LEI	450	16750	15	139	31						
le450_25a	LEI	450	8260	25	129	28						
le450_25b	LEI	450	8263	25	112	27						
le450_25c	LEI	450	17343	25	180	37						
le450_25d	LEI	450	17425	25	158	35						
mug100_1	MIZ	100	166	4	5	4	0,03	0,02	0,04	0,03	121,41	204,57
mug100_25	MIZ	100	166	4	5	4	0,03	0,02	0,04	0,03	189,6	381,3
mug88_1	MIZ	88	146	4	5	4	0,03	0,02	0,03	0,03	33,16	61,55
mug88_25	MIZ	88	146	4	5	4	0,03	0,02	0,03	0,03	51,95	96,52
myciel3	MYC	11	20	4	6	4	0,02	0,01	0,01	0,02	0,13	0,09
myciel4	MYC	23	71	5	12	5	0,06	0,04	0,05	0,05	3,52	5,05
myciel5	MYC	47	236	6	24	6	26,34	31,39	31,06	130,04	190,64	452,03
myciel6	MYC	95	755	7	48	7						
myciel7	MYC	191	2360	8	96	8						
fpsol2.i.1	REG	496	11654	65	253	65						
fpsol2.i.2	REG	451	8691	30	347	30					1607,03	1661,68
fpsol2.i.3	REG	425	8688	30	347	30					1390,73	1639,06
inithx.i.1	REG	864	18707	54	503	54						
inithx.i.2	REG	645	13979	31	542	31						
inithx.i.3	REG	621	13969	31	543	31						
mulsol.i.1	REG	197	3925	49	122	49					52,14	65,53
mulsol.i.2	REG	188	3885	31	157	31					59,02	66,69
mulsol.i.3	REG	184	3916	31	158	31					61,67	65,3
mulsol.i.4	REG	185	3946	31	159	31					68,68	94,96
mulsol.i.5	REG	186	3973	31	160	31					71,37	87,64
zeroin.i.1	REG	211	4100	49	112	49					246,82	299,98
zeroin.i.2	REG	211	3541	30	141	30					93,04	135,51
zeroin.i.3	REG	206	3540	30	141	30					75,35	119,95
anna	SGB-B	138	986	11	72	12	131,92	204,31	221,23	145,58	10,41	13,62
david	SGB-B	87	812	11	83	12	212,89	200,58	226,91	87,51	2,04	2,28
huck	SGB-B	74	602	11	54	11	155,58	167,33	167,67	101,16	0,66	0,7
jean	SGB-B	80	508	10	37	10	18,33	7,61	4,48	7,28	0,9	0,91
games120	SGB-G	120	1276	9	14	9	1,18	0,82	0,98	1,06	32,52	39,14
miles250	SGB-S	128	774	8	17	9	0,6	0,29	0,38	0,29	10,11	8,16
miles500	SGB-S	128	2340	20	39	22					18,16	18
miles750	SGB-S	128	4226	31	65	34					40,76	49,02
miles1000	SGB-S	128	6432	42	87	44					749,51	615,09
miles1500	SGB-S	128	10396	73	107	76					212,36	397,84
queen5_5	SGB-Q	25	320	5	17	8	0,09	0,04	0,03	0,07	0,28	0,21
queen6_6	SGB-Q	36	580	7	20	11	1,33	1,19	1,15	1,45		
queen7_7	SGB-Q	49	952	7	25	10	0,27	0,17	0,15	0,31	15,2	14,28
queen8_12	SGB-Q	96	2736	12	33	15					79,55	74,84
queen8_8	SGB-Q	64	1456	9	28	13						
queen9_9	SGB-Q	81	2112	10	33	16						