FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Assignment of bachelor's thesis

| | |
|---|---|
| **Title:** | Bounded Non-Linear Integer Constraint Solving |
| **Student:** | Leonid Burbygin |
| **Supervisor:** | doc. Dipl.-Ing. Dr. techn. Stefan Ratschan |
| **Study program:** | Informatics |
| **Branch / specialization:** | Computer Science |
| **Department:** | Department of Theoretical Computer Science |
| **Validity:** | until the end of summer semester 2023/2024 |

## Instructions

Solving non-linear integer constraints (i.e., conjunctions and disjunctions of equalities and inequalities over the integers) is an undecidable problem. Still, SAT modulo theory (SMT) solvers contain sophisticated algorithms that can solve many instances of such constraints. Especially, the problem can be made trivially decidable by adding finite bounds to all variables. In this case, the sophisticated algorithms contained in SMT solvers are often an overkill, and the goal of this thesis is to check, how far one can get with algorithms that solve such constraints by simply checking the constraints on the whole finite set of possible values.

1) Write a solver for bounded non-linear integer constraints based on the trivial "check all values" algorithm.
2) Compare its behavior against the SMT solver CVC5 using benchmark examples, for example from the SMTLIB database (https://smtlib.cs.uiowa.edu).
3) Improve the written solver in such a way that it can beat CVC5 in a few more cases.
4) Do systematic computational experiments that document the strong and weak points of the original trivial algorithm, the improved algorithm, and the SMT solver CVC5.

---

*Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 7 February 2023 in Prague.*

Bachelor's thesis

# BOUNDED NON-LINEAR INTEGER CONSTRAINT SOLVING

**Leonid Burbygin**

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: doc. Dipl. -Ing. Dr. techn. Stefan Ratschan
May 1, 2023

# Contents

# List of Figures



# List of Tables

# List of code listings

*I would like to thank all my teachers for their guidance and support, which has been instrumental in my academic journey. I am also grateful to my family for their unwavering love and encouragement throughout this process. Finally, I am thankful for the support of my friends and colleagues, who have been a constant source of motivation. Without them, this thesis would not have been possible.*

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of theAct.

In Praze on May 1, 2023 .....

# Abstract

Solving non-linear integer constraints (i.e., conjunctions and disjunctions of equalities and inequalities over the integers) is an undecidable problem. Still, SAT modulo theory (SMT) solvers contain sophisticated algorithms that can solve many instances of such constraints. Especially, the problem can be made trivially decidable by adding finite bounds to all variables. In this case, the sophisticated algorithms contained in SMT solvers are often an overkill, and the goal of this thesis is to check, how far one can get with algorithms that solve such constraints by simply checking the constraints on the whole finite set of possible values.

**Keywords** non-linear integer constraints, SAT modulo theory, SMT solvers, finite bounds, undecidable problem, sophisticated algorithms, constraints

# Abstrakt

Řešení nelineárních celočíselných omezení (tj. spojení a disjunkcí rovností a nerovností nad celými čísly) je nedostatečně rozhodnutelný problém. Nicméně SMT (SAT modulo teorie) řešiče obsahují sofistikované algoritmy, které dokáží vyřešit mnoho instancí takovýchto omezení. Zvláště problém může být triviálně rozhodnutelný přidáním konečných hranic ke všem proměnným. V tomto případě jsou sofistikované algoritmy obsažené v SMT řešičích často zbytečné a cílem této práce je zjistit, jak daleko lze s algoritmy dostat, které řeší taková omezení jednoduše kontrolou omezení nad celou konečnou množinou možných hodnot.

**Klíčová slova** nelineární celočíselná omezení, SAT modulo teorie, SMT řešiči, konečné hranice, nedostatečně rozhodnutelný problém, sofistikované algoritmy, omezení.

# Summary


## Introduction

In this thesis, I aim to address the problem of solving non-linear integer constraints, which is an undecidable problem. Despite its inherent difficulty, there are sophisticated algorithms contained in SMT solvers that can solve many instances of such constraints. However, in some cases, adding finite bounds to all variables can make the problem trivially decidable, and the use of such algorithms may be overkill. Therefore, I explore an alternative approach that solves such constraints by checking the constraints against the whole finite set of possible values. The goal of this thesis is to investigate the effectiveness of this approach and compare it to existing methods. To achieve this, I develop a methodology for solving non-linear integer constraints, which involves parsing the constraints, building an AST, and estimating the bounds of the constraints to generate a finite interval of possible values. Then I try all values in that interval to determine whether the constraint is satisfiable. Through my research, I aim to contribute to the development of new methods for solving non-linear integer constraints that are effective and fast, and have potential practical applications.

## Literature review

The literature review for this thesis involves an examination of existing research and documentation related to the problem of solving non-linear integer constraints. Specifically, I have reviewed the official documentation of the SMT-LIB standard, version 2.6, which was released in 2017 by Clark Barrett, Pascal Fontaine, and Cesare Tinelli. This standard provides a set of guidelines and specifications for SMT solvers, which are widely used for solving non-linear integer constraints. My review also includes an examination of the limitations of existing approaches, including the potential overuse of sophisticated algorithms in SMT solvers, and the benefits of using algorithms that check constraints against the whole finite set of possible values. Through this review, I aim to identify gaps in existing research and develop a deeper understanding of the challenges and opportunities associated with solving non-linear integer constraints.

## Methodology

The methodology used in this thesis involves a process for solving non-linear integer constraints by checking them against the whole finite set of possible values. Specifically, this involves reading a constraint, parsing it, and building an AST that represents the structure of the constraint. Then, using the AST, I estimate the bounds of the constraint and generate a finite interval of possible values. Finally, I try all values in that interval to determine whether the constraint is satisfiable. This approach does not require the use of sophisticated algorithms contained in SMT solvers and instead focuses on a simple yet effective method of solving non-linear integer constraints. By using this methodology, I aim to explore the effectiveness of this approach and compare it to other existing methods of solving non-linear integer constraints.

## Results

The SMT solver was developed, tested on different benchmarks, and compared to the original cvc5 solver. All the more detailed information is described in the chapter "Computational Experiments".

## Conclusion

The method of checking all integer values in some interval works faster than the original cvc5 solver, in 1 group of math problems, such as STC. And for that reason, it could be integrated into the original cvc5 to decrease the computational speed of constraints, if we could detect and classify the constraint type during the parsing.

# Seznam zkratek

| | |
|---|---|
| SMT | Satisfiability modulo theories |
| SAT | Boolean satisfiability problem or propositional satisfiability problem |
| AST | Abstract syntax tree |
| NP-complete | Nondeterministic polynomial-time complete |
| STC | Sum of Three Cubes problem |
| MC | Magic Square of Cubes problem |
| MF | Magic Five problem |
| MQ | Magic Quadratic problem |
| MS | Magic Square problem |
| RTA | Relaxed Taxicab Number problem |
| SC | Semi-Magic Square of Cubes problem |
| SF | Sum-Free problem |
| SQ | Semi-Magic Square of Fourth Power problem |
| TA | Taxicab Number problem |
| CSV | Comma-separated values |

# Introduction

In computer science and mathematical logic, satisfiability modulo theories (SMT) is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT) to more complex formulas involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings. The name is derived from the fact that these expressions are interpreted within ("modulo") a certain formal theory in first-order logic with equality (often disallowing quantifiers). SMT solvers are tools that aim to solve the SMT problem for a practical subset of inputs. SMT solvers such as Z3 and cvc5 have been used as a building block for a wide range of applications across computer science, including automated theorem proving, program analysis, program verification, and software testing.[1]

Satisfiability Modulo Theories solvers are powerful tools that have transformed the field of automated reasoning. These solvers have revolutionized the way complex mathematical problems are solved in various fields, including computer science, mathematics, and engineering. SMT solvers can handle a wide range of constraints, including non-linear integer constraints that involve conjunctions and disjunctions of equalities and inequalities over integers. This capability has made SMT solvers a valuable tool in many areas, from program analysis to formal verification.

Despite the sophistication of the algorithms used in SMT solvers, solving non-linear integer constraints is still an undecidable problem. However, by adding finite bounds to all variables, the problem can be made trivially decidable. This technique, known as bounded model checking, can significantly improve the quality of SMT solvers in some cases, potentially decreasing computing time and making them more efficient.

The purpose of my bachelor thesis aims to develop a simple SMT solver for non-linear integer constraints that will try all values in some interval. By doing this, it is possible to improve the quality of the cvc5 solver in a few more cases and potentially decrease the computing time. The developed solver will be based on the bounded model checking technique and will use a brute-force approach to test all values in the given interval.

To accomplish the goal of the thesis, several benchmarks will be used to compare the performance of the developed solver with that of the original cvc5 solver. The benchmarks will be carefully selected to cover a wide range of problem domains, including program analysis, formal verification, and optimization. The comparison will be based on several metrics, including the number of solved problems, the computing time, and the quality of the solution.

The development of a simple SMT solver for non-linear integer constraints has the potential to significantly improve the performance of existing SMT solvers. This improvement can have

a wide range of applications, from optimizing software to designing complex systems. Moreover, my implementation of the solver could be extended to handle other types of constraints, such as floating-point arithmetic and arrays, further increasing the applicability of the technique.

In conclusion, the proposed bachelor thesis aims to develop a simple SMT solver for non-linear integer constraints that will try all values in some interval. The developed solver will be based on the bounded model checking technique and will use a brute-force approach to test all values in the given interval. The comparison with the original cvc5 solver will be based on several metrics, including the number of solved problems, the computing time, and the quality of the solution. The proposed approach has the potential to significantly improve the performance of existing SMT solvers and can have a wide range of applications.

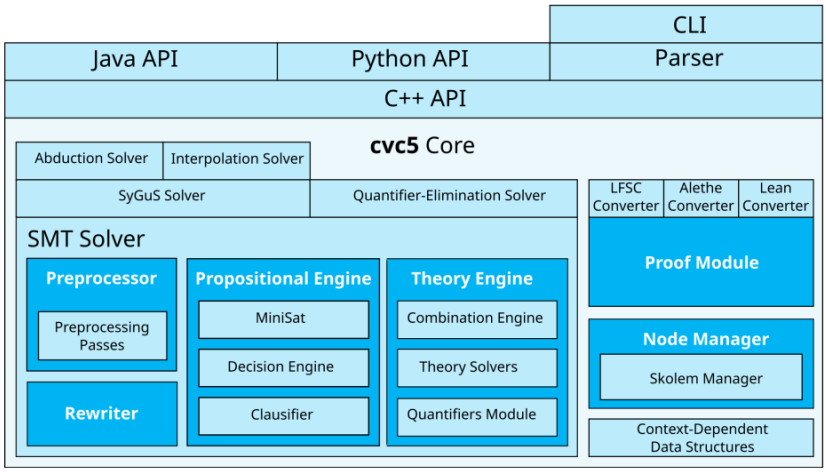



**Figure 1** High-level overview of cvc5's system architecture.

From the diagram of the components provided above[2], you could see that the original cvc5 solver supports a wide range of theories, including all theories standardized in SMT-LIB. Each theory solver relies on an Equality Engine Module, which implements congruence closure over a configurable set of operators, typically those that belong to the solver's theory. The Equality Engine is responsible for quickly detecting conflicts due to equality reasoning. In addition, all theories communicate reasoning steps to the rest of the system via the Theory Inference Manager. Every theory solver emits lemmas, conflict clauses, and propagated literals through this interface. The Theory Inference Manager implements or simplifies common usage patterns like caching and rewriting lemmas, proof construction, and collection of statistics. Every lemma or conflict sent from a theory is associated with a unique identifier for its kind, the inference identifier, which is a crucial debugging aid.

And the most important part for me is that for non-linear arithmetic problems, cvc5 resorts to linear abstraction and refinement. It uses a combination of independent sub-solvers integrated with the linear arithmetic solver and invoked only when the linear abstraction is satisfiable. One sub-solver implements cylindrical algebraic coverings, while the other sub-solvers are based on incremental linearization. A variety of lemma schemas are used to assert properties of non-linear functions (e.g., multiplication and trigonometric functions) in a counterexample-guided fashion. Non-linear integer problems are solved by incremental linearization and incomplete techniques based on reductions to bit-vectors.[3]

And I am going to test, if it is possible to reduce the complexity of architecture non-linear arithmetic problems, developing a simple but at the same time efficient approach that will test all integer values in the bounded interval.

# Background

Satisfiability Modulo Theories (SMT) solvers have been the subject of significant research over the years, with many different approaches and techniques explored. One of the most popular and widely used SMT solvers is cvc5 [4], which has been developed by a team of researchers led by Clark Barrett, Cesare Tinelli, and Morgan Deters since the mid-2000s.

cvc5 is known for its versatility and support for a wide range of theories, including the theory of non-linear arithmetic with integers. It uses a combination of decision procedures and heuristics to efficiently solve complex mathematical problems. In addition, cvc5 supports several input languages, including SMT-LIB and SMT-LIB2 [5], which are widely used in the SMT community.

One area of research in SMT solvers is the problem of solving non-linear integer constraints. This is a challenging problem due to the non-linearity of the constraints, which can lead to undecidability. There are a few techniques for solving non-linear integer constraints using SMT solvers, including linearization, bounded solving, and incremental redundancy encoding [6]. Also in recent years, there has been a significant research effort aimed at the usage of machine learning techniques to guide the search for solutions. For example, researchers have developed neural network-based models to predict the outcome of SMT queries, which can be used to guide the search for solutions and improve the performance of SMT solvers [7].

Linearization involves transforming non-linear constraints into linear constraints, which can then be solved using standard linear programming techniques. This approach has been shown to be effective in some cases, but it can also lead to a loss of precision and is not always practical.

Bounded solving involves adding finite bounds to all variables, which makes the problem decidable and significantly improves the performance of SMT solvers. Researchers have developed various techniques for bounding variables, including interval constraint propagation and divide-and-conquer techniques.

Interval constraint propagation involves iteratively refining the bounds of each variable based on the current bounds of the other variables. This technique has been shown to be effective in solving non-linear integer constraints. Divide-and-conquer techniques involve dividing the problem into smaller sub-problems that can be solved more efficiently.

Overall, the development of efficient SMT solvers for non-linear integer constraints is an active area of research, and researchers are constantly exploring new techniques to improve their performance. The use of SMT solvers has proven valuable in various fields, including computer science, mathematics, and engineering, and the continued development of efficient SMT solvers will likely have a significant impact on these fields.

In this work, I aim to develop a simple SMT solver that tries all values in a specific interval for bounded non-linear integer constraints and compares its performance with that of the original cvc5 solver using the benchmarks provided by cvc5. By doing so, I hope to identify cases where my solver can improve the quality of the results and potentially decrease the computing time.

# Goals

The main goal is to write an SMT solver for an input format smt-lib-version 2.6, for QF_NIA logic. Solver will start with some abstract small interval and check all possible combinations of integers to check if the constraint is satisfiable or not.

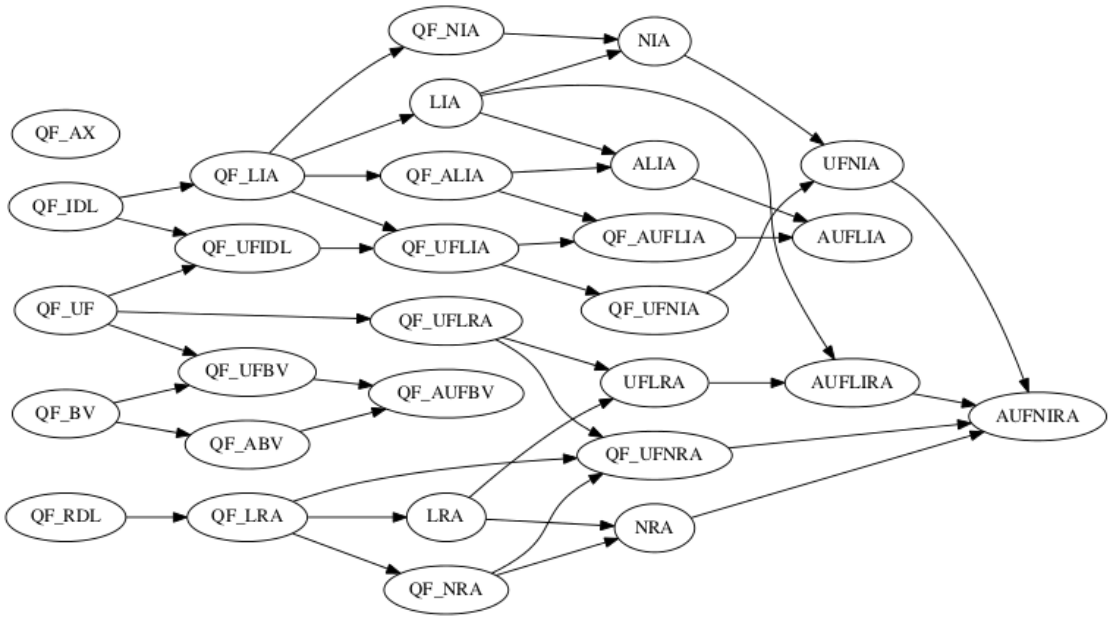



**Figure 2** SMT-LIB logic[5]

The logics have been named using letter groups that evoke the theories used by the logics and some major restrictions in their language, with the following conventions[8]:

- QF for the restriction to quantifier-free formulas
- A or AX for the theory ArraysEx
- BV for the theory FixedSizeBitVectors
- FP (forthcoming) for the theory FloatingPoint
- IA for the theory Ints (Integer Arithmetic)
- RA for the theory Reals (Real Arithmetic)

- IRA for the theory Reals_Ints (mixed Integer Real Arithmetic)
- IDL for Integer Difference Logic
- RDL for Rational Difference Logic
- L before IA, RA, or IRA for the linear fragment of those arithmetics
- N before IA, RA, or IRA for the non-linear fragment of those arithmetics
- UF for the extension allowing free sort and function symbols
- FP (forthcoming) for the theory FloatingPoint
- IA for the theory Ints (Integer Arithmetic)
- RA for the theory Reals (Real Arithmetic)
- IRA for the theory Reals_Ints (mixed Integer Real Arithmetic)
- IDL for Integer Difference Logic
- RDL for Rational Difference Logic
- L before IA, RA, or IRA for the linear fragment of those arithmetics
- N before IA, RA, or IRA for the non-linear fragment of those arithmetics
- UF for the extension allowing free sort and function symbols

# Method

The initial approach involves defining an interval, typically with a limited or small range, within which the solver will try all possible integer values for the given variables. If it is determined that the interval has a boundary, it may be shortened accordingly. However, it should be noted that this approach has an asymptotic complexity of the interval width raised to the power of the number of variables, which is denoted as $O(w^n)$. Here, $w$ represents the interval width, and $n$ represents the number of variables.

After initial computations, it became clear that the interval was too small to produce satisfactory results, even when increased slightly. Additionally, the issue of comparing different metrics arose. While the original cvc5 solver did not impose interval restrictions, it did set a timeout period of one minute, which was later increased to three minutes. As a result, a new approach was taken in which a small interval range was provided, along with a timeout period, to allow the solver to search for a solution within the given parameters. If a solution was found within the interval range, the job was considered complete. If a solution was not found within the range, but the timeout period had not elapsed, the interval range was increased and the solver was tasked with searching for new values. This process was repeated until either a solution was found or the timeout period was exceeded.

# Implementation

*This chapter focuses on the development process of a simple SMT solver.*

## 0.1 Introduction

The practical part of this Bachelor's thesis aims to implement a simple Satisfiability Modulo Theories (SMT) solver for integer non-linear constraints. All SMT solvers consist of three logical parts: lexer (lexan), parser, and Abstract Syntax Tree (AST). The lexer reads the input string character by character and tokenizes it into a stream of meaningful tokens. These tokens are then passed to the parser, which validates the input and generates an AST. The AST represents the input as a structured tree, making it easier to reason about and manipulate. Finally, the constraints are evaluated using the AST. In this practical part, I will develop each of these logical parts and integrate them to create a working SMT solver. To provide a clear overview of how the parts work together, I will present a simple diagram to illustrate the interactions between the lexer, parser, and AST.

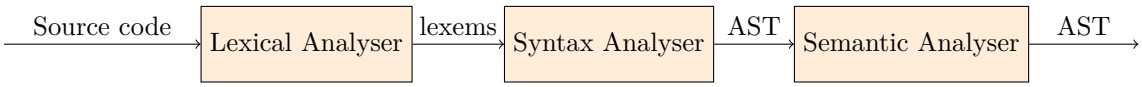



**Figure 3** Logical components of a solver

Additionally, the structure of the project looks the following way:

```
code
├── src ..... Directory with all source files.
│   ├── analysis.ipynb ..... Files with evaluation of all results
│   ├── input.cpp ..... Reading input files
│   ├── input.h .....
│   ├── lexan.cpp .....
│   ├── lexan.h .....
│   ├── main.cpp .....
│   ├── Makefile ..... Building the project
│   ├── parser.cpp .....
│   ├── parser.h .....
│   ├── tree.cpp .....
│   └── tree.h .....
└── test ..... Directory with all tests.
```

Those are only key components and their short description, without the cvc5 solver itself, bash scripts for running the original Linux-based cvc5 solver, and CSV files with the results. But everything is available in the public repository - `https://github.com/burbyleonid/BIE-BAP`

## 0.2 Development

In this section, I will provide a detailed explanation of each part of my solver.

1. input.h and input.cpp - is an implementation of the input handling functionality for a C++ program. The main task of this functionality is to read input data from either a file or standard input and pass it to the program. The code defines a maximum line length of 257 characters and provides a statically defined array of symbols to store individual lines.

   The code starts by checking if a file name is passed to the program or not. If the file name is not provided, it initializes the input to the standard input. However, if the file name is provided, it tries to open the file in read mode, and if successful, it sets the input to the file. In case the file is not found, an error message is displayed, and the function returns 0.

   The function getChar() is responsible for reading a single symbol from the input. It first checks if the pointer to the current line has reached the end. If the pointer has reached the end, it reads a new line from the input file and sets the pointer to the beginning of the new line. It also updates the line number and checks if the last character of the line is a new line character or not. If it is not a newline character, it sets the extended line flag to true.

   Overall, this implementation of input handling provides a robust way to read input data from a file or standard input.

2. lexan.h and lexan.cpp - is responsible for reading input from the source file and producing a sequence of tokens that represent the lexical structure of the input. The code includes the definition of the input character types such as LETTER, NUMBER, WHITE_SPACE, PIPE_LINE, END, and NO_TYPE. It also includes the definition of an array of symbol names and data types. The code consists of several functions that read input and classify it based on its type. The readInput() function reads a single character from the input file and identifies its type. The keyWord() function checks if a given identifier matches any of the reserved keywords in the language and returns its associated symbol type. The error() function is used to report errors during lexical analysis.

3. main.cpp - evaluates expressions by traversing an abstract syntax tree. The program is able to read input from either a file or keyboard and utilizes the Lexan and Parser modules to generate the abstract syntax tree. The program utilizes the high_resolution_clock function from the chrono library to determine execution time. The resulting program is checked for satisfiability, and the corresponding results are outputted. Additionally, there is an additional block of code included which creates a file to record the execution time, result, and values for math problems. The program's purpose is to run the developed tool on a large set of provided problems.

4. Makefile - specifies the target, compiler, compiler flags, and source file location, and sets up the build process by specifying the necessary object files and headers. The target is set to "comp" and the C++ compiler used is g++. The "-std=c++17" flag specifies the C++ version to use, "-Wall -pedantic -Wno-long-long" enables warning messages and "-O2" specifies the level of optimization to use. The code also includes rules for compiling object files, linking them to create the executable, and cleaning up object files after compilation. The "run" target is included to execute the built program. Overall, this Makefile serves as a tool for automating the build process of the C++ program and facilitating efficient testing and debugging during the development process.

5. parser.h and parser.cpp - uses recursive descent parsing to parse the input file and build an abstract syntax tree. The code consists of several functions that correspond to non-terminals in the grammar of the programming language. Each function is responsible for parsing a particular non-terminal and calls other functions as necessary to parse its sub-expressions.

   The main function in this code is ExpressionTMP, which is called by several other non-terminal functions. It takes two optional boolean flags, unaryFlag and constFlag, which indicate whether the expression is unary (has only one operand) or constant. The function reads the next lexical symbol from the input and matches it to one of several cases, depending on the type of the symbol. For each case, the function recursively calls itself or other functions to parse the subexpressions of the current expression and constructs a corresponding node in the abstract syntax tree.

   Other functions in this code handle errors that may occur during parsing, such as syntax errors or unexpected tokens. For example, SyntaxError is called when the parser encounters a symbol that is not expected at the current position in the input. CompareError and ExpansionError are called when the parser encounters a symbol that is not compatible with the expected non-terminal or production rule.

6. tree.h and tree.cpp - is an implementation of a tree data structure that is used to represent mathematical expressions. Specifically, the tree nodes can be of type IntConst, VarNode, or BinOp, where IntConst represents an integer constant, VarNode represents a variable, and BinOp represents a binary operation such as addition, subtraction, multiplication, or division. The BinOp node also contains methods for generating code for the expression and for computing constraints on the variables involved in the expression. The getConstraints() method computes the constraints on the variables involved in the binary operation, given the constraints on the variables in its left and right sub-trees. For instance, the case for multiplication computes the minimum and maximum values of the product of the minimum and maximum values of the constraints on the variables in its left and right sub-trees. The computed constraints can then be used to optimize the computation of the expression, subject to the constraints.

7. analysis.ipynb - is a Python Jupyter notebook, which reads comma-separated files with results of my original cvc5 solver as pandas data frames and compares results and execution time.

## 0.3 Discussion

The practical part presented the results of testing a solver and compared its performance with the original cvc5 solver. In this discussion, I will outline the results and provide directions for further research.

Overall, the findings described in each run suggest that my solver is more adept at handling challenging problem sets, relative to the original cvc5 solver. The results provide valuable insights into the performance of the solver and its comparison to the original cvc5 solver. However, further research is needed to optimize the solver's performance and evaluate its efficacy in solving more complex constraints.

Future research could include evaluating the solver's performance on larger and more complex problem sets, refining the solver's algorithm to improve its performance, and comparing the solver's performance to other popular constraint-solving software. Additionally, the reader could explore ways to optimize the solver's parallel computation to further reduce the runtime. These directions for further research will help improve the efficiency and effectiveness of the solver for constraint-solving applications.

# Computational Experiments

## 0.4 Benchmarks

The benchmarks were taken from the official repository: `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/tree/master/20220315-MathProblems`.

1. STC_0001.smt2 - STC_1000.smt2 - a collection of problems related to the Sum of Three Cubes. This set of benchmarks was chosen because it is easy to parse and contains simple math operations. The complexity of the problems varies, but all can be solved within a reasonable amount of time.
2. MC_01.smt2 - a benchmark for the Magic Square of Cubes problem. This benchmark was chosen because it is a challenging problem that requires the solver to find solutions for a set of equations that involve the sum of cubes of integers. The problem is known to be difficult to solve and requires sophisticated algorithms.
3. MF_01.smt2 - a benchmark for the Magic Five problem. This problem involves finding integer solutions to equations that involve the fifth power of integers. It was chosen because it is a challenging problem that requires solvers to handle high degrees of polynomials.
4. MQ_01.smt2 - a benchmark for the Magic Quadratic problem. This problem involves finding integer solutions to equations that involve the quadratic form. It was chosen because it is a challenging problem that requires solvers to handle quadratic equations and inequalities.
5. MS_01.smt2 - a benchmark for the Magic Square problem. This problem involves finding integer solutions to equations that involve the sum of integers in a square. It was chosen because it is a well-known problem that has been extensively studied in mathematics.
6. RTA_01.smt2 - a benchmark for the Relaxed Taxicab Number problem. This problem involves finding rational solutions to equations that involve relaxed versions of triangular numbers. It was chosen because it is a challenging problem that requires solvers to handle fractions and triangular numbers.
7. SC_01.smt2 - a benchmark for the Semi-Magic Square of Cubes problem. This problem involves finding integer solutions to equations that involve the sum of cubes of integers. It was chosen because it is a classic problem in number theory.
8. SF_01.smt2 - a benchmark for the Sum-Free problem. This problem involves finding integer solutions to equations that involve the sum of integers with the restriction that no three numbers sum to zero. It was chosen because it is a well-known problem that has been extensively studied in combinatorics.

9. SQ_01.smt2 - a benchmark for the Semi-Magic Square of Fourth Power problem. This problem involves finding integer solutions to equations that involve the sum of the fourth powers of integers. It was chosen because it is a classic problem in number theory.

10. TA_01.smt2 - a benchmark for the Taxicab Number problem. This problem involves finding integer solutions to equations that model the sum of two cubes in two different ways. It was chosen because it is a classic problem in number theory.

### 0.4.1 First run

During the first run of my solver, I set the interval from -100 to 100 to test my assumption that the solver would work correctly. I designed the solver such that if the constraints were satisfiable, it would print the "sat" and values, and if not, it would print "not sat." There were no other options available. I ran a total of 1000 STC tests during this run, and the total run time was 709 seconds.

Following this initial test, I wrote a simple bash script, which called the Linux executable file of the original cvc5 solver with a timeout of one minute. I set this timeout to test that everything was working correctly, and it took a total of 55544 seconds or almost 15 hours. This indicates that cvc5 is not able to solve many constraints within one minute.

I then compared the results from the original cvc5 solver with those from my solver, and there was a match in **144 tests**. To ensure that everything was working correctly, I manually checked some of the values.

Overall, these tests provide valuable insight into the performance of my solver and its comparison to the original cvc5 solver. These results will be useful in the further development and optimization of my solver.

### 0.4.2 Second run

For the second run of my solver, I aimed to expand on the groundwork laid in the initial run by implementing a few key adjustments. Specifically, I chose to widen the range of values within which my solver would operate and also augmented the allowable time limit for the original cvc5 solver. More specifically, I extended the interval of my solver from **-100 to 100**, which was used during the first run, to **-300 to 300** for the second run. This change allowed me to further test my assumption that my solver would perform adequately under more challenging conditions.

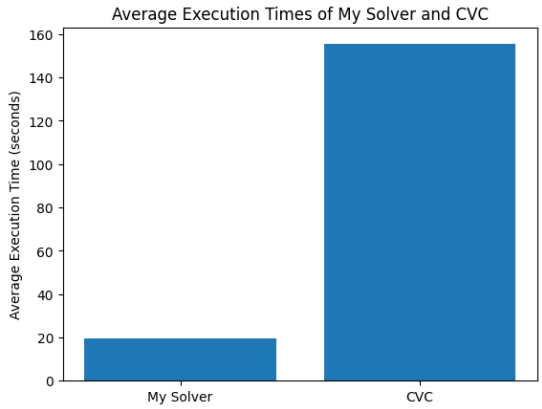
Moreover, to better facilitate the running of tests in the face of potentially long run times, I opted to run calculations in parallel using four identical bash scripts, each operating on a distinct interval (250 tests for 1 script). I calculated that, in the worst-case scenario where I would have to run 1000 tests, it would take approximately **3000 minutes** to complete them all.This duration is clearly quite long, so I concluded that parallel calculations would offer a more expedient approach, thereby allowing me to complete the necessary tests in a more reasonable time frame.

In terms of the specific results, I found that the programs gave the same **"sat"** output in **169 tests**. That guarantees the correctness of my solver, and moreover, my solver was able to compute the **"sat"** values in the **637** tests, while the original cvc5 solver yielded only **169** test. Furthermore, my observations indicated that the original cvc5 solver was still not able to solve a significant number of constraints within the allotted 3-minute time frame, with 831 cases in which it was interrupted by a Timeout value. And at the same time, my solver solved **637** tests with integer values as results, and it is possible to substitute those values to the original constraint to make sure that the result is valid. These findings suggest that my solver is more

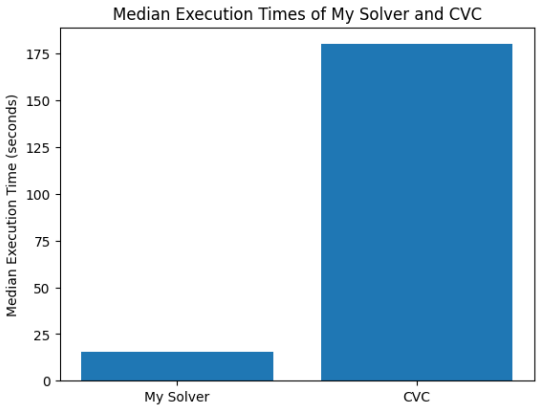
adept at handling STS problem sets, relative to the original cvc5 solver, and further solidifies the promise of my solver as a viable alternative for constraint-solving applications.

But, an important note, is that in the second run, as well as in the first run, I was setting the interval for my solver manually, as well as timeout for the original cvc5 solver. And for that reason, it is technically not correct to compare my solver and cvc5 since they are using different condition metrics, such as linear interval of numbers for my solver, and time in minutes for cvc5. For that reason, further modifications of the code are needed.

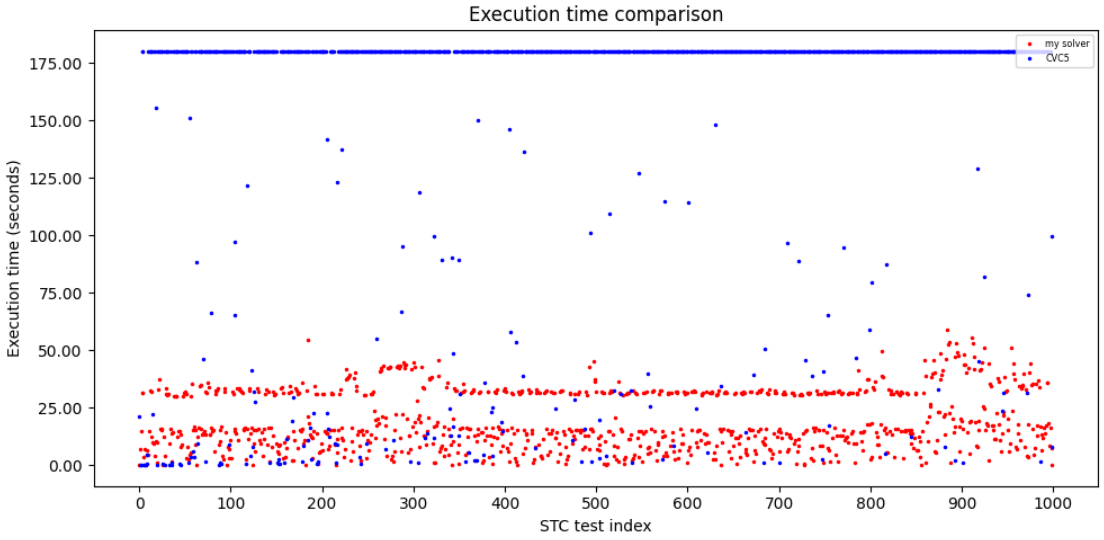
And the data could be visualized in this way:




**(a)** Average time comparison




**(b)** Median time comparison




**(c)** Execution time of each test*

**Figure 4** Execution time comparison of the 4th run

### 0.4.3 Third run, STC tests based on timeout and satisfiability

So, I decided to restructure my code and adopt a slightly different approach. Initially, I set a small interval from **-64** to **64**. If the timeout period has not elapsed, but all values are already checked, I increase the interval by a factor of two on both sides and check for new untested values. For example, I extend from **[-64,64]** to **[-128,128]**, and check only **[-128,-64)** and **(64, 128]**. This process is repeated until the timeout period expires.

Additionally, I use the same benchmarks to evaluate the performance of both solvers. Specifically, I measure the execution time and the result of each solver, allowing me to compare their efficiency and speed. This provides a more reliable and accurate comparison of the two solvers, as they are evaluated on the same parameters.

Here are the raw data (5 lines of head and tail from the resulting data frame) that I obtained:

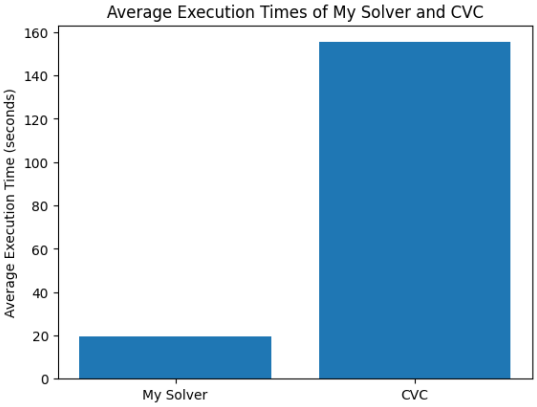
| Test name | Execution time my solver | Result my solver | Values | Execution time cvc5 | Result cvc5 | Comparison |
|---|---|---|---|---|---|---|
| STC_0001.smt2 | 0.03 | sat | x = -300 y = 1 z = 300 | 21.42 | sat | Match |
| STC_0002.smt2 | 7.17 | sat | x = -161 y = -54 z = 163 | 0.26 | sat | Match |
| STC_0003.smt2 | 15.04 | sat | x = -5 y = 4 z = 4 | 0.23 | sat | Match |
| STC_0004.smt2 | 31.52 | not sat | NaN | 180.02 | Timeout. | Mismatch |
| STC_0005.smt2 | 31.54 | not sat | NaN | 180.03 | Timeout. | Mismatch |
| ... | ... | ... | ... | ... | ... | ... |
| STC_0996.smt2 | 18.22 | sat | x = -7 y = 2 z = 11 | 180.02 | Timeout. | Mismatch |
| STC_0997.smt2 | 8.74 | sat | x = -153 y = 65 z = 149 | 180.02 | Timeout. | Mismatch |
| STC_0998.smt2 | 16.57 | sat | x = -9 y = -1 z = 12 | 180.02 | Timeout. | Mismatch |
| STC_0999.smt2 | 7.65 | sat | x = -168 y = 111 z = 150 | 99.80 | sat | Match |
| STC_1000.smt2 | 0.04 | sat | x = -300 y = 10 z = 300 | 8.27 | sat | Match |

Out of all the tests, both solvers gave the same result for 169 of them. That guarantees that the values found by my solver are correct. On average, the original solver took over 2 and a half minutes to solve each problem, while my solver took less than 20 seconds. The median time it took for the original solver to solve each problem was 3 minutes, while my solver took around 15 seconds.

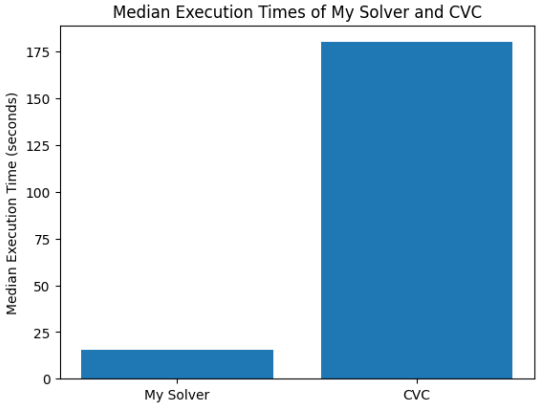
Interestingly, there were 468 cases where the original cvc5 solver was unable to solve the problem within the time limit, but my solver was able to find a solution.

Overall, my solver was able to solve 647 out of 1000 problems, while the original cvc5 solver could only solve 169 of them. This indicates that my approach worked well on the STC test. If it is possible to detect similar types of constraints during the parsing stage, we can test all possible values and potentially find solutions more efficiently.

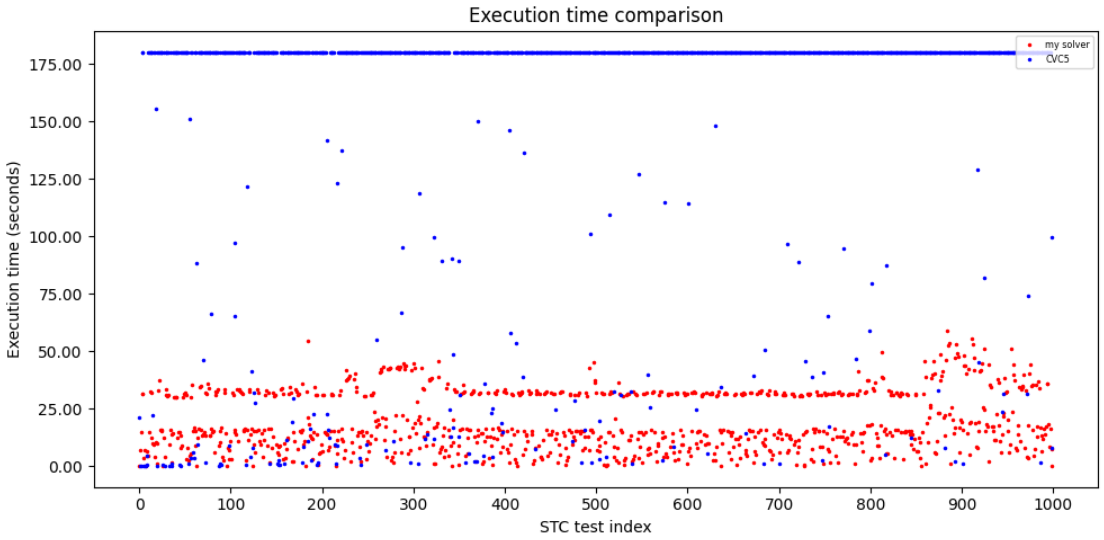
And here is a graphical representation of this run:




**(a)** Average time comparison




**(b)** Median time comparison




**(c)** Execution time of each test

**Figure 5** Execution time comparison of the 3rd run

From the plots above it is visible that on average my solver is able to solve STC problems faster than cvc5.

### 0.4.4 Fourth run

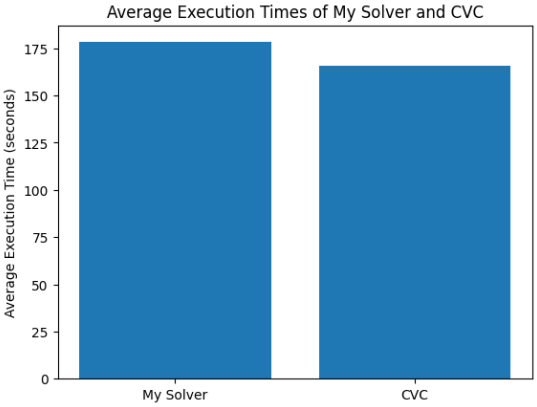
I decided to test the same approach as before but on different tests to confirm or reject the hypothesis that it might be even more efficient in comparison to the cvc5 solver. And this time I run on the rest of the math problems such as MC, MF, MQ, MS, RTA, SC SF, SQ, and TA problems.

Here are the raw data (5 lines of head and tail from the resulting data frame) that I obtained:

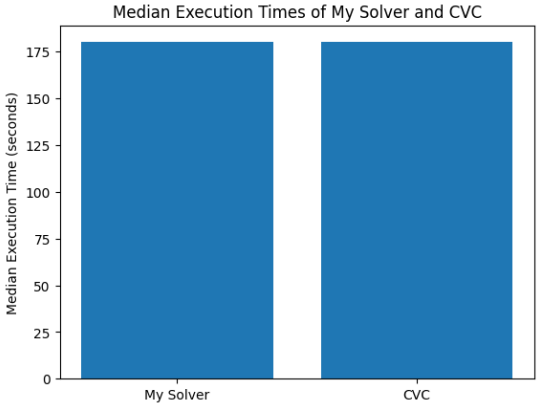
| Test name | Execution time my solver | Result my solver | Values | Execution time cvc5 | Result cvc5 | Comparison |
|---|---|---|---|---|---|---|
| MC_02.smt2 | 180.00 | unsat | NaN | 0.18 | unsat | Match |
| MC_03.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| MC_04.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| MC_05.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| MC_06.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| ... | ... | ... | ... | ... | ... | ... |
| TA_09.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| TA_10.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| TA_11.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| TA_12.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |
| TA_13.smt2 | 180.00 | unsat | NaN | 180.02 | Timeout | Mismatch |

And after analyzing the results, it becomes clear that my solver could not find values for these expressions, like cvc5, but in some examples, cvc5 spent much less time than my solver. And overall, cvc5's combination of efficient algorithms, preprocessing techniques, and theory combination, makes it very fast and effective at detecting the unsatisfiability of the problems.

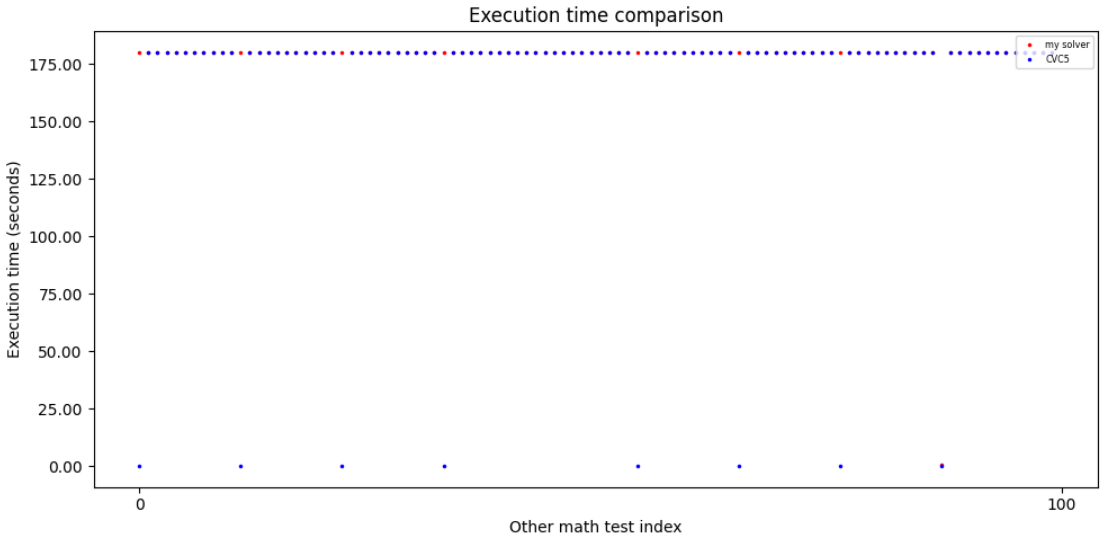
And visualization of results looks in the following way:




**(a)** Average time comparison




**(b)** Median time comparison




**(c)** Execution time of each test*

**Figure 6** Execution time comparison of the 4th run

*In this plot, a lot of dots are located at the same positions, since my solver, as well as cvc5, gave timeout in almost all cases, which is equal to 180 seconds.

From the diagrams above, we could see, that cvc5 has better performance on these tests. But also it is important to mention that there were only 100 such tests, and this is not enough to make such conclusions for sure. So, more experiments are needed on this part.

# Conclusions

1. The solver was developed for bounded non-linear integer constraints based on the trivial "check all values" algorithm. It consists of 3 logical parts, such as Lexical Analyser, Syntax Analyser Semantic Analyser. After I read the constraint as an input file in the format "SMT-LIB Standard: Version 2.6", I parse it, build an abstract syntax tree, simplify and optimize it, and check all possible combinations of values. More detailed implementation is described in the chapter "Implementation" and the source code is stored at the public GitHub repository - `https://github.com/burbyleonid/BIE-BAP`

2. I also discovered that the method of checking all integer values in some intervals might be faster than the original cvc5 solver algorithm but in a very limited constraint type. According to my computation experiments, I found that using this approach in STC problems works definitely better than the original cvc5 solver. And for that reason, it could be integrated as a part of the cvc5 solver to decrease the computational speed of constraints, but only in cases when we could detect STC constraint type during the parsing.

3. TODO Improve the written solver in such a way that it can beat CVC5 in a few more cases.

4. TODO Do systematic computational experiments that document the strong and weak points of the original trivial algorithm, the improved algorithm, and the SMT solver CVC5.

# Appendix A
# Nějaká příloha

Sem přijde to, co nepatří do hlavní části.

# Bibliography


1. NIEUWENHUIS, Robert; OLIVERAS, Albert; TINELLI, Cesare. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*. 2006, vol. 53, no. 6, pp. 937–977.
2. BARBOSA, Renato; BARRETT, Clark; BLAIR, Ryan; KONECNÝ, Jakub; LINSBAUER, Lukas; MOURA, Leonardo de; MÜLLER, Andreas; NIEKUM, Scott; PREINER, Michael; REYNOLDS, Andrew. CVC5: A Versatile and Industrial-Strength SMT Solver. In: *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 391–405. Available from DOI: 10.1007/978-3-030-85448-0_21.
3. BARRETT, Clark; TINELLI, Cesare. CVC4. In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*. Springer-Verlag, 2014, pp. 171–177. Available from DOI: 10.1007/978-3-319-08867-9_24.
4. BARRETT, Clark; FONTAINE, Pascal; TINELLI, Cesare. *CVC4*. Springer, 2018.
5. BARRETT, Clark; FONTAINE, Pascal; TINELLI, Cesare. *The SMT-LIB Standard: Version 2.6*. Association for Computing Machinery, 2017. Available also from: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.
6. DE MOURA, Leonardo; BJØRNER, Nikolaj. Z3: An efficient SMT solver. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340.
7. PIERCE, Benjamin; KODUMAL, Jyotirmoy; GOPALAKRISHNAN, Ganesh. Machine learning for SMT solving. In: *International Conference on Automated Deduction*. Springer, 2019, pp. 26–44.
8. *SMT-LIB Logic Page* [http://smtlib.cs.uiowa.edu/logics.shtml]. [N.d.]. Accessed: May 1, 2023.

# Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF