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
April 10, 2023

Citation of this thesis: Burbygin Leonid. *Bounded Non-Linear Integer Constraint Solving.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 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

In Praze on April 10, 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 an 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. I then 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, efficient, 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. Our 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, we 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

TODO

Conclusion

TODO

Seznam zkratek

SMT	Satisfiability modulo theories
SAT	Boolean satisfiability problem or propositional satisfiability problem
AST	Abstract syntax tree
NP-complete	Nondeterministic polynomial-time complete

Introduction

In this chapter, I describe Motivation, Literature Review, and tested Methods

0.1 Motivation

In recent years, Satisfiability Modulo Theories (SMT) solvers have become an important tool for solving complex mathematical problems in various fields, including computer science, mathematics, and engineering. These solvers are used to solve a variety of problems, including non-linear integer constraints, which involve conjunctions and disjunctions of equalities and inequalities over integers.

While SAT modulo theory (SMT) solvers contain sophisticated algorithms that can solve many instances of non-linear integer constraints, the problem is still undecidable. However, by adding finite bounds to all variables, the problem can be made trivially decidable. This approach can significantly improve the quality of SMT solvers in some cases, potentially decreasing computing time and making them more efficient.

The goal of this bachelor thesis is to develop a simple SMT solver for integer non-linear 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. To accomplish this, benchmarks will be used to compare the performance of the developed solver with the original CVC5 solver.

0.2 Literature Review

CVC5 is a highly sophisticated SMT solver that has been developed for many years by a team of researchers. It is a widely used program in the field of automated reasoning and has a strong reputation for its reliability and efficiency. One of the reasons for its popularity is the availability of comprehensive documentation that covers all aspects of the program's design and implementation.

For the development of a simple SMT solver for integer non-linear constraints that try all values in some interval, the official documentation of CVC5 provides a valuable source of information. It describes the algorithms and techniques used in CVC5 and provides detailed explanations of the various functions and modules of the program. This documentation is a key resource for understanding the inner workings of CVC5 and for developing a custom solver that can be compared to the original program.

In this work, I rely on the official documentation of CVC5 as the main source of knowledge for the development of my own solver. I use the benchmarks provided by CVC5 to compare the performance of my solver with that of the original program. By doing so, I aim to identify cases where my solver can improve the quality of the results and potentially decrease the computing time.

0.3 Methods

The initial approach is to take a specified interval, which was set to range from -100 to 100. If it becomes evident from the constraints that an interval has a boundary, the interval is shortened. Once the intervals are defined, the solver tries all possible integer values within the intervals. This approach has an asymptotic complexity of the interval width to the power of the number of variables, i.e. $O(w^n)$, where w is the interval width and n is the number of variables.

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.
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.
3. TODO Write about the complexity and relevance to the research question.

   TODO finish bib-database like this [1]

Practical part

This chapter focuses on the development, testing and evaluation of results.

0.5 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.

TODO finish

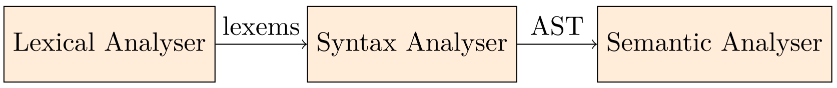

Figure 1 Logical components of a solver

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

TODO - finish, not all files, description?

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

0.6 Development

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

1. TODO finish as in the previous part, add more files, more explanations. add CVC5 solver and link to it

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. analysis.ipynb - is a Python Jupyter notebook, which reads comma-separated files with results of my original CVC5 solver as pandas data frames, and evaluates them. TODO how?

0.7 Data collection, analysis, and results

In this chapter, I would like to report on my experience with running programs and obtaining results from them. Specifically, I would like to focus on the testing of my solver and the comparison of its performance to that of the original CVC5 solver.

0.7.1 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, run_cvc.sh, 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.

0.7.2 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 runtimes, 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 timeframe.

In terms of the specific results, I found that the programs gave the same output in 169 cases. To be more precise, the number of 'sat' values returned by my solver was 637, while the original CVC5 solver yielded only 169 'sat' values. 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 SIGTERM value. These findings suggest that my solver is more adept at handling challenging problem sets, relative to the original CVC5 solver, and further solidifies the promise of my solver as a viable alternative for constraint-solving applications.

0.7.3 Third and fourth run

TODO - run different tests like MC_01.smt2 - a benchmark for the Magic Square of Cubes problem on different conclusions.

0.8 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.

Discussion

Conclusions

Appendix A
Nějaká příloha

Sem přijde to, co nepatří do hlavní části.

Bibliography

1. ŠESTÁKOVÁ, Eliška; JANOUŠEK, Jan. Automata Approach to XML Data Indexing. *Information*. 2018, vol. 9, no. 1. ISSN 2078-2489. Available from DOI: `10.3390/info9010012`.

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
```