Guiding Local Search using Approximations

Jesper Magnusson
Abstract

Guiding Local Search using Approximations

Jesper Magnusson

Proving that a program is correct can be done by translating it into a first-order formula and trying to prove that it is valid. Programs often contain data structures such as Floating-point numbers, for which current solvers struggle because of the computational complexity of these theories. By using approximations, the precision of the floating-point numbers can be reduced to lower the complexity making it easier for the solvers. If a solution to the approximate formula can be found, it is often close to a solution of the original formula. A reconstruction of this solution can be made by modifying it in different ways as an attempt to reconstruct it into a solution of the original formula.

In this thesis a local-search algorithm is implemented as a reconstruction. By continuously keeping a candidate solution, starting with the approximate one it is possible to iteratively make small modifications to search for nearby candidates and eventually find a solution to the original formula. Three different configurations are implemented, evaluated against each other and also against existing reconstructions. Tests shows that a local-search reconstruction can be viable and opens up for further testing of different configurations.

Handledare: Aleksandar Zeljić and Peter Backeman
Ämnesgranskare: Philipp Rümmer
Examinator: Olle Gällmo
IT 18 030
Tryckt av: Reprocentralen ITC
Contents

1 Introduction 2

2 Background 3
  2.1 The Satisfiability Problem ........................................... 3
  2.2 Satisfiability Modulo Theories ..................................... 3
  2.3 Floating Point Arithmetic (FPA) .................................... 5
  2.4 Local Search ........................................................... 6
  2.5 Related Work .......................................................... 9
    2.5.1 Decision Procedures for Floating Point Arithmetic .... 9
    2.5.2 SAT-specific local-search .................................... 9
    2.5.3 Stochastic Local-Search for SMT ............................ 9

3 UppSAT 10
  3.1 Reconstruction in UppSAT .......................................... 12

4 Local Search Reconstruction 13
  4.1 Generating Neighbors ................................................ 14
    4.1.1 Modifying Bits in Floating Point Numbers .............. 15
  4.2 Order by Score ...................................................... 17
  4.3 Picking a New Candidate Model ................................... 17
  4.4 End of Search ....................................................... 18
  4.5 Implementation .................................................... 18

5 Evaluation 19

6 Future Work 21
  6.1 More Efficient Probing ............................................. 21
  6.2 Score Calculation .................................................. 22
  6.3 Other Local-Search Algorithms .................................. 22

7 Conclusion 23

8 References 25

Appendices 27

A Test results 27
1 Introduction

When talking about verification and proving the correctness of programs, Satisfiability Modulo Theories is often mentioned as a tool to check this [1]. By using preconditions, postconditions and assertions in a program, an SMT-formula can be built which is unsatisfiable if and only if the program is correct.

The problem of constructing variable assignments for SMT-formulas [1] to make the formula satisfied or prove that it is unsatisfiable is addressed by multiple solvers such as Z3 [5] and MathSAT [3]. However these solvers struggle with certain theories, such as non-linear reals, bit-vectors and floating-point arithmetic because of the high computational complexity. UppSAT [16] is a framework that approximates formulas of these theories in order to find a solution.

UppSAT uses the previously mentioned solvers as back-ends to receive a solution to the approximate formula. If the approximate solution is also a solution to the original formula, the procedure is done. Otherwise, instead of doing another approximation one can use the obtained approximate solution and modify it somehow in an attempt to construct a solution to the original formula. This is called a reconstruction of the approximate model.

In this thesis a local search [10] algorithm is implemented as a reconstruction to see if this can be an efficient way to obtain a solution to the original formula. This is done by looking at variables that are causing certain Boolean atoms to fail in the formula and modifying them to yield new variable assignments. These can be used as a search space to find a model of the original formula.

Different local search algorithms are explored to analyze whether a specific method is appropriate for the reconstruction. Based on this research one method is implemented and tested with multiple parameter configurations to obtain an algorithm that works well for the purpose. The different configurations are also tested against previous reconstruction implementations to evaluate if local-search works well in this context.

The implemented method seems to be efficient for certain formulas, for which it manages to find models. However while the local-search algorithm can find a model in a few iterations with an approximate model, evaluation shows that refining precision and letting a back-end solver work often obtains a model in a shorter time. The local-search algorithm shows potential but to make the reconstruction more efficient the probing of variable assignments needs
work.

2 Background

This Section gives a brief description of concepts that are used in the thesis.

2.1 The Satisfiability Problem

The Satisfiability problem, often shortened SAT is the problem to decide if for a given propositional formula there exists an assignment of the Boolean variables to make the formula evaluate to true [4]. A variable assignment that satisfies the formula is called a model.

A formula in SAT consists of a conjunction of clauses, where a clause is a disjunction of Boolean variables or their negations. Below is an example of a SAT-formula.

Example 2.1.

\[ A \land (B \lor C) \land D \]

By setting the values of the Boolean variables to either true or false, one can construct a variable assignment for the formula. If this assignment makes the formula evaluate to true, the formula is satisfied and the assignment is a model of the formula. In Example 2.2 one variable assignment is set, namely \( A, C \) and \( D \) are true while \( B \) is false.

Example 2.2.

\[ A \land (B \lor C) \land D \]

By looking at the true variables (bold) it is possible to see that each clause will evaluate to true, which means that the formula is satisfied. This variable assignment is thus a model to the formula.

2.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) can be seen as an instance of the SAT-problem where some variables have been replaced by predicates over some background theory [1]. A predicate can for example be a linear inequality.
or an equality. The predicates are in respect to theories, such as Floating Point Arithmetic or Bit-Vector Arithmetic. These theories have certain rules on how numbers are represented and how they are modified. By giving the variables in Example 2.1 underlying predicates or functions, an SMT-formula could look as follows.

**Example 2.3.**

\[
\begin{align*}
A & : (c = a + 1.0) \\
B & : (c = 3.875) \\
C & : (c = 4.375) \\
D & : (a = 2.875) \\
\end{align*}
\]

\[ (c = a + 1.0) \land (c = 3.875 \lor c = 4.375) \land (a = 2.875) \]

The difference compared to SAT is that the Boolean variables have underlying predicates or functions that can contain values of real numbers, integers or even other data structures. To find a solution to the formula in Example 2.3, one needs to take the underlying theories into consideration. Trying to satisfy this formula in the same way as in Example 2.2, it is possible to see that this is not a solution to the SMT-formula since the underlying predicates cannot hold.

**Example 2.4.**

\[ (c = a + 1.0) \land (c = 3.875 \lor c = 4.375) \land (a = 2.875) \]

\[
\begin{align*}
 & a = 2.875 \implies c = 2.875 + 1.0 = 3.875 \\
 & c = 3.875 \implies c \neq 4.375 \\
\end{align*}
\]

A model to this formula would instead be when \( a = 2.875 \) and \( c = 3.875 \), which evaluates \( A, B \) and \( D \) to true and thus the formula is satisfied. This can be seen in Example 2.5 where the underlying predicates holds.

**Example 2.5.**

\[ (c = a + 1.0) \land (c = 3.875 \lor c = 4.375) \land (a = 2.875) \]

\[
\begin{align*}
 & a = 2.875 \implies c = 2.875 + 1.0 = 3.875 \\
 & c = 3.875 \\
\end{align*}
\]
When reasoning about Floating-Point Arithmetic (FPA) and more precisely about the verification of programs operating on FPA, one wants to decide whether it is possible to satisfy a problem in this domain. A decision procedure is an algorithm that will give a yes or no as an answer to a decision problem [11]. This is the main focus when working on verification for FPA, where approximation has been a tool that these rely on.

The Satisfiability problem (SAT) can be solved with a few different approaches, where local-search is one that has been executed successfully [15]. Lately attempts have also been made on using local-search for Satisfiability Modulo Theories (SMT) [9, 7], which has also been proven quite effective.

2.3 Floating Point Arithmetic (FPA)

A Floating Point Number (FPN) is an encoding of a number that is commonly used as a representation in computers [8].

The structure of a Floating Point Number, as can be seen in Figure 1, consists of a sign, an exponent and a significand.

![Figure 1: Structure of a 64-bit Floating Point number](image)

Much like in scientific notation a number is represented by a significant number that is scaled by the exponent. The radix point can be shifted anywhere in the significand depending on the exponent. The formula for a Floating Point Number can be seen in Equation 1.

\[
\text{exponent} = k - \text{bias} \\
(-1)^{\text{sign}} \cdot \text{significand} \cdot \text{base}^{\text{exponent}}
\]  

(1)

The sign determines whether the number is positive or negative. A 0 means that the number is positive while a 1 represents a negative number.
Since this is independent of the exponent and significand both a positive and negative zero can be represented in FPA.

**The exponent** represents how much the point is shifted in the significand. Since the exponent can be both positive and negative there needs to be a way to represent these numbers. In the IEEE 754 standard [13] a bias is used to be able to represent these. The bias is always the unsigned number with all bits except the most significant one as ones. This means that for a double precision float where there are 11 bits exponent, the bias is 1023. To calculate the exponent one takes the unsigned number k minus the bias. In Example 2.6 a single precision exponent is calculated, which means that the bias is 127.

**Example 2.6.**

\[
\text{exponent} = k - \text{bias} \\
\text{k} = 128 \\
\text{exponent} = 128 - 127 = 1
\]

**The significand** has an implicit 1 as the first bit, with the radix point lying just before the most significant bit. With an exponent value of 0 and a significand of 0 this means that the number will be 1.0. As the exponent shifts the point in the significand, just by increasing the value of the exponent the number will become bigger. In Example 2.7 it is possible to see that the radix point will be shifted based on the value of the exponent. With an exponent of 1 and a significand of all zeros, the number will evaluate to 2.

**Example 2.7.**

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Significand</th>
<th>FPN</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100...0</td>
<td>(1)000...0</td>
<td>10.00...0</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>100...0</td>
<td>(1)100...0</td>
<td>11.00...0</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>100...1</td>
<td>(1)000...0</td>
<td>100.0...0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

### 2.4 Local Search

The following section is about local search algorithms in general, to read more see [10].

Local search is a method that for computationally hard problems tries to find an optimal solution. For these problems only the solution is relevant,
the path it took to get there does not matter. By always working with a
candidate solution it is possible to make local changes to the candidate in an
attempt to get closer to an actual solution. The algorithms move between
candidate solutions until either an actual solution is found or a stop criterion
is met.

Local search algorithms try to find a solution to a given problem by iteratively
modifying a candidate solution in an attempt to find an actual one. This
could be trying to find the maximum or minimum point in a graph, or finding
a satisfying variable assignment for a SAT-formula.

A neighbourhood is a subset of the search space that the algorithm uses when
looking for these solutions, which depends on the type of problem. For ex-
ample in a SAT-problem the search space is all possible variable assignments
and a neighborhood of an assignment could be assignments in which the
values of variables in failing clauses have been flipped.

The Score for a candidate solution could for a SAT-formula be the number
of clauses that are failing. To differentiate the candidates this can be used to
find the solution with the lowest score. In this case a score of 0 would mean no
clauses are failing so the variable assignment is a model of the formula.

It is common that local-search algorithms get stuck in a local optimum,
meaning that it is impossible to find an actual solution continuing from there.
Stochastic Local Search (SLS) algorithms try to solve this problem in differ-
ent ways by using a stochastic mechanism to sometimes make worsening moves.
This means that when getting stuck in a local optimum, these algorithms
can escape from there and find a global optimum instead.

One can divide these algorithms into four categories; Iterative improve-
ment, ‘Simple’ SLS methods, Hybrid SLS methods and Population-Based
SLS methods.

Iterative Improvement is one of the simplest methods, which uses only
the current state solution to try to find a better one. Although they can be
quite simple, they can also be very effective. A problem that often arises
when implementing these is getting stuck in a local optimum. There are
multiple ways in which this can be solved, the simplest one being to make
the neighborhood bigger. Another one could be to sometimes start over with
a completely new candidate solution, called a random restart. Hill-Climbing
is an iterative improvement algorithm that searches for peaks in a graph
which can be fast but can get stuck in a local optimum.
‘Simple’ SLS methods try to solve the same problem that Iterative Improvement-methods solve by using different neighborhoods or random restarts but in a more robust and simple way. This is done by using one search function that in each step finds a new candidate solution and at the same time check if it should be accepted or not. A common type of search function is a randomized or probabilistic one, which evaluates the candidate solution and sometimes accepts a worsening one by using a random or probabilistic model. This means that by using a random variable a solution with a lower score could be accepted. Another way would be to evaluating the new candidate against the old one and look at the difference and a probabilistic value of how probable it is that a worse solution is accepted.

Simmulated Annealing and Tabu Search are two Simple SLS methods where the first one uses a probabilistic model and the other a search history as tools to escape from local optimas.

Hybrid SLS methods take advantage of different Simple SLS methods and combine them to hopefully achieve better performance. Since most algorithms needs to be tweaked for its specific purpose many use at least a combination of two techniques and can thus be called Hybrid SLS methods.

Population-Based SLS methods compared to the other methods uses multiple candidate solutions at the same time. For example while working on a SAT-problem a candidate solution would be a set of variable assignments, which means that implementing a population-based method would work with multiple sets of variable assignments at the same time. The purpose of this is to try widening the search and exploring a bigger search space.

Ant Colony Optimization is a Population-Based SLS method that is inspired by how ants communicate via chemical trails to collectively find a shortest path. This algorithm works with multiple candidate solutions to use the information from all of them to find an actual solution.
2.5 Related Work

2.5.1 Decision Procedures for Floating Point Arithmetic

Reducing floating-point arithmetic to bit-vector arithmetic one can use bit-blasting on the new formula to then give it to a SAT-solver. It is easy to see that a floating point number can be converted to a bit-vector, since in FPA the numbers consists of three bit vectors. Although this reduction can be done efficiently, it still suffers from the problems of bit-blasting with high computational complexity [16].

The ACDCL algorithm successfully solves formulas in FPA by reasoning about the domain of intervals instead of floating-point numbers. It uses interval constraint propagation and case splitting to become an efficient decision procedure for FPA [16].

2.5.2 SAT-specific local-search

Conflict Driven Clause Learning (CDCL) is a widely used approach by SAT-solvers to obtain solutions to the problems. These work by setting values on Boolean variables in the formula and use unit propagation and check for conflicts to make a satisfiable variable assignment [2]. Another approach that has seen more use lately is Stochastic Local-Search(SLS). This technique works by randomly assigning values to variables and by then flipping these values to search for nearby candidate solutions(See Section 2.4).

WalkSAT is a well known architecture of SLS-algorithms specific for SAT. The architecture implements a 2-stage method to choose a variable to flip. In the first stage an unsatisfied clause is chosen and in the second stage a variable is chosen in the clause according to some heuristic [10]. There are multiple WalkSAT-algorithms that build upon this architecture but with small modifications, where some have been successfully effective for solving SAT-problems [14].

2.5.3 Stochastic Local-Search for SMT

Since SLS algorithms have successfully been implemented for SAT-problems, it is also possible that it can be exploited for SMT-formulas. Griggio et al. implements a variant of WalkSAT and combines it with a theory solver to produce WalkSMT [9]. This implementation works on the Boolean atoms of
the formula which does not take into consideration structure of the theories of the formula. Fröhlich et al. describes an implementation that works directly on the SMT-formula, focusing on the theory of bit-vectors. Their research show that SLS could benefit from the structural information that you get when looking at an SMT-formula [7].

3 UppSAT

When trying to satisfy SMT-formulas, there are still areas in which solvers have a hard time constructing models. These areas are mostly theories such as bit-vectors or floating point arithmetic. Using bit-blasting [6] to tackle this problem it quickly becomes a hard computational problem and current solvers have a difficult time to construct models in these domains.

One way to lower the computational complexity is to use approximation, which for floating point as an example could be lowering the number of bits in the exponent and significand. In Example 3.1 full precision floating point numbers are approximated to small floats with 3 bits exponent and 3 bits significand by picking the bits in bold. Doing this will drastically reduce the computational complexity of the theory and solvers will have an easier time constructing models. Even if this does not necessarily construct a model that satisfy the original formula, it should be rather close and there is some information to work with.

Example 3.1.

Using existing SMT solvers such as Z3 [5] and MathSAT [3] as back-ends UppSAT is a framework that uses approximation to work with theories such as floating-point arithmetic, non-linear real arithmetic and bit-vectors [17]. Beginning with an approximation that is easy to solve, UppSAT receives a model to the approximated formula from the back-end solver. With the given model, UppSAT determines the problem unsatisfiable or tries to reconstruct the model into an actual model. If the reconstruction does not work the precision is refined and the formula can be sent to the back-end solver again.
The framework is written in Scala [12] and built of a few important parts, namely encode, checkSAT, decode, reconstruct and refine as can be seen in Figure 2. The unsatRefine part is used if the back-end solver determines the formula unsatisfied. In this thesis only a satisfied instance is relevant which means that this part can be overseen.

*Encode* serves as a translation from one theory to another. As an example a formula with Floating Point Numbers can be decoded to a formula with Real Numbers. This will make the formula easier to solve and works as an approximation tool.

*CheckSAT* gives the encoded model to one of the back-end solvers and checks if it is satisfied or not.

*Decode* receives a model to the approximated formula and translates it into a variable assignment of the original theory.

*Reconstruct* receives a variable assignment from decode and in the case where it doesn’t satisfy the original formula it will try to modify the variable assignment into an actual model. Since it is time consuming to send the formula to a back-end solver, finding a model here is a lot better. This will be described in more detail in Section 3.1.

*Refine* uses the candidate model from the reconstruction in case that the reconstruction did not work. With the failed candidate model and the decoded model, the refining will increase the precision of the formula to receive a more precise model in the next iteration.
3.1 Reconstruction in UppSAT

Given a decoded approximate model from the decoding part of UppSAT, the reconstruction will try to make the decoded model into a model of the original formula. If the received decoded model already is an actual model (model to the original formula), there is no work to be done and the reconstruction can simply return the decoded model. In the other case, it will try to modify the decoded model in an attempt to construct an actual model.

The received approximated model is often close to an actual model. This means that it is quite common that there exists an actual model that is closely related to the decoded model. If the reconstruction manages to find such a model, it will save a lot of time. The other way of getting closer to an actual model is by refining the precision and sending it to a back-end solver again. This is much more time consuming than evaluating a modified model, so obtaining an actual model from the reconstruction is preferred.

There are different approaches to how the reconstruction is done. One good way to start is to look at which predicates (Boolean atoms) are satisfied in the approximate model. These are called critical atoms and is one way to satisfy the formula. Therefore it is a good idea to look at the critical atoms and try to satisfy the reconstructed model in the same way by tweaking some variables. In Example 3.2, critical atoms are in bold. The variable assignment in the approximate model might not satisfy the original formula, but the reconstruction can now try to modify the variables in the critical atoms to satisfy the formula in the same way.

**Example 3.2.**  
\[(c = a + 1.0) \land (c = 3.875 \lor c = 4.375) \land (a = 2.875)\]

As an example, if the formula in Example 3.2 uses Floating Point Arithmetic one way of approximating this would be to use Floats with lower precision as in Example 3.1. With a precision of 3 bits exponent and 2 bits significand the numbers in the example can not be represented. Instead the the numbers will be approximated and decoded as shown in Example 3.3.
Example 3.3.

\[
\text{Approximated Model} \\
c = 4.0 = 0 \ 101 \ 00 \\
a = 3.0 = 0 \ 100 \ 10
\]

\[
\text{Decoded Model} \\
c = 4.0 = 0 \ 10000000001 \ 00000\ldots0 \\
a = 3.0 = 0 \ 10000000000 \ 10000\ldots0
\]

The decoded model is not an actual model, since it will not satisfy the original formula in Example 3.2. In an attempt to construct an actual model one could flip bits.

Example 3.4.

\[
\text{Decoded Model} \\
c = 4.0 = 0 \ 10000000001 \ 00000\ldots0 \\
a = 3.0 = 0 \ 10000000000 \ 10000\ldots0
\]

\[
\text{Reconstructed Model} \\
c = 3.875 = 0 \ 10000000000 \ 01110\ldots0 \\
a = 2.875 = 0 \ 10000000000 \ 01110\ldots0
\]

The decoded model is not an actual model, but looking at the floating point numbers we can see that it is quite close. In Example 3.4 we can see that by flipping the marked bits the decoded model can be turned into an actual model.

4 Local Search Reconstruction

In this thesis a local search algorithm is implemented as reconstruction. As can be seen in Figure 3 the algorithm consists of three big steps, Generating neighbors, Ordering models by score, and Picking a new model. This design is inspired by Iterative Improvement algorithms as described in Section 2.4 by using the current candidate solution to find the next one. When a stop criterion is met a random restart can be made by refining the precision of the formula and getting a new approximate model.
4.1 Generating Neighbors

The neighborhood is defined as a set of variable assignments that are closely related to the approximate model. Given an approximate model neighbors are generated by looking at the failed critical atoms in the model, namely what predicates are failing. Using the failed atoms, new variable assignments are created by modifying variables that are causing the atoms to fail. By modifying the variables in different ways this generates a set of variable assignments that are the neighbors to the previous candidate solution.

In Example 4.1 a formula is approximated, and a model is found. However the approximate model is not a model of the original formula. In the approximate model the satisfied clauses can be seen in bold. These are the critical atoms and thus the variables that occur here will be modified to generate new variable assignments.
Example 4.1.

Original formula
\((c = a + 1.0) \land (c = 3.875 \lor c = 4.375) \land (a = 2.875)\)

Approximate formula (Floating Points Precision(3, 3))
\((c = a + 1.0) \land (c = 3.5 \lor c = 4.0) \land (a = 2.5)\)

Model to the approximate formula
\[ a = 2.5 \\
   c = 3.5 \]

4.1.1 Modifying Bits in Floating Point Numbers

The simplest way of modifying Floating Point Numbers to search for nearby numbers would probably be to find the least significant one in the significand, and flip bits that are close to it. This way it will generate neighbors that are nearby the previous candidate model. As the approximate model provided to the reconstruction is probably quite close to the actual model, it is possible that this will generate solutions that are closer.

In Example 4.2 the least significant one is found in the first step. In the second step it is determined which bits could be flipped and in the third step bits are flipped to generate new variable assignments.

Example 4.2.

\[
\begin{align*}
(1) & \quad 0 1000000000 1000100000\ldots0 \\
(2) & \quad 0 1000000000 1000100000\ldots0 \\
& \quad \text{Flip bits} \\
(3) & \quad 0 1000000000 1000100000\ldots0 \\
& \quad 0 1000000000 1000101000\ldots0 \\
& \quad 0 1000000000 1000100100\ldots0
\end{align*}
\]

It is certainly possible that this kind of modification finds a solution for some problems, but when the formulas become larger (more variables) this will produce many new variable assignments. With too many variable assignments the algorithm will not be that effective anymore since many models will have
the same score and it will be difficult to differentiate them to pick the best one. In this thesis a method is implemented that starts off by generating only a few new variable assignments in the beginning to later generate more. This could reduce the number of variables that are modified early and enabling us to generate a larger neighborhood later on.

A few options of how to modify variables that are causing the critical atoms to fail are described below. It is very difficult to decide what is a good modification, therefore benchmarks of different methods need to be done.

- Look at the precision on the approximation. If a precision with 3 exponent bits and 2 significand bits were used, one could look for rounding errors by flipping bits in the significand after the second bit.
- Similarly looking at the least significant one in the significand, it can be moved either one step to the right or one step to the left. This way it’s possible to search for numbers that are a little bit bigger or smaller.
- For even more precision one could try combinations of adding ones to the x following bits after the least significant one.
- If an equality does not hold, one could try comparing the two numbers and which bits would need to be flipped to make them equal. Then try different combinations of flips to see if there is a possible solution.
- For inequalities, say \( y > x \) for example, a possible way of modifying the numbers would be to set \( y \) to be equal to \( x \) and then flip the first 0 after the last one, making \( y \) bigger than \( x \).
- Use the exponent to modify a number into a smaller one. If the significand is all zeros the number could be made smaller by lowering the exponent and flipping a few bits in the significand as in Example 4.3.

**Example 4.3.**

\[
\begin{array}{c}
0 \ 10000000001 \ 0000000000...0 \\
\downarrow \\
0 \ 10000000000 \ 1110000000...0
\end{array}
\]

In the implementation only the least significand one is looked at to determine how the floating point numbers can be modified. All neighbors are generated which is quite ineffective. Instead of generating all these neighbors another type of searching could be to use a method like Simulated Annealing or Tabu
Search, described in Section 2.4. These algorithms work by picking a new candidate solution and evaluate it against some conditions to see whether it should be accepted or not.

### 4.2 Order by Score

With the generated neighbors, one needs to separate the variable assignments from each other to have some form of measurement of which one is the closest to an actual model. For example one measure of fitness could be the number of failed critical atoms, although this would probably lead to many variable assignments having the same score. In this thesis another method is implemented that calculates a sub score for each predicate, shown in Example 4.4. The sub scores are then summed together to get the final score of the variable assignment. This way it is easier to differentiate between the models and continue to work on the best one.

**Example 4.4.**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Sub score</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x = y))</td>
<td>(</td>
</tr>
<tr>
<td>((x &lt; y, x \leq y))</td>
<td>(\max(x - y, 0))</td>
</tr>
<tr>
<td>((x &gt; y, x \geq y))</td>
<td>(\max(y - x, 0))</td>
</tr>
</tbody>
</table>

When a score is calculated the variable assignments can be ordered by this score, lower score meaning closer to an actual model. A variable assignment with a score of 0.0 means that none of the critical atoms are failing, and thus it is a model of the original formula.

### 4.3 Picking a New Candidate Model

In Section 4.2 different ways of ordering the models according to a score is explained. To make the local-search algorithm efficient the first approach would be to always go with the variable assignment with the best score. This can very well be a good working method of obtaining the next candidate solution. Although as described in section 2.4 it is possible that by doing this one could get stuck in a local optimum.

There are different approaches to avoid getting stuck in a local optimum, where one could be by not always going with the ”best” variable assignment. In this thesis the variable assignment with the lowest score is always chosen
since the approximate model should be quite close to an actual model. If
the algorithm would still get stuck in a local optimum the precision will be
refined and a new approximate model will be obtained, which serves as a
random restart.

4.4 End of Search

If the reconstruction of the approximate model is successful and the local-
search algorithm finds a model for the original formula, the search can in-
stantly terminate and there is no more work to be done. However if a model
is not found, there needs to be some stop criterion to avoid searching for-
ever. In theory the local-search could search for a long time and maybe find
a solution at some point, but when being stuck with the same score for a
while the algorithm could be stuck in a local optimum. At that point it is
better to refine the precision and send the formula back to a back-end solver
again.

Since it is not worth continuing the search when there is no possibility of
finding a solution, a stop criterion is needed. However it is difficult knowing
when that occurs, so there are different methods that could be implemented.
One example of how to set a stop criterion would be to set a maximum
number of iterations in the local-search loop, which is what is used in this
thesis. This is easy to implement and also quite easy to benchmark to see
what could be a good number of max iterations. Another way could be to
look at the score of the current variable assignment and see if the increment
in each iteration is too small.

4.5 Implementation

This Section describes multiple ways of implementing the different parts. In
this thesis three different implementations are tested against each other.

All three implementations generate candidate solutions by looking at the least
significant one in the significand and modifying the numbers accordingly. In
each new candidate solution only one variable has been modified. The first
implementation generates these by creating one new candidate model for
each neighbor. The other two implementations do this by instead creating a
list of candidate values for each variable that should be modified.

To order the models the first two implementations work in the same way,
by evaluating a whole model for each neighbor to get the score and then ordering them by the lowest score. The third implementation does this more efficiently by going through the critical atoms once and calculating the score for all neighbors at the same time.

To pick a new candidate model the one with the lowest score is always chosen and the stop criterion is simply when a certain number of iterations has been made.

5 Evaluation

The local-search reconstruction is implemented in the UppSAT framework written in Scala [12]. Two different algorithms are tested, where one uses a static size of neighborhood and the other one a dynamic size. The first implementation generates neighbors by creating copies of the approximate model with a variable modified. The other implementation instead generates a set of new values for each variable, which can later be evaluated in a model.

To evaluate the different algorithms, they are compared to an implementation with a node-by-node reconstruction. This works by traversing the formula, evaluating the predicates by using the approximate model.

<table>
<thead>
<tr>
<th></th>
<th>IJCAR(node-by-node)</th>
<th>local-search 1</th>
<th>local-search 2</th>
<th>local-search 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solved</td>
<td>67</td>
<td>33</td>
<td>34</td>
<td>47</td>
</tr>
<tr>
<td>Timeouts</td>
<td>85</td>
<td>119</td>
<td>117</td>
<td>105</td>
</tr>
<tr>
<td>Best</td>
<td>44</td>
<td>9</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Average Iterations</td>
<td>3.27</td>
<td>1.22</td>
<td>1.23</td>
<td>2.41</td>
</tr>
<tr>
<td>Max Precision</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Average Rank</td>
<td>3.04</td>
<td>3.71</td>
<td>3.71</td>
<td>3.57</td>
</tr>
</tbody>
</table>

Table 1: Test results

In Table 1 164 different test cases are evaluated and compared with the different implementations. IJCAR(node-by-node) approximates a formula using floats with a lower precision as described in Section 3 but with a simple reconstruction. The local-search implementations use the same approximation but three different local-search reconstructions.

Solved shows how many test cases the implementation manages to solve,
while *Timeouts* shows on how many test cases it times out. *Best* shows in how many test cases that implementation is best out of all four. *Average Iterations* displays the average number of iterations of the approximation framework for all solved test cases. *Max Precision* is the number of test cases in which the framework reached maximum precision. *Average Rank* shows the average rank for each implementation where the rank is 1 if the implementation was best on a certain test case and 4 if it was the worst.

The local-search implementations perform similarly so the different configurations are not important in this context. Comparing these to the simple IJCAR-implementation it is possible to see that this implementation solves the problem fastest in 44 test cases while the local-search implementations are best in 22 test cases in total. This shows that the local-search implementation is able to be faster than a simple reconstruction in some cases but it still struggles in most.

The interesting part is that the average iterations in UppSAT of the ls-1 and ls-2 is close to 1 while IJCAR is over 3 iterations. This shows that by using local-search the number of iterations in the approximation loop can be reduced. However due to the implementation the local-search is often slower than simply refining precision. By making the local-search algorithm more efficient this can prove to be an effective method.

Looking at the max precision that is reached with the different implementations local-search generally reaches max precision in fewer test cases. This shows that local-search can find solutions with a lower precision which can be effective.

<table>
<thead>
<tr>
<th></th>
<th>local-</th>
<th>local-</th>
<th>local-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search 1</td>
<td>search 2</td>
<td>search 3</td>
</tr>
<tr>
<td>Solved</td>
<td>33</td>
<td>34</td>
<td>47</td>
</tr>
<tr>
<td>Timeouts</td>
<td>119</td>
<td>117</td>
<td>105</td>
</tr>
<tr>
<td>Best</td>
<td>12</td>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>Average Iterations</td>
<td>1.22</td>
<td>1.23</td>
<td>2.41</td>
</tr>
<tr>
<td>Max Precision</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Average Rank</td>
<td>2.80</td>
<td>2.79</td>
<td>2.65</td>
</tr>
</tbody>
</table>

Table 2: Test results

In Table 2 the three local-search implementations are compared against each other. The third implementation shows a big improvement in how many test cases that are actually solved. This proves that the calculation of score is
actually much more efficient than in the other two configurations. Although it can solve more test-cases this configuration has a higher number of average iterations. This is because it actually manages to solve the harder cases where the other two does time-out. The third configuration is also best in 24 cases, which is not that surprising since many of these the other two did not even manage to solve.

In Figure 4 the third local-search configuration is compared to the other two. We can see that most instances are solved in the same time but that the third configuration manages to solve a few instances where the other two times out. This shows that because of the different implementation of the scoring calculation a few more instances can be solved. In Appendix A all plots can be seen.

6 Future Work

This Section describes how the implementation could be improved by only making small modifications. Implementing a local-search algorithm takes time since there are many parameters that can be altered and thus, to obtain a good result more benchmarks would be needed.

6.1 More Efficient Probing

The current implementation generates a set of new variable assignments by either copying the approximate model and modifying single variables or by generating a set of new values for each variable. The first implementation is inefficient for larger formulas since it copies and creates many new variable assignments. The second option is more efficient in this sense since it does
not need to make copies of the old model. However it will still need to modify a copy with the new values of variables to evaluate and calculate the fitness.

This implementation of generating a neighborhood works in theory and for small formulas, but for bigger formulas a more efficient method of probing is necessary. One way of doing this would be to calculate how much the fitness would change based on the difference of the old value and the new value of the variable. This method does not need to evaluate each new variable assignment and could thus save time.

While probing the neighborhood other ways of modifying variables should also be tested. Below are a few different approaches that could be viable for the reconstruction.

- Two step method, generating one new model for each variable first. Continuing with models that did not lead to a worse fitness.
- Taking predicate-kind into consideration, doing different modifications if an equality or an inequality.
- Look at rounding mode, to figure out which bits to modify.
- Take precision into consideration to find rounding errors that happened due to the low precision.

6.2 Score Calculation

The way the score is calculated currently is by for each new value of a variable, evaluate the entire formula to calculate the score. This is not very efficient and is the most time consuming part of the algorithm. A better way to calculate the score would probably be to traverse the tree and when a variable is found, the subscore could be calculated for each of the new values at the same time.

6.3 Other Local-Search Algorithms

The iterative improvement algorithm that is implemented could be improved upon by using one of many other local-search algorithms that are proved to be more efficient. For example Stochastic Local-Search algorithms could be implemented such as simulated annealing [10]. Using a randomized algorithm one could evaluate more candidate solutions in a shorter time, which
in some cases is more efficient than always finding the solution with the best score.

For this specific purpose, the reconstruction works by satisfying the same critical atoms. However it is possible that the formula can be satisfied with another set of clauses, which is where an implementation of a WalkSAT [14] algorithm could fit in. With another set of clauses the algorithm could possibly have an easier time searching and finding an actual model of the formula.

7 Conclusion

In this thesis a local-search algorithm is implemented as a reconstruction in the UppSAT-framework, which manages to construct an actual model from an approximate model in some cases.

Three different setups were tested during the implementation, where they differ on how variables are modified, how neighbors are generated and how the score is calculated. For smaller formulas the results showed that all configurations solved instances in an almost equal amount of search steps. Due to how the local-search is built there are many different parameters that can be altered to obtain a new result. To come up with an optimal setup of parameters more benchmarks would need to be made.

In the thesis a few ways of modifying floating point numbers are described, however only three different approaches were tested. One option that was not tested is to look at the predicate and modify numbers accordingly. For an equal-predicate the two sides need to be equal, so modifying numbers in a random way does not make sense. Therefore implementing this should give a big performance boost.

The evaluation of the different local-search algorithms shows that the three configurations perform similarly. Local-search does fit for this purpose however due to how it is implemented in this thesis it performs worse than previously implemented reconstructions. It does seem that the generation of neighbors and the scoring calculation is too costly which makes the algorithm inefficient. By using another type of local-search method this could be made more efficient. A Simple SLS method like Simulated Annealing or Tabu Search seems like it would fit and should be more efficient due to how they work. Instead of generating all neighbors these methods can obtain one
neighbor at random and evaluate it to determine if it should be accepted or not.

The implementation shows potential for small formulas, but struggle when the number of variables become too many. The biggest problem with the implementation is how the candidate solutions are ordered by score, where most of the time is spent. By making this part more efficient the algorithm could prove to be quite good for some formulas.
8 References


Appendices

A Test results